usenix

Proceedings of the 2nd USENIX Conference on Web Application Development

Portland, OR, USA June 15–16, 2011

Sponsored by
usenix

conference

proceedings

2nd USENIX Conference on Web Application Development (WebApps '11)

Portland, OR, USA
June 15–16, 2011

ISBN 978-931971-86-7

USENIX Association

Proceedings of the 2nd USENIX Conference on Web Application Development

June 15–16, 2011
Portland, OR, USA

Conference Organizers

Program Chair
Armando Fox, *University of California, Berkeley*

Program Committee
Adam Barth, *Google Inc.*
Abdur Chowdhury, *Twitter*
Jon Howell, *Microsoft Research*
Collin Jackson, *Carnegie Mellon University*
Bobby Johnson, *Facebook*
Emre Kıcıman, *Microsoft Research*
Michael E. Maximilien, *IBM Research*
Owen O'Malley, *Yahoo! Research*
John Ousterhout, *Stanford University*
Swami Sivasubramanian, *Amazon Web Services*
Geoffrey M. Voelker, *University of California, San Diego*
Nickolai Zeldovich, *Massachusetts Institute of Technology*

The USENIX Association Staff

WebApps '11: 2nd USENIX Conference on Web Application Development
June 15–16, 2011
Portland, OR, USA

Message from the WebApps '11 Program Chair

Welcome to WebApps '11, the second annual USENIX Conference on Web Application Development. Our continuing emphasis is ensuring that attendees are exposed to the most interesting new work from both industry and academia. To that end, both the program committee and the accepted papers represent a balanced mix of industry practitioners from the highest-profile Web companies and university researchers working on cutting-edge Web technologies.

The twelve papers presented (of 28 submissions received) were subjected to the rigorous review standards for which USENIX's academically focused conferences are known. All papers received at least three thorough reviews and some received more; each paper got a fair and complete discussion at the in-person program committee meeting in Berkeley; and each paper was assigned a shepherd to help improve the final presentation. We hope you will be pleased with the results, which showcase a wide variety of Web applications and technologies.

I'd like to thank the authors for taking the time to submit a paper, whether it was accepted or not. Preparing a paper is a lot of work, and we are still exploring ways to engage industrial authors and get the best industrial work along with the best academic work. I personally welcome any suggestions from authors or prospective authors in this regard.

I also thank the program committee for their efforts in reviewing and shepherding, especially some of the industrial participants, whose schedules can be particularly hectic.

Lastly, as always, USENIX's professional organization makes the logistical aspects of running a program committee a breeze, especially Ellie Young, Anne Dickison, Casey Henderson, and Jane-Ellen Long, along with the rest of the USENIX staff.

I hope that what you see and hear at WebApps '11 inspires you to submit your own best work to future WebApps conferences.

Armando Fox, ***University of California, Berkeley***

GuardRails: A Data-Centric Web Application Security Framework

Jonathan Burket Patrick Mutchler Michael Weaver Muzzammil Zaveri David Evans
http://guardrails.cs.virginia.edu
University of Virginia

Abstract

Modern web application frameworks have made it easy to create powerful web applications. Developing a secure web application, however, still requires a developer to posses a deep understanding of security vulnerabilities and attacks. Even for experienced developers it is tedious, if not impossible, to find and eliminate all vulnerabilities. This paper presents GuardRails, a source-to-source tool for Ruby on Rails that helps developers build secure web applications. GuardRails works by attaching security policies defined using annotations to the data model itself. GuardRails produces a version of the input application that automatically enforces the specified policies. GuardRails helps developers prevent a myriad of security problems including cross-site scripting attacks and access control violations while providing a large degree of flexibility to support a range of policies and development styles.

1 Introduction

Web application frameworks have streamlined development of web applications in ways that relieve programmers from managing many details like how data is stored in the database and how output pages are generated. Web application frameworks do not, however, provide enough assistance to enable developers to produce secure web applications without a great deal of tedious effort. The goal of this work is to demonstrate that incorporating data-centric policies and automatic enforcement into a web application framework can greatly aid developers in producing secure web applications.

When developing web applications, developers typically have an idea of what security policies they want to enforce. Ranging from which users should have access to which data to how certain pieces of user input should be sanitized, these policies are rarely documented in any formal way. Code for enforcing security policies is scattered throughout the application, making access checks at a variety of locations or sanitizing strings where they might be potentially harmful. This decentralized approach to security makes it difficult, if not impossible, to be certain that all access points and data flow paths are correctly mediated. Further, security policies are rarely completely known from the start; rather they evolve along with the development of the application or in response to new threats. Changing an existing policy can be very difficult, as changes must be made across the entire application.

To alleviate these problems, we developed *GuardRails*, a source-to-source tool for Ruby on Rails applications that centralizes the implementation of security policies. GuardRails works by attaching security policies directly to data, and automatically enforcing these policies throughout the application. Developers define their policies using annotations added to their data model. GuardRails automatically generates the code necessary to enforce the specified policies. The policy decisions are centralized and documented as part of the data model where they are most relevant, and developers do not need to worry about missing security checks or inconsistent enforcement.

Like most web application frameworks, Ruby on Rails provides an object interface to data stored in database tables. This enables GuardRails to attach policies to objects and make those policies persist as data moves between the application and database. The abstract data model provided by Ruby on Rails is one key advantage over systems like DBTaint [4, 18], which have no knowledge of what the relevant data actually represents within the context of the application. While our implementation of GuardRails is specific to Ruby on Rails, we expect most of our approach could also be applied to similar frameworks for other languages.

Contributions. Our major contribution is a new approach for managing web application security focused on attaching policies directly to the data objects they control. We present a tool that demonstrates the effectiveness of this approach using a source-to-source transformation for Ruby on Rails applications. Although our approach applies to many security vulnerabilities, we focus on two of the most common and problematic security issues for web applications:

- To address *access control violations*, we provide an annotation language for specifying access control policies as part of a data model and develop mechanisms to automatically enforce those policies in the resulting application (Section 3).
- To address *injection attacks*, we implement a context-sensitive, fine-grained taint-tracking system and develop a method for automatically applying context and data-specific transformers. The transformers ensure the resulting strings satisfy the security requirements of the use context (Section 4).

To evaluate our approach, we use GuardRails on a set of Ruby on Rails applications and show that it mitigates several known access control and injection bugs with minimal effort (Section 5).

2 Overview

GuardRails takes as input a Ruby on Rails application with annotations added to define security policies on data. It produces as output a new Ruby on Rails application that behaves similarly to the input application, but includes code for enforcing the specified security policies throughout the application.

Design Goals. The primary design goal of GuardRails is to allow developers to specify and automatically enforce data policies in a way that minimizes opportunity for developer error. Hence, our design focuses on allowing developers to write simple, readable annotations and be certain that the intended policies are enforced throughout the application. We assume the developer is a benevolent partner in this goal—since the annotations are provided by the developer we cannot provide any hope of defending against malicious developers. Our focus is on external threats stemming from malicious users sending inputs to the application designed to exploit implementation flaws. This perspective drives our design decisions.

Another overarching design goal is ensuring that no functionality is broken by the GuardRails transformations and that there is no need to modify an application to use GuardRails. This allows developers to add GuardRails annotations to an application slowly or selectively use parts of GuardRails' functionality.

Ruby on Rails. We chose to target Ruby on Rails because it is a popular platform for novice developers. Rails abstracts many details of a web application including database interactions but does not provide robust security support. There is a need for a tool that abstracts data policy enforcement. Rails uses the ActiveRecord design pattern, which abstracts database columns into ordinary object instances. This makes it very easy to attach policies to data objects. In addition, all field access is through getter and setter methods that can be overridden, which provides a convenient way to impose policies.

Source-to-source. Implementing GuardRails as a source-to-source tool rather than using Ruby's metaprogramming features offers several advantages. Some features of GuardRails, like annotation target inference, could not be implemented using metaprogramming. Location-aware annotations are important because they encourage developers to write policy annotations next to object definitions, which further establishes the connection between data and policies. A source-to-source tool also reduces the runtime overhead incurred. Instead of editing many methods and classes during each execution the changes are made once at compile time. Finally, a source-to-source approach can be more effectively applied to frameworks other than Ruby on Rails. We do use some of Ruby's metaprogramming features as an implementation convenience but we believe that all key aspects of GuardRails could implemented for any modern web framework.

3 Access Control Policies

Access control policies are security policies that dictate whether or not a particular principal has the right to perform a given action on some object. For example, a photo gallery application might have a policy to only allow a photo to be deleted by the user who uploaded it. Poorly implemented access control policies are a major security threat to web applications. Unlike content spoofing and cross-site request forgery, however, access control cannot be handled generically since access control policies are necessarily application-specific and data-dependent.

Traditionally, access control policies are implemented in web applications by identifying each function (or worse, each SQL query) that could violate a policy and adding code around the function to check the policy. This approach rarely produces secure web applications since even meticulous developers are likely to miss some needed checks. As applications evolve, it is easy to miss access control checks needed in new functions. Further, this approach leads to duplicated code and functional code cluttered with policy logic. It is difficult to tell what policies have been implemented just by looking at the code, and to tell if the desired properties are enforced throughout the application.

Policy	Annotation
Only admins can delete User objects	**@delete**, *User*, :**admin**, :**to_login**
Only admins can delete the inferred object	**@delete**, :**admin**, :**to_login**
Users can only change their own password	**@edit**, *pswrd*, **self**.*id* == *user.id*, :**to_login**
Log creation of new User objects	**@create**, *User*, *log_function*; **true**, :**nothing**

Table 1: Access policies and corresponding annotations

To address this problem, we propose a major change in the way access control policies are implemented in web applications. Instead of applying data policies to functions, developers define access control policies as part of the data objects they protect. These *data-centric policies* are specified using annotations added to data model declarations. The code needed to enforce the policies is generated automatically throughout the application. By putting policies close to data and automating enforcement, we reduce the burden on the developer and limit the opportunity for implementation error.

3.1 Annotations

To specify access control policies, developers add *access control annotations* of the form:

@<policytype>, *<target>*, *<mediator>*, *<handler>*

to ActiveRecord class definitions.

The four parameters define: (1) which one of the five data access operations the policy concerns (read, edit, append, create, destroy); (2) the object this annotation concerns (either instances of the annotated class or individual variables within the class); (3) the mediator function that checks if the protected action is allowed; and (4) the handler function that is invoked when an unauthorized action is attempted. If the target field is omitted, GuardRails infers the annotation's target from the location of the annotation (see the second annotation in Table 1).

Keywords are defined for mediators and handlers to define common policies. For example, the :**admin** keyword indicates a mediator policy that checks whether the requesting user has administrator privileges (Section 3.2 explains how developers configure GuardRails to identify the current user object and determine if that user is an administrator). In addition to making it easy to translate policies from a design document into annotations, using keywords makes it easy to define and recognize common policies.

Some policies are application and data-specific. To define these more specific policies, developers use a Ruby expression instead of a keyword. The expression is evaluated in the context of the object being accessed so it can access that object (as **self**) as well as any global application state. The expressions also have access to the current user object, which is made visible to the entire application at the beginning of each execution.

Some examples of data policies are shown in Figure 1. Each annotation would be included as a comment in a data model file above the class definition.

Privileged Functions. Our annotation system is flexible enough to support nearly all policies found in web applications. Policies that depend on the execution path to the protected action, however, cannot be specified because the policy functions do not have access to the call stack. We argue that such policies should be avoided whenever possible—they provide less clear restrictions than policies tied solely to the data and global state (e.g., the logged in user) that can be defined using annotations.

One exception is the common "forgot my password" feature that allows an unauthenticated user to reset their password. To a stateless data policy, however, there is no observable difference between a safe password modification using the "forgot my password" routine and an unsafe password modification.

To handle cases like this, we allow developers to mark functions as privileged against certain policies. To create this password edit policy, the developer annotates the password local variable in the User class with the general policy that a users cannot change a password other their own (see Table 1 for the actual annotation) and adds an annotation to the declaration of the "forgot my password" function to indicate that it is privileged. Since we are assuming developers are not malicious, this simple solution seems preferable to the more complex alternative of providing access to the execution stack in data policy annotations.

3.2 Policy Enforcement

GuardRails enforces access policies by transforming the application. It creates *policy mappings*, objects that map data access operations to mediator and handler functions, for each class or variable. All objects in the generated web application contain policy mappings. By default, policy objects map all accesses to True.

To enforce policies correctly, policies must propagate to local variables. For example, if an ActiveRecord object has a serialized object as a local variable, then edits to the local variable should be considered edits to the ActiveRecord object for policy purposes. To address this we compose the container object's policy with the contained object's policy and assign this new policy to the contained object. Most propagation can be done at compile time but some must be done at runtime since we do not know the complete shape of the data structures.

Once the policy objects have been built, GuardRails modifies the data access methods (generally getters and setters) for each class and adds code to call the appropriate policy function. Because all data access in Ruby is done through getter or setter methods, this is sufficient to enforce data access policies. The code below illustrates how this is done for an example variable:

```
alias old_var= var=
def var=(val)
  if eval_policy(:edit) and
      var.eval_policy(:edit)
    old_var=(val)
  end
end
```

Database Access. No database object should be accessed by the application without first checking its read policy. However, it is not possible to determine in advance which objects will be returned by a given query. This means that we cannot check the read policy before a database access. Instead, GuardRails performs the check *after* the object is retrieved but *before* it is accessible to the application. Since we only want to modify the specific application, not the entire Ruby on Rails framework, we cannot modify the database access functions directly. Instead, we take advantage of the fact that all database accesses are done through static methods defined in the ActiveRecord class.

To enforce access policies on database methods, we replace all static references to ActiveRecord classes with *proxy objects*. These objects intercept function calls and pass them to the intended ActiveRecord class. If the result is an ActiveRecord object or list of ActiveRecord objects, the proxy object checks that the result is readable before returning it to the original caller. This allows GuardRails to enforce access policies for all of the database methods without needing to modify the Ruby on Rails framework.

List Violations. An interesting situation arises when a program attempts to access a list of protected objects, some of which it is not permitted to access. GuardRails supports three ways of resolving this situation: treating it as a single violation for the list object; treating each object that violates the policy as a violation individually; or not handling any violations but silently removing the inaccessible objects from the list. Often, the same choice should be used in all cases of a certain access type so we let developers specify which option to use for each access type. The default choice is to silently remove inaccessible objects from lists, following our goal of providing as little disruption as possible to application functionality.

Configuration. To enable GuardRails to support a wide range of applications, it uses a configuration file to specify application-specific details. For example, in order to give the policy functions access to the current user object, the configuration file must specify how to retrieve this object (typically just a function or reference name). Some built-in policy functions also require extra information such as the **:admin** function, which needs to know how to determine if the requesting user is an administrator. The developer provides this information by adding a Ruby function to the configuration file that checks if a user object is an administrator.

3.3 Examples

We illustrate the value of data policy annotations with a few examples from our evaluation applications (see Table 3 for descriptions of the applications).

Read Protected Objects. The Redmine application contained a security flaw where unauthorized users were able to see private project issues. In response to a request for the list of issues for a selected project, the application returns the issues for the desired project and all of its subprojects with no regard to the subproject's access control list. For example, if a public project included a private subproject, all users could see the subproject's issue list by viewing the parent project's issue list.

Adding the single annotation below to the Issue model fixed this bug by guaranteeing that users cannot see private Issue objects:

```
@read, user.memberships.include? self.project,
    :to_login
class Issue
  ...
```

Note that because of the default policy to silently remove inaccessible items from a list, this policy automatically provides the desired functionality without any code modifications.

Edit Protected Attributes. The Spree application contained a security flaw where users could alter the price of a line item in their order by modifying a POST request to include an assignment to the price attribute instead of an assignment to the quantity attribute. This bug appeared because Spree unsafely used the mass assignment function *update_attributes*. Adding a single GuardRails annotation to the *Line_Item* model prevents the price attribute from being changed:

@edit, *price*, false, :**nothing**

To maintain the behavior of the application, the functions that safely modify the price attribute can be marked as privileged against this policy.

4 Context-Sensitive Sanitization

In general, an injection attack works by exploiting a disconnect between how a developer intends for input data to be used and the way it is actually used in the application. Substantial effort has been devoted to devising ways to prevent or mitigate these attacks, primarily by ensuring that no malicious string is allowed to be used in a context where it can cause harm. Despite this, injection attacks remain one of the greatest threats to modern web applications. The Open Wep Application Security Project (OWASP) Top Ten for 2010 lists injection attacks (in general) and cross-site scripting attacks (a form of injection attack) as the top two application security risks for 2010 [16].

Like access control checking, data sanitization is typically scattered throughout the application and can easily be performed in unsafe ways. To prevent a wide range of injection attacks, including SQL injection and cross-site-scripting, GuardRails uses an extensible system of fine-grained taint tracking with context-specific sanitization. Next, we describe how GuardRails maintains fine-grained taint information on data, both as it is used in the program and stored in the database. Section 4.2 describes how context-sensitive transformers protect applications from misusing tainted data.

4.1 Fine-Grained Taint Tracking

Taint tracking is a common and powerful approach to dealing with injection vulnerabilities that has frequently been applied to both web applications [2, 10, 11, 13, 15, 23, 25, 29] and other applications [1, 7, 9, 12, 17, 20, 22, 28]. A taint-tracking system marks data from untrusted sources as *tainted* and keeps track of how tainted information propagates to other objects. Taint tracking may be done dynamically or statically; we only use dynamic taint tracking. In the simplest model, a single taint bit is associated with each object, and every object that is influenced by a tainted object becomes tainted. Object-level dynamic taint-tracking is already implemented in Ruby.

The weakness of this approach, however, is that it is too simplistic to handle the complexity of how strings are manipulated by an application. When tainted strings are concatenated with untainted strings, for example, an object-level tainting system must mark the entire result as tainted. This leads to *over-tainting*, where all of the strings that interact with a tainted string become tainted and the normal functionality of the application is lost even when there is no malicious data [17]. One way to deal with this is to keep track of tainting at a finer granularity. Character-level taint systems, including PHPrevent [13], Chin and Wagner's Java tainting system [2], and RESIN [29], track distinct taint states for individual characters in a string. This solves the concatenation problem by allowing the tainted and untainted characters to coexist in the final resulting string according to their sources, but requires more overhead to keep track of the taint status of every character independently.

GuardRails provides character-level taint-tracking, but instead of recording taint bits for every character individually, groups sequences of adjacent characters into *chunks* with the same taint status. In practice, most strings in web applications exhibit *taint locality* where tainted characters tend to be found adjacent to each other. This allows GuardRails to minimize the amount of space needed to store taint information, while still maintaining the flexibility to track taint at the character level.

In our current implementation, tainting is only done on strings, meaning that if data in a string is converted into another format (other than a character or a string), the taint information will be lost.[1] We believe this decision to be well-justified, as only tracking strings is sufficient assuming a benevolent developer. A malicious developer could easily lose taint information by extracting each character of a string, converting it to its ASCII value (an integer), and then converting the resulting integers back into their ASCII characters and the original string, but such operations are not likely to be present in a non-malicious application.

Our system only marks string objects with taint information, limiting our ability to track *implicit flows*. GuardRails does not prevent the use of tainted strings in choosing what code path to traverse, such as when the contents of a string play a role in a conditional statement. While it is conceivable that an attacker might manipulate input to direct the code in a specific way, we do not believe this risk to be a large one. On the other hand, tracking implicit flows and preventing the use of tainted strings in code decisions can easily break the existing functionality of many applications. Following our design goals and aim to assist benevolent developers in producing more secure applications, it seems justified to ignore the risks of implicit flows.

4.2 Sanitization

The main distinguishing feature of our taint system is the ability to perform arbitrarily complex *transformations* on

[1]In Ruby, characters are simply strings of length one, so taint information is not lost when characters are extracted from strings and manipulated directly.

tainted strings. Rather than stopping with an error when tainted data may be misused, GuardRails provides developers with a way to apply context-sensitive routines to transform tainted data into safe data based on the *use context*. Each chunk in a taint-tracked string includes a reference to a *Transformer* object that applies the appropriate context-specific transformation to the string when it is used. We define a use context as any distinct situation when a string is used such that malicious input could affect the result. Examples of use contexts include SQL queries, HTML output, and HTML output inside a link tag, but programmers can define arbitrarily expressive and precise use contexts. The transformer method takes in a use context and a string and applies the appropriate context-specfic transformation routine to the input.

If a chunk is *untainted*, its Transformer object is the *identity transformer*, which maps every string to itself in every context. Each *tainted* chunk has a Transformer object that may alter the output representation of the string depending on the context. Taint status gives information about the *current* state of a string, whereas the Transformer objects control how the string will be transformed when it is used.

Our goal is to sanitize tainted strings enough to prevent them from being dangerous but avoid having to block them altogether or throw an error message. After all, it is not uncommon for a benign string to contain text that should not be allowed, and simply sanitizing this string resolves the problem without needing to raise any alarms. As with access policies, GuardRails seeks to minimize locations where developers can make mistakes by attaching the sanitization rules directly to the data itself. In this case, chunks of strings contain their own policies as to how they must be sanitized before being used in different contexts, contexts that are automatically established by GuardRails, as discussed in later sections. Default tainting policies prevent standard SQL injection and cross-site scripting attacks, but the transformation system is powerful enough for programmers to define custom routines and contexts to provide richer policies. Weinberger's study of XSS sanitization in different web application frameworks reveals the importance of context-sensitive sanitization with a rich set of contexts and the risks of subtle sanitization bugs when sanitization is not done carefully [26].

Example. Figure 1 shows a simple default Transformer used by GuardRails. If a chunk containing this Transformer is used in a SQL command, the sanitizer associated with the SQL context will be applied to the string to produce the output chunk. The *SQLSanitize* function (defined by GuardRails) returns a version of the chunk that is safe to use in a SQL statement, removing any text that could change the meaning of a SQL command. Similarly, if the chunk is used within a Ruby eval statement, then it will be sanitized with the *Invisible* filter, which always returns an empty string. In HTML, the applied sanitization function differs based on the use context of the string *within* the HTML. Several HTML contexts are predefined, but new contexts can be defined using an XPath expression. The default policy specifies that if a tainted string appears between `<script>` tags, then the string will be removed via the *Invisible* filter. Elsewhere, the *NoHTMLAllowed* function will only strip HTML tags from the string. The sanitization routines used in the figure (*NoHTMLAllowed*, *BoldTagsAllowed*, etc.) are provided by GuardRails, but the sanitization function can be any function that takes a string as input and returns a string as output. Similarly, the context types for HTML (LinkTag, DivTag, etc.) are predefined by GuardRails, but developers can define their own and specify new contexts using XPath expressions (as shown in the script example).

```
{ :HTML =>
  { "//script" => Invisible,
    :default => NoHTMLAllowed },
  :SQL => SQLSanitize,
  :Ruby_eval => Invisible }
```

Figure 1: Default Transformer

The Transformer approach supports rich contexts with context-specific policies. Contexts can be arbitrarily nested, so we could, for example, apply a different policy to chunks used in an `<img>` tag that is inside a `<div>` tag with a particular attribute compared to chunks used inside other `<img>` tags.

4.2.1 Specifying Sanitization Policies

In many cases, developers want policies that differ from the default policy. Following the format of the data policies, this is done using annotations of the form:

@taint, <*field*>, <*transformer*>

As with the data policies, the *field* component may either be used to specify which field should be marked with this taint status or may be left blank and placed directly above the declaration of a field in the model. The *Transformer* specifies the context hierarchy that maps the context to a sanitization routine within a Transformer object.

One example where such policies are useful is Redmine, a popular application for project management and bug tracking. Its *Project* object has name and description fields. A reasonable policy requires the *name* field to contain only letters and numbers, while the *description* field can contain bold, italics, and underline tags. Redmine uses the RedCloth plugin to allow for HTML-like tags in the Project *description*, but GuardRails makes this both simpler and more systematic allowing developers to use annotations to specify which rules to apply to specific fields. We could specify this using the following annotation:

@taint, {:**HTML** => {:**default** => *BIU_Allowed*}}

This annotation establishes that whenever the string from the *description* field is used in HTML, it should, by default, be sanitized using the *BIU_Allowed* function, which removes all HTML except bold, italics, and underline tags. This :**default** setting replaces the one specified in the default Transformer, but preserves all other context rules, meaning the string will still be removed when used in `<script>` tags, as detailed in Figure 1. If the use context is not already present in the default Transformer, then it will be added at the top as the highest priority rule when a match occurs.

It may be the case, however, that the developer does not want to append to the default Transformer, but overwrite it instead. Adding an exclamation point to the end of a category name specifies that the default rules for this category should not be included automatically, as in the following example:

@taint, {:**HTML**! => {:**default** => *AlphaNumeric*}}

This annotation specifies that in any HTML context the *name* field will be sanitized using *AlphaNumeric*, which removes all characters that are not letters or numbers. As the :**HTML**! keyword was used none of the other HTML context rules will be carried over from the default Transformer. Because the other top-level contexts (such as :**SQL** and :**Ruby_eval**) were not mentioned in the annotation, they will still be included form the default Transformer.

4.2.2 Determining the Use Context

GuardRails inserts additional code throughout the application that applies the appropriate transformers. While our system allows for any number of different use contexts, we focus primarily on dealing with SQL commands and HTML output. We identified all the locations in Rails where SQL commands are executed and HTML is formed into an output page and added calls to the Transformer objects associated with the relevant input strings, passing in the use context.

Many SQL injection vulnerabilities are already eliminated by the original Ruby on Rails framework. By design, Ruby on Rails tries to avoid the use of SQL queries altogether. Using prepared queries when SQL is necessary also helps prevent attacks. Nonetheless, it is still possible to construct a SQL statement that is open to attack. In these scenarios, GuardRails intercepts the SQL commands and sanitizes the tainted chunks using the string's Transformer object.

A more common danger is posed by cross-site scripting vulnerabilities. To ensure that no outgoing HTML contains a potential attack, GuardRails intercepts the final generated HTML before it is sent by the server. The output page is collected into a single string, where each chunk in that string preserves its taint information. GuardRails processes the output page, calling the appropriate transformer for each string chunk. We use Nokogiri [14] to parse the HTML and determine the context in which the chunk is being used in the page. This context information is then passed to the Transformer, which applies the appropriate sanitization routine. The detailed parse tree produced by Nokogiri is what allows for the arbitrarily specific HTML contexts. Note that it is important that the transformations are applied to the HTML chunks in order, as the result of a chunk being transformed earlier in the page may affect the use context of a chunk later in the document. After all of the tainted chunks have been sanitized, the resulting output page is sent to the user. As the entire HTML page must be constructed, then analyzed in its entirety before being sent to the user, this approach may have unacceptable consequences to the latency of processing a request, but could be avoided by more aggressively transmitting partial outputs once they are known to be safe. This is discussed further in Section 5.

4.3 Implementation

GuardRails defines taint propagation rules to keep track of the transformers attached to strings as they move throughout the application. Whenever user input enters the system either through URL parameters, form data, or uploaded files, the content is immediately marked as tainted by assigning it the default Transformer. If the application receives user input in a non-conventional way (e.g. by directly obtaining content from another site), the developer can manually mark these locations to indicate that the data obtained is tainted.

We modify the Ruby String class to contain an additional field used to store taint information about the string. Note that this change is made dynamically by GuardRails within the context of Ruby on Rails and does not involve any modification to the Ruby implementation. A string can contain any number of chunks, so we use an array of pairs, one for each chunk, where the first element represents the last character index of that chunk and the second element is a reference to the corresponding Transformer. For example, the string

`<a href='profile'>`**Joe**`</a>`

generated by concatenating three strings (where underlining represents untainted data and boldface represents tainted data), would be represented using the chunks:

```
[[ 18, <Transformer::Identity>],
 [ 21, <Transformer::Default>],
 [ 25, <Transformer::Identity>]]
```

To maintain taint information as string objects are manipulated, we override many of the methods in the String class, along with some from other related classes such as *RegExp* and *MatchData*. Our goal when deciding how to propagate taint information in these string functions was to be as conservative as possible to minimize the possibility of any exploits. Generally, the string resulting from a string function will be marked as at least as dangerous as the original string. Functions like concatenation preserve the taint status of each input string in the final result, just as one would expect. Other functions like copying and taking substrings also yield the appropriate taint statuses that reflect the taint of the original string. In some cases, discussed in Section 4.3.2, the conservative approach is too restrictive.

4.3.1 Persistent Storage

Web applications often take input from a user, store it in a database, and retrieve and use that information to respond to later requests. This can expose applications to *persistent cross-site scripting* attacks, which rely on malicious strings being stored in the database and later used in HTML responses. Therefore, we need a way to store taint information persistently, as the database is outside of the scope of our Ruby string modifications.

To solve this problem, we use a method similar to that used by RESIN [29] and DBTaint [4]. For every string that is stored in the database, we add an additional column that contains that string's taint information. We then modify the accessors for that string so that when the string is saved to the database, it is broken into its raw content and taint information, and when it is read from the database, both the content and the taint are recombined back into the original string. We also modify several other functions that interact with the database to ensure that the taint information is always connected to the string. This solution makes more sense than serializing the entire object, as it does not disrupt any existing lookup queries made by the application that search for specific text in the database.

4.3.2 Problematic Functions

In our tests, we found that there are some cases where being overly safe can result in overtainting in a way that interferes with the behavior of the application. Our rules are slightly more relaxed in these situations, but only when necessary and the security risk is minimal. Next, we discuss several of these cases and others where determining the appropriate tainting is more complex.

Pattern Substitution. Ruby provides the *sub* and *gsub* procedures that provide regular expression substitution in strings. With these functions, the contents of one chunk affect different parts of the output string in complex ways. If the input string is tainted and the replacement is untainted, then the resulting taint status is ambiguous, as the tainted string affects where the untainted string is placed.

A maximally conservative approach might consider the untainted replacement as tainted, as its location was specifically dictated by the contents of the tainted string. While our is generally to take a conservative approach to tainting, we found in our test applications that this approach frequently leads to overtainting. Hence, we adopt a more relaxed model where output characters are tainted only when they directly result from a tainted chunk. Figure 2 illustrates some examples of how taint information is manipulated in commands such as *gsub*.

Composing Transformers. Another set of special cases are those functions that blend multiple tainted chunks in a way where it is difficult or impossible to keep the resulting taint statuses separate. One such function is *squeeze*, which replaces identical consecutive characters in a string with a single copy of that character (see Figure 2 for examples). If the repeated characters have the same taint status then there is no issue: the resulting single character should also have the same taint status. If, however, each of the repeated characters has a different taint status, the resulting character depends on both inputs. Picking one of the two taint statuses could potentially leave the application vulnerable, so we mix the different taint statues by composing the transformers. A Transformer object can simply be considered a function that takes in a string and a context and returns a sanitized string for that context. This means that we can combine Transformers simply by composing their respective functions. When the composed Transformer is given a string and context, it applies the first Transformer with the given context, then applies the second Transformer with the same context to the result of the first.

As the order in which the two Transformers are applied might affect the final result, we perform the transformations in both possible orders and check that the results are the same. If they are not, then GuardRails acts as conservatively as possible, either throwing an error or emptying the contents of the string. In practice, this issue is not particularly problematic as only a few uncommonly used string functions (*squeeze*, *next*, *succ*, *upto*, and *unpack*) need to compose Transformers and the majority of Transformers produce the same result regardless

String Command	Result
"**foobar**".gsub("o","0")	"**f**00**bar**"
"**medium**".gsub(/(datu|mediu|agendu|bacteriu)m/,"\1a")	"**medi**a"
"uto**pia**".gsub(/(a|e|i|o|u)/) { |x| x.swapcase }	"UtO**pIA**"
"**foot**ball".squeeze	"**fot**bal"
"**bat**tle".squeeze	"**ba*t***le"

Table 2: Example String Commands with Taint
Underlined and bold text in these examples indicate different taint statuses. The second *gsub* example is very similar to the matching used by Ruby on Rails in the *pluralize* function, which converts words to their plural form. The second *squeeze* example demonstrates how taint must be merged in cases where the value of a chunk comes from multiple sources.

of the order in which they are applied.

String Interpolation. One key advantage of taint tracking system employed by GuardRails is that modifies string operations dynamically, with no need to directly alter any Ruby libraries. *String interpolation*, a means of evaluating Ruby code in the middle of a string, is managed by native Ruby C code, however, and cannot be changed dynamically. To resolve this problem, the source-to-source transformation done by GuardRails transforms all instances in the web application where interpolation is used with syntactically equivalent concatenation. Additionally, because Ruby on Rails itself also uses interpolation, GuardRails runs the same source-to-source transformation on the Ruby on Rails code, replacing all uses of interpolation with concatenation.

4.4 Examples

We illustrate how one taint-tracking system eliminates vulnerabilities and simplifies application code by describing a few examples from our evaluation applications.

SQL Injection. Substruct, an e-commerce application, handles most forms safely but one particular set of fields was left vulnerable to SQL injection [21]. The issue lay with the use of the function *update_all*, which performs direct database updates and can easily be used unsafely, as in the following code written by the developers:

```
update_all("value = '#{value}'", "name = '#{name}'")
```

This code directly includes the user-provided *value* string in the SQL command. A malicious user could exploit this to take control of the command and potentially take control of the database.

GuardRails prevents this vulnerability from being exploited. Since *update_all* is known to be a way of passing strings directly into SQL commands, GuardRails modifies the function to transform the provided strings for the SQL use context. As form data is marked automatically with the default Transformer, the potentially harmful strings will be sanitized to remove any dangerous text that might modify the SQL command. Note that for this example, the vulnerability is eliminated by using GuardRails even if no annotations are provided by the developer.

Cross-Site Scripting. Onyx correctly sanitizes input that is used to post comments on images, but does not perform the same checks on input to certain administrator fields. This allows any administrator to inject code into the application to attack application users or other site administrators. In the case of the *site_description* field, simply putting in a double quote as part of the input is enough to break the HTML structure of the resulting pages. These are examples of simple persistent cross-site scripting issues, where the offending string is first saved in the database, then attacks users when later loaded from the database and placed into HTML responses.

Applying GuardRails fixes these cross-site scripting vulnerabilities. Recall that the default Transformer (Figure 1) removes all tags when an unsafe string is used in an HTML context. Thus, when the attacker submits the malicious string in the web form, it is immediately assigned the default Transformer. When that string is later used in HTML, it is sanitized using the *NoHTMLAllowed* filter. The taint information is preserved when strings are saved and loaded from the database (Section 4.3.1), so the vulnerability is not exploitable even though it involves strings read from the database.

5 Evaluation

We conducted a preliminary evaluation of GuardRails by testing it on a variety of Ruby on Rails applications. Table 3 summarizes the test applications. These applications are diverse in terms of complexity and cover a range of application types and design styles. We were able to use GuardRails effectively on all the applications without any modifications.

As detailed in Section 3.3, we have had success at preventing known access control issues with simple policy annotations. Our system of fine-grained taint tracking

Application	Description	Source	Lines of Code
Onyx	image gallery	http://www.hulihanapplications.com/projects/onyx	680
Spree	shopping cart	http://spreecommerce.com/	11561
Substruct	shopping cart	http://code.google.com/p/substruct/	5556
Redmine	project management	http://www.redmine.org/	30747
PaperTracks	publication and citation tracker	developed ourselves	1980

Table 3: Test Applications

also succeeded at blocking SQL injection and cross-site scripting attacks, as explained in Section 4.4.

While performance was not a major design goal, it is still a practical concern. To estimate the overhead imposed by our system, we transformed the image gallery application Onyx with various configurations of GuardRails and measured the average throughput for 50 concurrent users. Table 4 summarizes the results.

As currently implemented, GuardRails does impose a significant performance cost. But, we believe most of this performance overhead is due to limitations of our prototype system rather than intrinsic costs of our approach.

Performing the access control checking decreases throughput by around 25 percent. Most of the performance overhead comes from the code needed to assign policies dynamically. This code is independent of the number of policies, so the performance does not greatly depend on the number of annotated policies. We could reduce this overhead by using static analysis to determine which policies can be assigned statically instead of dynamically.

Taint tracking incurs substantial overhead, reducing throughput by more than 75 percent for some requests. Our taint tracking implementation replaces the native C string implementations provided by Ruby with interpreted Ruby implementations. Since the Ruby string implementations are highly optimized, and interpreting Ruby code is much slower than native C, it is not surprising that this incurs a substantial performance hit. Complex functions like *gsub*, *split*, *delete*, and *slice* require more code to ensure that taint status is handled correctly. The *split* method, for example, took 0.14 seconds to run 400 times without the taint system applied in one test. With the taint system applied, the same test took 0.15 seconds when operating on untainted strings but nearly 5 seconds to split tainted strings. In future work, we hope both to optimize string methods both by rewriting them in C and using more efficient algorithms, and we are optimistic that much of the performance overhead imposed by GuardRails could be eliminated by doing this.

6 Related Work

Much research has been done towards the goal of improving security of web applications and developing access control policies. Here, we review the most closely related work on data policy enforcement and taint tracking.

6.1 Data Policy Enforcement

Aspect-oriented programming is a design paradigm that centralizes code that would normally be spread throughout an application, often referred to as cross-cutting concerns [8]. Data policy enforcement is such a concern and several authors have suggested using aspect-oriented programming to implement security policy enforcement [24, 27]. Like our project, this work seeks to reduce implementation errors and improve readability by centralizing information about security policies.

Automated data policy enforcement is becoming a popular method for preventing security vulnerabilities. Some projects let developers specify data policies, assign the policies to object instances explicitly, and enforce the policies using a runtime system [29, 19]. It is often difficult to define security policies in a clear and concise format. Some projects attempt to remedy this by creating a policy description language [5] while others aim to infer appropriate policies [3] without developer input.

The most similar previous work is RESIN, a tool that enforces developer-specified policies on web applications [29]. GuardRails and RESIN differ in several fundamental ways. RESIN handles policies attached to individual object instances, so developers must manually add the policies on each object instance they want to protect. GuardRails instead associates policies with data models (classes), so the appropriate policy is automatically applied to all instances of the class. Additionally, GuardRails automates much more of the work required to build a secure web application than RESIN does. RESIN requires the developer to write an entire class for each security policy while GuardRails only requires a small annotation.

Transformation Status	Homepage	Login Interface	Image Gallery
Original Application	8.9	9.6	9.2
Access Control Only	7.1	7.1	6.6
Taint Tracking w/o HTML Parsing	2.5	2.8	2.5
Full System	2.0	2.5	2.2

Table 4: Performance Measurements from Onyx
Each number indicates the number of transactions per second for the given request and configuration.

6.2 Taint Tracking

Taint tracking techniques have been used to find format string vulnerabilities [20, 22, 28], prevent buffer overflows [22, 28], improve signature generation [12], and even to track information flow at the operating system level [7]. Several systems, like the GIFT framework [9], are designed, like GuardRails, to be extensible to prevent many types of injection attacks [1, 15]. As mentioned in Section 4.1, some recent research has focused on solving the over/undertainting problem with character-by-character taint tracking [2, 13, 29]. Many systems are limited to using boolean taint states [22, 28] or make use of the compiler, making them difficult to directly apply to a dynamic, interpreted language like Ruby [1, 11].

Similar to our context-specific transformers, the Context-Sensitive String Evaluation (CSSE) [15] system treats tainted strings differently depending on the context of their use. CSSE uses meta-data tags to allow for complex taint statuses. CSSE, however, focuses on propagating information about where the content originated from, with the context-specific code dealing with the tainted strings at the location of their use based on this origin information. The Auto Escape mode in Google's Template System is another similar system that uses different sanitization routines depending on the context of a string in HTML [6]. Without taint-tracking, however, Auto Escape cannot distinguish between safe and unsafe strings without explicit specifications from the developer, so it is necessary to explicitly identify templates that should use auto escape mode.

Other systems do not modify the web application itself or the underlying platform, but instead operate between the application's key entry and exit points. Sekar developed one such tool [18] that records the input received by the application, and later uses *taint inference* in output and database commands to find similar strings that may have been derived from this input. The tool also focuses on looking for changes in syntax of important commands that might be indicative of an injection attack. Another system, DBTaint [4] works outside of the application, helping to preserve arbitrary taint information given from an arbitrary application in the database. Both of these tools have the advantage of being largely platform-independent, and neither needs any application modifications.

7 Conclusion

GuardRails seeks to reduce the effort required to build a secure web application by enforcing security policies defined with the data model, in particular, access control policies and context-sensitive string transformations. The main novelty of GuardRails is the way policies are tied directly to data models which fits developer understanding naturally, provides a large amount of expressiveness, and centralized policies in a way that minimizes the likelihood of missing necessary access control checks. Our early experience with GuardRails provides cause for optimism that application developers can be relieved of much of the tedious and error-prone work typically required to build a secure web application. Although the performance overhead is prohibitive for large scale commercial sites, many web applications can tolerate fairly poor performance. Further, although our current prototype implementation incurs substantial overhead, we believe many of techniques we advocate could be implemented more efficiently if they are more fully integrated into the underlying framework implementation, and that reducing developer effort and mitigating security risk will become increasingly important in rapid web application development.

Availability

GuardRails is available under an open source license from http://guardrails.cs.virginia.edu/.

Acknowledgements

This work was funded in part by grants from the National Science Foundation and a MURI award from the Air Force Office of Scientific Research. The authors thank Armando Fox for his helpful comments and suggestions, and thank Dawn Song, Prateek Saxena, and the attendees at RubyNation for helpful discussions about this work.

References

[1] CHANG, W., STREIFF, B., AND LIN, C. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 39–50.

[2] CHIN, E., AND WAGNER, D. Efficient Character-level Taint Tracking for Java. In *2009 ACM Workshop on Secure Web Services* (2009).

[3] DALTON, M., KOZYRAKIS, C., AND ZELDOVICH, N. Nemesis: preventing authentication & access control vulnerabilities in web applications. In *Proceedings of the 18th conference on USENIX security symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association, pp. 267–282.

[4] DAVIS, B., AND CHEN, H. DBTaint: Cross-application Information Flow Tracking via Databases. In *2010 USENIX Conference on Web Application Development* (2010), WebApps'10.

[5] EFSTATHOPOULOS, P., AND KOHLER, E. Manageable fine-grained information flow. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 301–313.

[6] GOOGLE. Auto escape. http://google-ctemplate.googlecode.com/svn/trunk/doc/auto_escape.html, 2010.

[7] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (New York, NY, USA, 2006), EuroSys '06, ACM, pp. 29–41.

[8] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming* (1997).

[9] LAM, L. C., AND CHIUEH, T.-C. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 463–472.

[10] LIVSHITS, V. B., AND LAM, M. S. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), USENIX Association, pp. 18–18.

[11] NANDA, S., LAM, L.-C., AND CHIUEH, T.-C. Dynamic Multi-process Information Flow Tracking for Web Application Security. In *2007 ACM/IFIP/USENIX International Conference on Middleware Companion* (2007).

[12] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploitson commodity software. In *Proceedings of the 12th AnnualNetwork and Distributed System Security Symposium* (2005), NDSS05.

[13] NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., SHIRLEY, J., AND EVANS, D. Automatically Hardening Web applications Using Precise Tainting. In *Security and Privacy in the Age of Ubiquitous Computing* (2005).

[14] PATTERSON, A., DALESSIO, M., NUTTER, C., ARBEO, S., MAHONEY, P., AND HARADA, Y. Nokogiri: an HTML, XML, SAX, and Reader Parser. http://nokogiri.org/, 2008.

[15] PIETRASZEK, T., AND BERGHE, C. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, A. Valdes and D. Zamboni, Eds., vol. 3858 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 124–145.

[16] PROJECT, O. W. A. S. OWASP Top 10 — The Ten Most Critical Web Application Security Risks. http://www.owasp.org/index.php/Top_10, 2010.

[17] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy (Oakland)* (2010).

[18] SEKAR, R. An Efficient Black-box Technique for Defeating Web Application Attacks. In *16th Annual Network and Distributed System Security Symposium (NDSS)* (2009).

[19] SEO, J., AND LAM, M. S. InvisiType: Object-Oriented Security Policies. In *17th Annual Network and Distributed System Security Symposium* (2010).

[20] SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10* (Berkeley, CA, USA, 2001), SSYM'01, USENIX Association, pp. 16–16.

[21] SUBSTRUCT DEVELOPER. Preference.save_settings is insecure. http://code.google.com/p/substruct/issues/detail?id=36, Mar. 2008.

[22] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2004), ASPLOS-XI, ACM, pp. 85–96.

[23] TRIPP, O., PISTOIA, M., FINK, S. J., SRIDHARAN, M., AND WEISMAN, O. TAJ: Effective Taint Analysis of Web Applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 87–97.

[24] VIEGA, J., BLOCH, J. T., AND CH, P. Applying aspect-oriented programming to security. *Cutter IT Journal 14* (2001), 31–39.

[25] VOGT, P., NENTWICH, F., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '07)* (2007).

[26] WEINBERGER, J., SAXENA, P., AKHAWE, D., FINIFTER, M., SHIN, R., AND SONG, D. An empirical analysis of xss sanitization in web application frameworks. Tech. Rep. UCB/EECS-2011-11, EECS Department, University of California, Berkeley, Feb 2011.

[27] WIN, B. D., VANHAUTE, B., AND DECKE, B. D. Developing secure applications through aspect-oriented programming. *Advances in Network and Distributed Systems Security* (2001), 125–138.

[28] XU, W., BHATKAR, S., AND SEKAR, R. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th conference on USENIX Security Symposium (USENIX-SS '06)* (2006).

[29] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving Application Security with Data Flow Assertions. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009).

PHP Aspis: Using Partial Taint Tracking To Protect Against Injection Attacks

Ioannis Papagiannis
Imperial College London

Matteo Migliavacca
Imperial College London

Peter Pietzuch
Imperial College London

Abstract

Web applications are increasingly popular victims of security attacks. Injection attacks, such as Cross Site Scripting or SQL Injection, are a persistent problem. Even though developers are aware of them, the suggested best practices for protection are error prone: unless all user input is consistently filtered, any application may be vulnerable. When hosting web applications, administrators face a dilemma: they can only deploy applications that are trusted or they risk their system's security.

To prevent injection vulnerabilities, we introduce PHP Aspis: a source code transformation tool that applies *partial taint tracking* at the language level. PHP Aspis augments values with taint meta-data to track their origin in order to detect injection vulnerabilities. To improve performance, PHP Aspis carries out taint propagation only in an application's most vulnerable parts: third-party plugins. We evaluate PHP Aspis with Wordpress, a popular open source weblog platform, and show that it prevents all code injection exploits that were found in Wordpress plugins in 2010.

1 Introduction

The most common types of web application attacks involve code injection [4]: Javascript that is embedded into the generated HTML (Cross Site Scripting, or XSS), SQL that is part of a generated database query (SQL Injection, or SQLI) or scripts that are executed on the web server (Shell Injection and Eval Injection). These attacks commonly exploit the web application's trust in user-provided data. If user-provided data are not properly filtered and sanitised before use, an attacker can trick the application into generating arbitrary HMTL responses and SQL queries, or even execute user-supplied, malicious code.

Even though web developers are generally aware of code injection vulnerabilities, applications continue to suffer from relevant exploits. In 2010, 23.9% of the total reported vulnerabilities to the CVE database were classified as SQLI or XSS [12]. Morover, injection vulnerabilities are often common in third-party plugins instead of the well-tested core of a web application: in 2010, 10 out of 12 reported Wordpress injection exploits in the CVE database involved plugins and not Wordpress itself.

Such vulnerabilities still remain because suggested solutions often require manual tracking and filtering of user-generated data throughout the source code of an application. Yet, even a single unprotected input channel in an application is enough to cause an injection vulnerability. Thus, less experienced and therefore less security-conscious developers of third-party plugins are more likely to write vulnerable code.

Past research has suggested *runtime taint tracking* [19, 18, 14] as an effective solution to prevent injection exploits. In this approach, the origin of all data within the application is tracked by associating meta-data with strings. When an application executes a sensitive operation, such as outputting HTML, these meta-data are used to escape potentially dangerous values. The most efficient implementation of taint tracking is within the language runtime. Runtime taint tracking is not widely used in PHP, however, because it relies on custom runtimes that are not available in production environments. Thus, developers are forced to avoid vulnerabilities manually.

We show that injection vulnerabilities in PHP can be addressed by applying taint tracking entirely at the source code level without modifications to the PHP language runtime. To reduce the incurred performance overhead due to extensive source code rewriting, we introduce *partial taint tracking*, which limits taint tracking only to functions of the web application in which vulnerabilities are more likely to occur. Partial taint tracking effectively captures the different levels of trust placed into different parts of web applications. It offers better performance because parts of the application code remain unchanged.

We demonstrate this approach using PHP Aspis[1], a

[1]An Aspis was the circular wooden shield carried by soldiers in ancient Greece.

tool that performs taint tracking only on third-party plugins by rewriting their source code to explicitly track and propagate the origin of characters in strings. PHP Aspis augments values to include taint meta-data and rewrites PHP statements to operate in the presence of taint meta-data and propagate these correctly. PHP Aspis then uses the taint meta-data to automatically sanitise user-provided untrusted values and transparently prevent injection attacks. Overall, PHP Aspis does not require modifications to the PHP language runtime or to the web server.

Our evaluation shows that, by using partial taint tracking, PHP Aspis successfully prevents most XSS and SQLI exploits reported in public Wordpress plugins since 2010. Page generation time is significantly reduced compared to tracking taint in the entire Wordpress codebase.

In summary, the contributions of this paper are:

- a taint tracking implementation for PHP that uses source code transformations only;
- a method for applying taint tracking only to parts of a web application, in which exploits are more likely to occur;
- an implementation of a code transformation tool and its evaluation with real-world exploits reported in the Wordpress platform.

The next section provides background on code injection vulnerabilities and introduces partial taint tracking as a suitable defence. In §3, we describe our approach for achieving taint propagation in PHP based on code rewriting, and we show how to limit the scope of taint tracking to parts of a codebase. Finally, we evaluate our approach by securing Wordpress in §4 and conclude in §5.

2 Preventing Injection Vulnerabilities

2.1 Injection Vulnerabilities

Consider a weblog with a search field. Typically, input to the search field results in a web request with the search term as a parameter:

```
http://goodsite.com/find?t=spaceship
```

A response of the web server to this request may contain the following fragment:

```
<p> The term ``spaceship'' was not found. </p>
```

The important element of the above response is that the user-submitted search term is included as is in the output. This can be easily exploited by an attacker to construct an XSS attack. The attacker first creates the following URL:

```
http://goodsite.com/find?t=<script\%20
src='http://attack.com/attack.js'/>
```

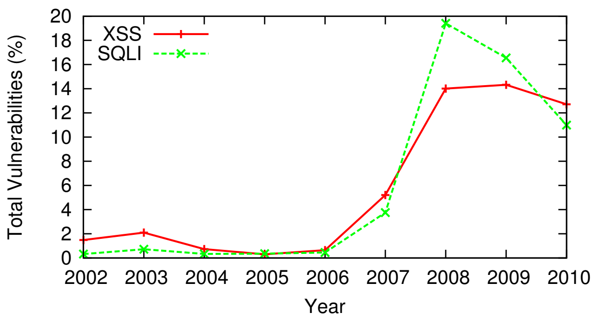


Figure 1: Historical percentage of XSS and SQLI in all CVE vulnerabilities

When an unsuspecting user clicks on this link, the following HTML fragment is generated:

```
<p> The term ``<script src='http://attack.com/
    attack.js' />'' was not found. </p>
```

The victim's browser then fetches the malicious Javascript code and executes it. Since the HTML document originated from `goodsite.com`, the script is executed with full access to the victim's web browser session. If the user has an account with this weblog and is already logged on, their cookies can be sent to the attacker. In general, the script can issue any unverified operations on behalf of the user.

SQL Injection attacks are analogous: they take advantage of applications that process user input to form an SQL query to a database. Similarly, Eval and Shell Injection attacks target PHP's `eval()` and `exec()` statements respectively. Since these statements execute arbitrary code at runtime, such an attack can easily compromise the host machine.

Figure 1 shows the percentage of reported SQLI and XSS vulnerabilities in recent years, as classified in the CVE database. Both problems continue to affect applications despite an increase of developer awareness. Table 1 shows our classification of vulnerabilities for two popular open source web platforms, Wordpress and Drupal. Code injection vulnerabilities are the most common type in both, with higher percentages compared to traditional applications across all programming languages in Figure 1. By relying on prepared statements, an automated way to avoid SQLI (see §2.2), the Drupal platform has comparatively few such exploits.

2.2 Existing Practices

The traditional approach to address injection vulnerabilities is to treat all information that may potentially come from the user as untrusted—it should not be used for sensitive operations, such as page generation or SQL queries, unless properly sanitised or filtered. In the previous example, a proper *sanitisation function* would translate all

	Num. of Occurrences	
Type	*Wordpress*	*Drupal*
Cross Site Scripting	9	28
SQL Injection	3	1
Information Leakage	1	0
Insufficient Access Control	1	9
Eval Injection	0	1
Cross Site Request Forgery	1	0

Table 1: Web application vulnerabilities in 2010

characters with special meaning in HTML to their display equivalents (e.g. "<" to "`&lt;`"). Sanitisation functions such as `htmlentities()` or `escapeshellarg()` are part of the PHP language. After sanitisation, the string can safely be echoed to the client because it can no longer change the semantics of the output. SQLI filtering functions operate similarly but they also check for user-provided SQL keywords in the query [13, 19].

Unfortunately, sanitisation functions are difficult to apply in practice. Each sensitive operation requires a different sanitisation function. For example, if the same string is echoed to the user and used as part of an SQL query, two different strings must be generated based on the original value. Centralised filtering of input data when they are received is impractical because there is no single data representation that is both meaningful and secure in all possible contexts. For example, the string "`WHERE`" is safe in HTML but not in an SQL query. Therefore, developers have to propagate the original user data and only sanitise them before being used.

In addition, sanitisation assumes that developers can effectively track the origin of data and enforce that user-generated data always pass through their respective sanitisation function. In practice, left-out checks by inexperienced developers or unforeseen interactions that result in unexpected data flow (e.g. assuming that a script cannot be called from an external user) are likely to occur.

2.2.1 Static Approaches

Past research has suggested static analysis tools that detect injection vulnerabilities in PHP scripts. Pixy [11] and WebSSARI [10] rely on data flow analysis to detect sensitive functions that may receive user data without sanitisation and produce warnings. Wassermann and Su [16] model string values and operations as grammars and then inspect them before query operations to reduce the false positive rate for SQLI detection. Xie and Aiken [17] use symbolic execution to support PHP's dynamic features and report a low false positive rate.

Although static analysis tools do not introduce runtime overhead, they are not fully automated, cannot support all PHP features and do not always achieve a low false positive rate. In addition, they cannot handle vulnerabilities that involve the file system or the database. As a result, such tools are not widely used for PHP development.

Prepared statements are a way to avoid SQLI exploits. Instead of concatenating queries, an application defines static placeholder queries with parameters filled in at runtime. Parameters passed to the placeholders cannot change the semantics of the query, as its structure is determined in advance when the statement is prepared. In practice, many PHP applications do not use them because they were not traditionally supported by PHP or MySQL and instead manually sanitise SQL queries.

2.2.2 Dynamic Taint Tracking

A dynamic approach to addressing injection vulnerabilities in existing applications when they occur is *runtime taint tracking* [19, 6]. It automates the tracking of the origin of data and enforces that data pass through their respective sanitisation functions. Runtime taint tracking involves three different steps:

1. **Data entry points.** All data entering the application that may originate from the user are transparently augmented with *taint meta-data.* The form of these meta-data may vary: from one bit that marks that a particular string is user-provided (or *tainted*) [14] to a pointer that links to arbitrary policy objects [19].
2. **Taint propagation.** As the application processes data, the runtime system transparently propagates the associated taint meta-data. For example, when a tainted string is concatenated with another string, the result must be marked as tainted.
3. **Guarded sinks.** Every operation that can be used in an injection vulnerability (e.g. `echo()` and `eval()`) is intercepted. The interceptor examines the corresponding taint meta-data and calls the relevant sanitisation function or aborts the operation.

Taint tracking has been shown to be effective in securing existing web applications [19, 18, 9, 14, 6]. Compared to static approaches, it does not require either debugging or refactoring an existing codebase. Perl and Ruby support it in some form (through Perl's taint mode and Ruby's safe levels) but not PHP. Taint tracking can be applied to PHP by modifying the core of the PHP runtime [19]. Typically, it has been implemented by augmenting the interpreter's `zval` struct with taint data. Simple approaches [13, 14] assign one bit of taint meta-data per string character and propagate that meta-data to sinks independently of the application's sanitisation efforts.

Later systems stored more meta-data per character in order to provide more fine-grained guarantees. Nemesis [7] uses two taint bits to automatically infer authentication and enforce access control. Resin [19] uses a pointer to arbitrary policy objects that can be also used to prevent injection vulnerabilities.

Xu et al. [18] suggest that a taint tracking implementation in C can be used to compile the PHP runtime and transparently add taint tracking support. Their approach ignores sanitisation efforts of the hosted PHP application and therefore suffers from false positives. Also, it does not support different policies for different applications running in the same runtime, a common scenario for many PHP deployments.

However, unless taint tracking is considered part of PHP and is officially adopted, third party implementations are impractical. As the PHP manual puts it:

> *"modifications to the Zend[2] engine should be avoided. Changes here result in incompatibilities with the rest of the world, and hardly anyone will ever adapt to specially patched Zend engines. Modifications can't be detached from the main PHP sources and are overridden with the next update using the "official" source repositories. Therefore, this method is generally considered bad practice"*

In the past, taint tracking support has been suggested as a feature to the PHP community but it has not been adopted, partly because of fears that it may lead to a false sense of security [15].

2.3 Partial Taint Tracking

PHP is the most popular web development language, as indicated by web surveys [2], and its gentle learning curve often attracts less experienced developers. Inexperienced developers are more likely to extend web applications through third-party code in the form of *plugins*.

Such extensibility is frequently a popular feature for web applications but leads to a significant security threat from plugins. In 2009, the CVE database reported that the Wordpress platform suffered from 15 injection vulnerabilities, out of which 13 were introduced by third-party plugins and only 2 involved the core platform. In 2010, the breakdown was similar: 10 vulnerabilities were due to plugins and only 2 due to Wordpress itself.

As a result, not all application code is equally prone to injection vulnerabilities. For example, Wordpress spends much of its page generation time in initialisation code, setting up the platform before handling user requests. This involves time-consuming steps such as querying the database for installed plugins, setting them up, and generating static parts of the response involving theme-aware headers and footers.

Injection vulnerabilities, on the other hand, tend to appear in code that handles user-generated content: CVE-2010-4257, an SQLI vulnerability, involved a function that handles track-backs after a user published a post; CVE-2009-3891, an XSS vulnerability, involved a function that validates uploaded files; and CVE-2009-2851 and CVE-2010-4536, again XSS vulnerabilities, involved multiple functions that display user comments.

[2]Zend is the name of the official PHP scripting engine

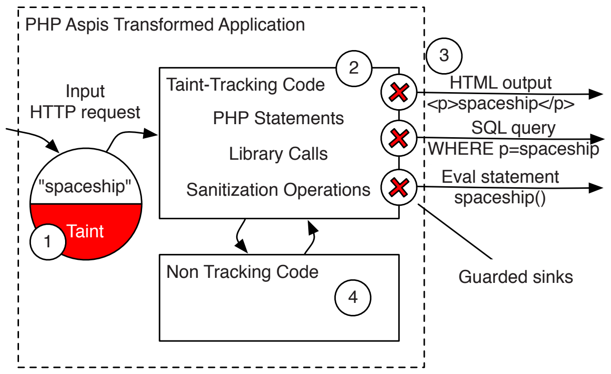


Figure 2: Partial taint tracking using PHP Aspis

We exploit this observation by introducing *partial taint tracking*, which only transforms the source code of the most vulnerable parts of an application in order to support taint tracking. By relying on source-level transformations, partial taint tracking does not require the development and maintenance of a modified version of the PHP runtime.

We use a simple approach to decide when to track taint: we focus on parts of third-party plugins that handle user-generated data. This restricts source code transformations to a small fraction of the codebase of a web application. As a consequence, we mitigate the large performance penalty that exhaustive taint tracking at the source level would incur.

3 PHP Aspis

We describe the design and implementation of *PHP Aspis*, a PHP source code transformation tool for partial taint tracking. Figure 2 presents an overview of how PHP Aspis transforms applications. First, it modifies code that receives data from users and marks the data as user-generated (label 1). Second, it divides the application's codebase in two different categories: *tracking* and *non-tracking* code. Instead of tracking taint uniformly, it focuses on parts of the codebase that are more likely to contain code injection vulnerabilities (label 2). In tracking code, PHP Aspis records the origin of data at the character level and filters data at output statements when an injection vulnerability could exist (label 3). For non-tracking code, it does not perform taint tracking, trusting the code not to be vulnerable (label 4).

Next we introduce the representation of taint meta-data used to record the origin of the data in each variable. In §3.2, we describe the transformations that can be ap-

Sanitisation functions	`htmlentities()`
	`htmlspecialchars()`

Guarded sinks	`echo()` ⇒ `AspisAntiXss()`
	`print()` ⇒ `AspisAntiXss()`
	`...`

Table 2: Excerpt of the definition of the XSS taint category

plied to PHP source code to (a) ensure its correct operation in the presence of taint meta-data; (b) propagate taint meta-data correctly; and (c) attach checks that inspect taint meta-data before each "sensitive" operation. Finally, we discuss how the non-tracking code can interact with the parts of the application that have been transformed to track taint in §3.3

3.1 Taint Representation

PHP Aspis uses *character-level* taint tracking, i.e. tracks the taint of each string character individually [19]. Traditional variable-level taint tracking implementations (e.g. Ruby's safe levels) require the developer to untaint values explicitly before they are used. Instead, PHP Aspis prevents injection attacks transparently, and for this, it needs to know the exact characters that originate from the user. For example, consider an application that concatenates a user-provided value with a static HTML template, stores the result in `$v` and then returns `$v` to the client as a response. Inferring that variable `$v` is tainted is of little use because `$v` also contains application-generated HTML. Instead, PHP Aspis uses character-level taint meta-data and only sanitises the user-generated parts of `$v`.

3.1.1 Taint Categories

PHP Aspis can track multiple independent and user provided *taint categories*. A taint category is a generic way of defining how an application is supposed to sanitise data and how PHP Aspis should enforce that the application always sanitises data before they are used.

Each taint category is defined as a set of *sanitisation functions* and a set of *guarded sinks*. Sanitisation functions can be PHP library functions or can be defined by the application. A sanitisation function is called by the application to transform untrusted user data so that they cannot be used for a particular type of injection attack. Commonly, sanitisation functions either transform unsafe character sequences to safe equivalents (e.g. `htmlentities`) or filter out a subset of potentially dangerous occurrences (e.g. by removing `<script>` but not `<b>`). Calls to sanitisation functions by the application are intercepted and PHP Aspis untaints the corresponding data to avoid sanitising them again.

Guarded sinks are functions that protect data flow to sensitive sink functions. When a call to a sink function is made, PHP Aspis invokes the guard with references to the parameters passed to the sink function. The guard is a user-provided function that has access to the relevant taint category meta-data and typically invokes one or more sanitisation functions for that taint category.

For example, Table 2 shows an excerpt of an XSS taint category definition. It specifies that a user-provided string can be safely echoed to the user after either `htmlentities` or `htmlspecialchars` has been invoked on it. The second part exhaustively lists all functions that can output strings to the user (e.g. `echo`, `print`, etc.) and guards them with an external filtering function (`AspisAntiXss`). The guard either aborts the print operation or sanitises any remaining characters. The administrator can change the definitions of taint categories according to the requirements of the application.

By listing all the sanitisation functions of an application in the relevant taint category, PHP Aspis can closely monitor the application's sanitisation efforts. When applied to a well designed application, PHP Aspis untaints user data as they get sanitised by the application, but before they reach the sink guards. Thus, sink guards can apply a simple, application agnostic, sanitisation operation (e.g. `htmlentities`) acting as a "safety net".

On the other hand, an application may not define explicit sanitisation functions or these functions may be omitted from the relevant taint category. In such cases, sink guards have to replicate the filtering logic of the application. In general, however, sink guards lack contextual information and this prevents them from enforcing context-aware filtering, e.g. guards cannot enforce sanitisation that varies according to the current user.

A different taint category must be used for each type of injection vulnerability. PHP Aspis tracks different taint categories independently from each other. For example, when a sanitisation function of an XSS taint category is called on a string, the string is still considered unsanitised for all other taint categories. This ensures that a sanitisation function for handling one type of injection vulnerability is not used to sanitise data for another type.

3.1.2 Storing taint meta-data

It is challenging to represent taint meta-data so that it supports arbitrary taint categories and character-level taint tracking. This is due to the following properties of PHP:

P1 PHP is not object-oriented. Although it supports objects, built-in types such as `string` cannot be augmented transparently with taint meta-data. This precludes solutions that rely on altered class libraries [6].

P2 PHP does not offer direct access to memory. Any solution must track PHP references because variables' memory addresses cannot be used [18].

String	Taint meta-data
`$s='Hello'`	`array(0=>false)`
`$n='John'`	`array(0=>true)`
`$r=$s.$n`	`array(0=>false, 5=>true)`

Table 3: Representation of taint meta-data for a single taint category

P3 PHP uses different assignment semantics for objects ("by reference") compared to other types including arrays ("by copy"). This does not allow for the substitution of any scalar type with an object without manually copying objects to avoid aliasing.

P4 PHP is a dynamically typed language, which means that there is no generic method to statically identify all string variables.

Due to these properties, our solution relies on PHP arrays to store taint meta-data by enclosing the original values. Table 3 shows how PHP Aspis encodes taint meta-data for a single taint category. For each string, PHP Aspis keeps an array of character taints, with each index representing the first character that has this taint. In the example, string `$s` is untainted, `$n` is tainted and their concatenation, `$r`, it untainted from index 0 to 4, and tainted from index 5 onwards. Numerical values use the same structure for taint representation but only store a common taint for all digits.

Taint meta-data must remain associated with the value that they refer to. As shown in Table 4, we choose to store them together. First, all scalars such as `'Hello'` and `12` are replaced with arrays (rows 1 and 2). We refer to this enclosing array as the value's *Aspis*. The Aspis contains the original value and an array of the taint meta-data for all currently tracked taint categories (`TaintCats`). Similarly, scalars within arrays are transformed into Aspis-protected values.

According to P4, PHP lacks static variable type information. Moreover, it offers type identification functions at runtime. When scalars are replaced with arrays, the system must be able to distinguish between an Aspis-protected value and a proper array. For this, we enclose the resulting arrays themselves in an Aspis-protected value, albeit without any taint (`false` in rows 3 and 4). The original value of a variable can always be found at index 0 when Aspis-protected. Objects are handled similarly. Aspis-protected values can replace original values in all places except for array keys: PHP arrays can only use the types `string` or `int` as keys. To circumvent this, the key's taint categories are attached to the content's Aspis (`KeyTaintCats`) and the key retains its original type (row 5).

Overall, this taint representation is compatible with the language properties mentioned above. By avoiding storing taint inside objects, we ensure that an assignment cannot lead to two separate values referencing the same taint category instances (P3). By storing taint categories in place, we ensure that variable aliasing correctly aliases taints (P2). Finally, by not storing taint meta-data separately, the code transformations that enable taint propagation can be limited to correctly handling the original, Aspis-protected values. As a result, the structure of the application in terms of functions and classes remains unchanged, which simplifies interoperability with non-tracking code as explained in § 3.3.

	Original value	Aspis-protected value
1.	**`'Hello'`**	`array(` **`'Hello'`**`,TaintCats )`
2.	**`12`**	`array(`**`12`**`,TaintCats)`
3.	**`array()`**	`array(` **`array()`** `,false)`
4.	**`array('John')`**	`array(` **`array(`** `array(` **`'John'`**`,TaintCats )` **`)`** `,false)`
5.	**`array(13=>20)`**	`array(` **`array(13=>`** `array(` **`20`**`,ContentTaintCats, KeyTaintCats )` **`)`** `,false)`

Table 4: Augmenting values with taint meta-data

3.2 Taint-tracking Transformations

Based on this taint representation, PHP Aspis modifies an application to support taint tracking. We refer to these source code transformations as *taint-tracking transformations*. These transformations achieve the three steps described in §2.2.2 required for runtime taint tracking: data entry points, taint propagation and guarded sinks.

3.2.1 Data Entry Points

Taint-tracking transformations must mark any user-generated data as fully tainted, i.e. all taint meta-data for every taint category in an Aspis-protected value should be set to `true`. Any input channel such as the incoming HTTP request that is not under the direct control of the application may potentially contain user-generated data.

Original expression	Transformed expression
`$s.$t`	`concat($s,$t)`
`$l = &$m`	`$l=&$m`
`$j = $i++`	`$j=postincr($i)`
`$b+$c`	`add($b,$c)`
`if ($v) {}`	`if ($v[0]) {}`
`foreach ($a as $k=>$v) {...}`	`foreach ($a[0] as $k=>$v) {restoreTaint($k,$v)...}`

Table 5: Transformations to propagate taint and restore the original semantics when Aspis-protected values are used

In each transformed PHP script, PHP Aspis inserts initialisation code that (1) scans the superglobal arrays to identify the HTTP request data, (2) replaces all submitted values with their Aspis-enclosed counterparts and (3) marks user submitted values as fully tainted. All constants defined within the script are also Aspis-protected, however, they are marked as fully untainted (i.e. all taint meta-data for every taint category have the value `false`). As a result, all initial values are Aspis-protected in the transformed script, tainted or not.

3.2.2 Taint Propagation

Next all statements and expressions are transformed to (1) operate with Aspis-protected values, (2) propagate their taint correctly and (3) return Aspis-protected values.

Table 5 lists some representative transformations for common operations supported by PHP Aspis. Functions in the right column are introduced to maintain the original semantics and/or propagate taint. For example, `concat` replaces operations for string concatenating in PHP (e.g. double quotes or the concat operator ".") and returns an Aspis-protected result. Control statements are transformed to access the enclosed original values directly. Only the `foreach` statement requires an extra call to `restoreTaint` to restore the taint meta-data of the key for subsequent statements in the loop body. The meta-data is stored with the content in `KeyTaintCats`, as shown in row 5 of Table 4.

PHP function library. Without modification, built-in PHP functions cannot operate on Aspis-protected values and do not propagate taint meta-data. Since these functions are commonly compiled for performance reasons, PHP Aspis uses *interceptor functions* to intercept calls to them and attach wrappers for taint propagation.

By default, PHP Aspis uses a generic interceptor for built-in functions. The generic interceptor reverts Aspis-protected parameters to their original values and wraps return values to be Aspis-protected again. This default behaviour is acceptable for library functions that do not propagate taint (e.g. `fclose`). However, the removal of taint from result values may lead to false negatives in taint tracking. PHP Aspis therefore provides custom interceptor functions for specific built-in functions. By increasing the number of intercepted functions, we improve the accuracy of taint tracking and reduce false negatives.

The purpose of a custom interceptor is to propagate taint from the input parameters to the return values. Such interceptors rely on the well defined semantics of the library functions. When possible, the interceptor calculates the taint of the return value based on the taints of the inputs (e.g. `substr`). It then removes the taint meta-data from the input values, invokes the original library function and attaches the calculated taint to the result value. Alternatively, the interceptor compares the result value to the passed parameter and infers the taint of the result. As an example, assume that the interceptor for `stripslashes` receives a string with a taint category `array(0=>false,5=>true)`. The comparison of the original to the result string identifies the actual character indices that `stripslashes` removed from the original string; assume here, for simplicity, that only index 2 was removed. To calculate the taint of the result, the interceptor subtracts from all taint category indices the total number of characters removed before each index. Thus, it returns `array(0=>false,4=>true)` in the given example. In total, 66 provided interceptors use this method.

For other functions, the interceptor can use the original function to obtain a result with the correct taint automatically. For example, `usort` sorts an array according to a user-provided callback function and thus can sort Aspis-protected values without changes. If the callback is a library function, the function is unable to compare Aspis-protected elements and calls to `usort` would fail. When callbacks are used, custom interceptors introduce a new callback replacing the old. That new callback calls the original callback after removing taint from its parameters. In total, 21 provided interceptors used this method.

In cases in which the result taint cannot be determined, such as for `sort`, an interceptor provides a separate, taint-aware version of the original PHP library function. We had to re-implement 19 library functions in this way.

Our current prototype intercepts 106 functions. These include most of the standard PHP string library that are enough to effectively propagate taint in Wordpress (§4).

Dynamic features. PHP has many dynamic features such as variable variables, variable function calls and the `eval` and `create_function` functions. These are not compatible with Aspis-protected values, but PHP Aspis must nevertheless maintain their correct semantics.

Variable variables only require access to the enclosed string. A dynamic access to a variable named `$v` is transformed from `$$v` to `${$v[0]}`. *Variable func-*

tion calls that use variables or library functions (e.g. `call_user_func_array`) allow a script to call a function that is statically unknown. PHP Aspis transforms these calls and inspects them at runtime. When a library call is detected, PHP Aspis generates an interceptor, as described in the previous section, at runtime.

The functions `eval` and `create_function` are used to execute code generated at runtime. Since application generated code does not propagate taint, PHP Aspis must apply the taint-tracking transformations at runtime. To avoid a high runtime overhead, PHP Aspis uses a caching mechanism such as Zend Cache when available.

3.2.3 Guarded Sinks

PHP Aspis can protect code from injection vulnerabilities using the taint meta-data and defined taint categories. As described in §3.1.1, guard functions specified as part of active taint categories are executed before the calls to their respective sensitive sink functions. Guards use PHP's sanitisation routines or define their own. For example, we use an SQL filtering routine that rejects queries with user-provided SQL operators or keywords [13].

3.3 Partial Taint Tracking

The taint tracking transformations used by PHP Aspis require extensive changes to the source code, which has an adverse impact on execution performance. To preserve program semantics, transformations often involve the replacement of efficient low-level operations by slower, high-level ones (see Table 5).

Partial taint tracking aims to improve execution performance by limiting taint tracking to the parts of the application in which injection vulnerabilities are more likely to exist. Partial taint tracking can be applied at the granularity of *contexts*: functions, classes or the global scope. The administrator can assign each of these to be of the following *types*: *tracking* or *non-tracking*.

Next we discuss how the presence of non-tracking code reduces the ability of PHP Aspis to prevent exploits. We also present the additional transformations that are done by PHP Aspis to support partial taint tracking.

3.3.1 Missed Vulnerabilities

When partial taint tracking is used, all code must be classified into tracking or non-tracking code. This decision is based on the trust that the application administrator has in the developers of a given part of the codebase. When parts of the codebase are classified as non-tracking, injection vulnerabilities within this code cannot be detected. On the other hand, PHP Aspis must still be able to detect vulnerabilities in tracking code. However, in the presence of non-tracking code, tracking code may not be the place where an exploit manifests itself and thus can be detected.

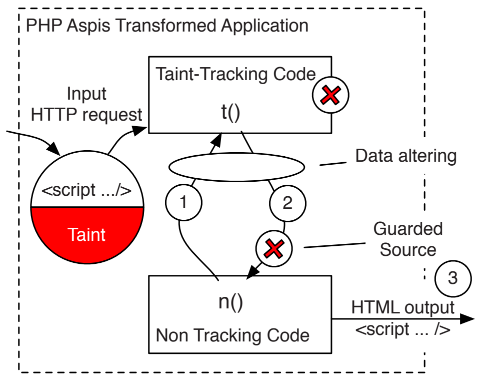


Figure 3: XSS vulnerability manifesting in non-tracking code

For example, a non-tracking function `n` in Figure 3 calls a tracking function `t` (step 1). It then receives a user-provided value `$v` from function `t` (step 2) and prints this value (step 3). If `t` fails to escape user input, `n` is unable to sanitise the data transparently. This is because, in non-tracking code, taint meta-data are not available and calls to sensitive sinks such as `print` are not intercepted. From the perspective of `n`, `t` acts as the source of user generated data that must be sanitised before they leave the tracking context.

To address this issue, PHP Aspis takes a conservative approach. It can sanitise data at the boundary between tracking and non-tracking code. PHP Aspis adds *source guards* to each taint category to provide sanitisation functions for this purpose. A source is a tracking function and the guard is the sanitisation function applied to its return value when called from non-tracking code. In the above example, the tracking function `t` can act as a source of user generated data when called from `n`. A guard for `t` would intercept `t`'s return value and apply `htmlentities` to any user-generated characters. Source guards ensure that user data are properly sanitised before they can be used in non-tracking code.

Note though that this early sanitisation is an additional operation introduced by PHP Aspis. Thus, if the non-tracking context that received the data attempts to sanitise them again, the application would fail. Moreover, there is no generic sanitisation routine that can always be applied because the final use of the data is unknown. Instead, this solution is only suitable for cases when both the final use of the data is known and the application does not perform any additional sanitisation. This is often the case for third-party plugin APIs.

3.3.2 Compatibility Transformations

The taint-tracking transformations in §3.2 generate code that handles Aspis-protected values. For example, a tracking function that changes the case of a string parameter `$p` expects to find the actual string in `$p[0]`. Such a function can no longer be called directly from non-tracking code with a simple string for its parameter. Instead, PHP Aspis requires additional transformations to intercept this call and automatically convert `$p` to an Aspis-protected value, which is marked as fully untainted. We refer to these additional transformations for partial taint tracking as *compatibility transformations*.

Compatibility transformations make changes to both tracking and non-tracking code. These changes alter the data that are exchanged between a tracking context and a non-tracking context, i.e. data exchanged between functions, classes and code in the global scope. They strip Aspis-protected values when passed to non-tracking contexts and restore Aspis protection for tracking contexts.

Function calls. A function call is the most common way of passing data across contexts. PHP Aspis transforms all cross-context function calls: a call from a tracking to a non-tracking context has its taint removed from parameters and the return value Aspis-protected again. The opposite happens for calls from non-tracking to tracking contexts. This also applies to method calls.

Adapting parameters and return values is similar to using the default interceptor function from §3.2. User code, however, can share objects of user-defined classes. Instead of adapting every internal object property, PHP Aspis uses proxy objects that decorate passed values. Consider an object `$o` of class `c` and assume that `c` is a tracking context. When `$o` is passed to the non-tracking context of function `f`, `f` is unable to access `$o`'s state directly or call its methods. Instead, it receives the decorator `$do` that points to `$o` internally. `$do` is then responsible for adapting the parameters and the return values of method calls when such calls occur. It also handles reads and writes of public object properties.

PHP also supports call-by-reference semantics for function parameters. Since changes to reference parameters by the callee are visible to the caller, these parameters effectively resemble return values. Compatibility transformations handle reference parameters similarly to return values—they are adapted to the calling context after the function call returns.

This behaviour can lead to problems if references to a single variable are stored in contexts of different types, i.e. if a tracking class internally has a reference to a variable also stored in a non-tracking class. In such cases, PHP Aspis can no longer track these variables effectively across contexts, forcing the administrator to mark both contexts as tracking or non-tracking. Since shared references to internal state make it hard to maintain class invariants, they are considered bad practice [5] and a manual audit did not reveal any occurrences in Wordpress.

Accessing global variables. PHP functions can access references to variables in the global scope using the `global` keyword. These variables may be Aspis-protected or not, dependent on the type of the current global context and previous function calls. The compatibility transformations rewrite `global` statements: when the imported variable does not match the context of the function, the variable is altered so that it can be used by the function. After the function returns, all imported global variables must be reverted to their previous forms—`return` statements are preceded with the necessary reverse transformations. When functions do not return values, reverse transformations are added as the last function statement.

Accessing superglobal variables. PHP also supports the notion of *superglobals*: arrays that include the HTTP request data and can be accessed from any scope without a `global` declaration. Data in these arrays are always kept tainted; removing their taint would effectively stop taint tracking everywhere in the application. As a result, only tracking contexts should directly access superglobals. In addition, compatibility transformations enable limited access from non-tracking contexts when access can be statically detected (i.e. a direct read to `$_GET` but not an indirect access through an aliasing variable). This is because PHP Aspis does not perform static alias analysis to detect such indirect accesses [11].

Include statements. PHP's global scope includes code outside of function and class definitions and spans across all included scripts. Compatibility transformations can handle different context types for different scripts. This introduces a problem for variables in the global scope: they are Aspis-protected when they are created by a tracking context but have their original value when they are created by a non-tracking context.

To address this issue, PHP Aspis alters temporarily all variables in the global scope to be compatible with the current context of an included script, before an `include` statement is executed. After the `include`, all global variables are altered again to match the previous context type. To mitigate the performance overhead of this, global scope code placed in different files but used to handle the same request should be in the same context type.

Dynamic features. Compatibility transformations intercept calls to `create_function` and `eval` at runtime. PHP Aspis then rewrites the provided code according to the context type of the caller: when non-tracking code calls `eval`, only the compatibility transformations are applied and non-tracking code is generated. Moreover, `create_function` uses a global array to store the context type of the resulting function. This information is then used to adapt the function's parameters and return value in subsequent calls.

3.4 Discussion

The taint tracking carried out by PHP Aspis is not precise. PHP Aspis follows a design philosophy that avoids uncertain taint prediction (e.g. in library functions without interceptors), which may result in false positives and affect application semantics. Instead, it favours a reduced ability to capture taint on certain execution paths, leading to false negatives. For this, it should not be trusted as the sole mechanism for protection against injection attacks.

Partial taint tracking is suited for applications where a partition between trusted and untrusted components is justified, e.g. third-party code. In addition, interactions across such components must be limited because if data flow from a tracking to non-tracking context and back, taint meta-data may be lost. PHP Aspis also does not track taint in file systems or databases, although techniques for this have been proposed in the past [8, 19].

PHP is a language without formal semantics. Available documentation is imprecise regarding certain features (e.g. increment operators and their side effects) and there are behavioural changes between interpreter versions (e.g. runtime call-by-reference semantics). Although our approach requires changes when the language semantics change, we believe that this cost is smaller than the maintenance of third-party runtime implementations that require updates even with maintenance releases.

Our taint tracking transformations support most common PHP features, as they are specified in the online manual [1]. We have yet to add support for newer features from PHP5 such as namespaces or closures.

4 Evaluation

The goals of our evaluation are to measure the effectiveness of our approach in preventing real-world vulnerabilities and to explore the performance penalty for the transformed application. To achieve this, we use PHP Aspis to secure an installation of *Wordpress* [3], a popular open source web logging platform, with known vulnerabilities.

We first describe how an administrator sets up PHP Aspis to protect a Wordpress installation. We then discuss the vulnerabilities observed and show how PHP Aspis addresses them. Finally, we measure the performance penalty incurred by PHP Aspis for multiple applications.

4.1 Securing Wordpress

Wordpress' extensibility relies on a set of hooks defined at certain places during request handling. User-provided functions can attach to these hooks and multiple types are supported: *actions* are used by plugins to carry out operations in response to certain event (e.g. send an email when a new post is published), and *filters* allow a plugin to alter data before they are used by Wordpress (e.g. a post must receive special formatting before being displayed).

CVE	Type	Guarded Sources	Prevented
2010-4518	XSS	1	Yes
2010-2924	SQLI	2	Yes
2010-4630	XSS	0	Yes
2010-4747	XSS	1	Yes
2011-0740	XSS	1	Yes
2010-4637	XSS	2	Yes
2010-3977	XSS	5	Yes
2010-1186	XSS	15	Yes
2010-4402	XSS	6	Yes
2011-0641	XSS	2	Yes
2011-1047	SQLI	1	Yes
2010-4277	XSS	3	Yes
2011-0760	XSS	1	No
2011-0759	XSS	9	No
2010-0673	SQLI	–	–

Table 6: Wordpress plugins' injection vulnerabilities; reported in 2010 and in the first quarter of 2011.

A plugin contains a set of event handlers for specific actions and filters and their initialisation code. Plugin scripts can also be executed through direct HTTP requests. In such cases, plugin scripts execute outside of the main Wordpress page generation process.

We secure a plugin from injection vulnerabilities using PHP Aspis as follows: first, we list the functions, classes and scripts defined by the plugin and mark the relevant contexts as *tracking*; second, we automatically inspect the plugin for possible sensitive sinks, such as print statements and SQL queries. We then decide the taint categories to be used in order to avoid irrelevant tracking (i.e. avoid tracking taint for eval injection if no `eval` statements exist); third, we obtain a list of event handlers from the `add_filter` statements used by the plugin. We augment the taint category definitions with these handlers as guarded sources because filters' return values are subsequently used by Wordpress (§3.3); and fourth, we classify the plugin initialisation code as non-tracking as it is less likely to contain injection vulnerabilities (§2.3).

4.2 Security

Table 6 lists all injection vulnerabilities reported in Wordpress plugins since 2010. For each vulnerable plugin, we verify the vulnerability using the attack vector described in the CVE report. We then try the same attack vector on an installation protected by PHP Aspis.

The experiments are done on the latest vulnerable plugin versions, as mentioned on each CVE report, running on Wordpress 2.9.2. PHP Aspis manages to prevent most vulnerabilities, which can be summarised according to three different categories:

Direct request vulnerabilities. The most common type of vulnerability involves direct requests to plugin scripts. Many scripts do not perform any sanitisation for some parameters (2010-4518, 2010-2924, 2010-4630, 2010-4747, 2011-0740). Others do not anticipate invalid parameter values and neglect to sanitise when printing error messages (2010-4637, 2010-3977, 2010-1186).

PHP Aspis manages to prevent all attack vectors described in the CVE reports by propagating taint correctly from the HTTP parameters, within the taint-transformed plugin scripts and to various printing or script termination statements such as such as `die` and `exit`, when its sanitisation functions are invoked.

Action vulnerabilities. Some of the plugins tested (2010-4402, 2011-0641, 2011-1047) introduce a vulnerability in an action event handler. Similarly, a few other plugins (2010-2924, 2010-1186, 2011-1047) only exhibit a vulnerability through a direct request but explicitly load Wordpress before servicing such a request. Wordpress transforms `$_GET` and `$_POST` by applying some preliminary functions to their values in `wp-settings.php`. As the Wordpress initialisation code is classified as non-tracking, it effectively removes all taint from the HTTP parameters and introduces a false negative for all plugins that execute after Wordpress has loaded.

Given that this behaviour is common for all plugins, we also use taint tracking in a limited set of Wordpress contexts—the functions `add_magic_quotes`, `esc_sql` and the `wpdb` class, which are invoked by this code. As the assignment statements that alter the superglobal tables are in the global scope of `wp-settings.php`, we also perform taint tracking in this context.

Unfortunately, this file is central in Wordpress initialisation: enabling taint tracking there leads to substantially reduced performance. To avoid this problem, we introduce a small function that encloses the assignment statements and we mark its context as tracking. This change required three extra lines of code to define and call the function but it significantly improved performance.

Filter vulnerabilities. From all tested plugins, only one (2010-4277) introduces a vulnerability in the code attached to a filter hook. Although we can verify the behaviour described in the CVE report, we believe that it is intended functionality of Wordpress: the Javascript injection is only done by a user who can normally post Javascript-enabled text. PHP Aspis correctly marks the post's text as untainted and avoided a false positive.

To test the filter, we edit the plugin to receive the text of posts from a tainted `$_GET` parameter instead of the Wordpress hook. After this change, PHP Aspis properly propagates taint and correctly escapes the dangerous Javascript in the guard applied to the filter hook.

4.2.1 False positives and negatives.

As discussed in §3.4, PHP Aspis may introduce both false negatives and false positives. By propagating taint correctly, we largely avoid the problem of false positives. False negatives, however, can be common because they are introduced (1) by built-in library functions that do not propagate taint, (2) by calls to non-tracking contexts and (3), by data paths that involve the file system or the database. In these cases, taint is removed from data, and when that data are subsequently used, vulnerabilities may not be prevented. PHP Aspis' current inability to track taint in the database is the reason why the XSS vulnerabilities 2011-0760 and 2011-0759 are not prevented.

To reduce the rate of false negatives, we use interceptors that perform precise taint tracking for all built-in library functions used by the tested plugins. In addition, we find that classifying the aforementioned set of Wordpress initialisation routines as tracking contexts is sufficient to prevent all other reported injection vulnerabilities. Note that the last vulnerable plugin (2010-0673) has been withdrawn and was not available for testing.

4.3 Performance

To evaluate the performance impact of PHP Aspis, we measure the page generation time for:

- a simple prime generator that tests each candidate number by dividing it with all smaller integers (Prime);
- a typical script that queries the local database and returns an HTML response (DB).
- Wordpress (WP) with the vulnerable Embedded Video plugin (2010-4277). Wordpress is configured to display a single post with a video link, which triggers the plugin on page generation.

Our measurements are taken in a 3 Ghz Intel Core 2 Duo E6850 machine with 4 GiB RAM, running Ubuntu 10.04 32-bit. We use PHP 5.3.3 and Zend Server 5.0.3 CE with Zend Optimizer and Zend Data Cache enabled. For each application, we enable tracking of two taint categories, XSS and SQLI.

Table 7 shows the 90$^{\text{th}}$ percentile of page generation times over 500 requests for various configurations. Overall, we observe that fully tracking taint has a performance impact that increases page generation between 3.4× and 10.4×. The overhead of PHP Aspis depends on how CPU intensive the application is: DB is the least affected because its page generation is the result of a single database query. On the other hand, Prime has the worst performance penalty of 10.4×, mostly due to the replacement of efficient mathematical operators with function calls.

Wordpress (WP) with full taint tracking results in a 6.0× increase of page generation time. With par-

App.	Tracking	Page generation	Penalty
Prime	Off	44.9 ms	-
Prime	On	466.8 ms	10.4×
DB	Off	0.4 ms	-
DB	On	1.3 ms	3.4×
WP	Off	65.6 ms	-
WP	On	394.4 ms	6.0×
WP	Partial	144.3 ms	2.2×

Table 7: Performance overhead of PHP Aspis in terms of page generation time

tial taint tracking configured only on the installed plugin, page generation overhead is significantly reduced to 2.2×. Given that Wordpress uses globals extensively, the main source of performance reduction for the partial taint tracking configuration are the checks on global variable access as part of the compatibility transformations.

Although full taint tracking at the source code level incurs a significant performance penalty, partial taint tracking can reduce the overhead considerably. In practice, 2.2× performance overhead when navigating Wordpress pages with partial taint tracking is acceptable for deployments in which security has priority over performance.

5 Conclusions

In this paper, we presented PHP Aspis, a tool that applies partial taint tracking at the source code level, with a focus on third-party extensions. PHP Aspis avoids the need for taint tracking support in the PHP runtime. Although the performance penalty of PHP Aspis can increase the page generation time by several times, we have shown that if taint tracking is limited only to a subset of a web application, the performance penalty is significantly reduced while many real world vulnerabilities are mitigated. Our evaluation with the Wordpress platform shows that PHP Aspis can offer increased protection when a moderate increase in page generation time is acceptable.

Acknowledgements. We thank David Eyers, Evangelia Kalyvianaki, the anonymous reviewers and our shepherd, Nickolai Zeldovich, for their comments on earlier paper drafts. This work was supported by the grant EP/F04246 ("SmartFlow: Extendable Event-Based Middleware") from the UK Engineering and Physical Sciences Research Council (EPSRC).

References

[1] PHP online manual. `www.php.net/manual`.

[2] Programming Languages Popularity website. `www.langpop.com`.

[3] Wordpress website. `www.wordpress.org`.

[4] PHP Security Guide 1.0. Tech. rep., PHP Security Consortium, 2005.

[5] BLOCH, J. *Effective Java*. Prentice Hall, 2008.

[6] CHIN, E., AND WAGNER, D. Efficient character-level taint tracking for Java. In *Secure Web Services* (Chicago, IL, 2009), ACM.

[7] DALTON, M., KOZYRAKIS, C., AND ZELDOVICH, N. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *Security Symposium* (Montreal, Canada, 2009), USENIX.

[8] DAVIS, B., AND CHEN, H. DBTaint: Cross-Application Information Flow Tracking via Databases. In *WebApps* (Boston, MA, 2010), USENIX.

[9] HALFOND, W., ORSO, A., AND MANOLIOS, P. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering 34*, 1 (2008).

[10] HUANG, Y.-W., YU, F., ET AL. Securing web application code by static analysis and runtime protection. In *World Wide Web* (New York, NY, 2004), ACM.

[11] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Symposium on Security and Privacy* (Berkeley, CA, 2006), IEEE.

[12] MITRE CORPORATION. Common Vulnerabilities And Exposures (CVE) database, 2010.

[13] NGUYEN-TUONG, A., GUARNIERI, S., ET AL. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference* (Chiba, Japan, 2005).

[14] PIETRASZEK, T., AND BERGHE, C. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection* (Menlo Park, CA, 2006).

[15] VENEMA, W. Runtime taint support proposal. In *PHP Internals Mailing List* (2006).

[16] WASSERMANN, G., AND SU, Z. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI* (San Diego, CA, 2007), ACM.

[17] XIE, Y., AND AIKEN, A. Static detection of security vulnerabilities in scripting languages. In *Security Symposium* (Vancouver, Canada, 2006), USENIX.

[18] XU, W., BHATKAR, S., AND SEKAR, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Security Symposium* (Vancouver, Canada, 2006), USENIX.

[19] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving application security with data flow assertions. In *SOSP* (New York, NY, 2009), ACM.

Secure Data Preservers for Web Services

Jayanthkumar Kannan — *Google Inc.*
Petros Maniatis — *Intel Labs*
Byung-Gon Chun — *Yahoo! Research*

Abstract

We examine a novel proposal wherein a user who hands off her data to a web service has complete choice over the code and policies that constrain access to her data. Such an approach is possible if the web service does not require raw access to the user's data to implement its functionality; access to a carefully chosen interface to the data suffices.

Our data preserver framework rearchitects such web services around the notion of a *preserver*, an object that encapsulates the user's data with code and policies chosen by the user. Our framework relies on a variety of deployment mechanisms, such as administrative isolation, software-based isolation (e.g., virtual machines), and hardware-based isolation (e.g., trusted platform modules) to enforce that the service interacts with the preserver only via the chosen interface. Our prototype implementation illustrates three such web services, and we evaluate the cost of privacy in our framework by characterizing the performance overhead compared to the status quo.

1 Introduction

Internet users today typically entrust web services with diverse data, ranging in complexity from credit card numbers, email addresses, and authentication credentials, to datasets as complex as stock trading strategies, web queries, and movie ratings. They do so with certain expectations of *how* their data will be used, *what* parts will be shared, and with *whom* they will be shared. These expectations are often violated in practice; the Dataloss database [1] lists 400 data loss incidents in 2009, each of which exposed on average half a million customer records outside the web service hosting those records.

Data exposure incidents can be broadly categorized into two classes: *external* and *internal*. External violations occur when an Internet attacker exploits service vulnerabilities to steal user data. Internal violations occur when a malicious insider at the service abuses the possession of user data beyond what the user signed up for, e.g., by selling customer marketing data. 65% of the aforementioned data-exposure incidents are external, while about 30% are internal (the remainder have no stated cause).

The impact of such data exposure incidents is exacerbated by the fact that data owners are powerless to *proactively* defend against the possibility of abuse. Once a user hands off her data to a web service, the user has *given up control* over her data irreversibly; "the horse has left the barn" forever and the user has no further say on how her data is used and who uses it.

This work introduces a novel proposal that restores control over the user's data to the user; we insist that *any code* that needs to be trusted by the user and *all* policies on how her data is accessed are *specified* by her. She *need not* rely on the web service or any of its proprietary code for performing access control over her data; the fate of her data is completely up to her. The user is free to choose code from any third party (e.g., an open source repository, or a security company) to serve as her software trusted computing base; she need not rely on a proprietary module. This trust can even be based on a proof of correctness (e.g., Proof Carrying Code [20]). Further, *any policies* pertaining to how her data is stored and accessed (e.g., by whom, how many times) are under her control. These properties of personalizable code and policies distinguish us from currently available solutions.

At first look, personalizable trust looks difficult to achieve: the web service may have arbitrary code that requires access to the user's data, but such code cannot be revealed publicly and the user cannot be allowed to choose it. However, for certain classes of web services where the application does not require access to the raw data and a *simple interface* suffices, such personalizable trust is possible. We will show later that for several classes of web services such restricted access is sufficient.

For such services, it is possible to enforce the use of a *well-defined, access-controlled interface* to user data. For example, a movie recommendation service that requires only statistical aggregates of a user's movie watching history need never access the user's detailed watching history. To provide such enforcement, we leverage the idea of preserving the user's own data with code and policy she trusts; we refer to such a preserved object as a *secure data preserver* (see Figure 1).

One could imagine a straightforward implementation of the secure data preserver (SDaP) approach that serves the user's data via the interface from a trusted server (say the user's home machine, or a trusted hosting service). Unfortunately, this solution has two main drawbacks. First, placing user data at a maximally isolated hosting service may increase provisioning cost and access latency, and reduce bandwidth, as compared to the status quo which offers no isolation guarantees. Second, even when data placement is explicitly decided by a particular business

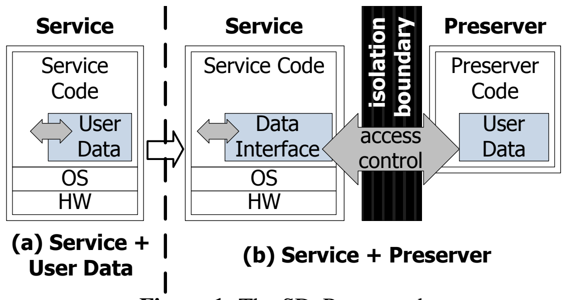


Figure 1: The SDaP approach.

model—for instance, charging services such as Google Checkout that store user data at Google's data centers—the user control available may be lacking: Google Checkout only offers a simple, click-to-pay interface, leaving the user with limited access-control options.

In this work, we explore a general software architecture to enable the use of preservers by users and web services. Our architecture has three salient features. First, it supports *flexible* placement and hosting decisions for preservers to provide various trade-offs between performance, cost, and isolation. Apart from the options of hosting the preserver at a client's machine or at a trusted third party, our framework introduces the *colocation* option; the preserver can be hosted securely at the service itself by leveraging a trusted hardware or software module (e.g., in a separate virtual machine). This colocation option presents a different trade-off choice; higher performance, zero provisioning cost on part of the user, and the involvement of no third parties. Second, we provide a *policy layer* around the available data placement mechanisms with two purposes: to mediate *interface access* so as to enable fine-grained control even for coarse-grained interfaces, and to mediate *data placement* so as to fit the user's desired cost, isolation, and performance goals. Third, we offer preserver *transformation mechanisms* that reduce the risk of data exposure (by filtering raw data) or increase anonymity (by aggregating data with those of other users).

We evaluate SDaPs experimentally in a prototype that implements three applications with diverse interface access patterns: day-trading, targeted advertising, and payment. We evaluate these applications along a spectrum of placement configurations: at the user's own computer or at a trusted third party (TTP), offering maximum isolation from the service but worst-case performance, and at the service operator's site, isolated via virtualization, offering higher performance. The isolation offered by client- or TTP-hosted preservers comes at the cost of Internet-like latencies and bandwidth for preserver access (as high as 1 Gbps per user for day-trading). Our current virtual machine monitor/trusted platform module based colocation implementation protects against external threats; it can be extended to resist internal threats as well by leveraging secure co-processors using standard techniques [32]. Colocation offers significantly better performance than off-site placement, around 1 ms interface latency at a reasonable storage overhead (about 120 KB per user). Given the growing cost of data exposure (estimated to be over 200 dollars per customer record [22]), these overheads may be an acceptable price for controls over data exposure, especially for financial and health information.

While we acknowledge that the concept of encapsulation is itself well-known, our main contributions in this work are: (a) introducing the principle of interface-based access control in the context of web services, and demonstrating the wide applicability of such a model, (b) introducing a new colocation model to enforce such access control, (c) design of a simple yet expressive policy language for policy control, and (d) an implementation that demonstrates three application scenarios.

Our applicability to a service is limited by three factors. First, SDaPs make sense only for applications with simple, narrow interfaces that expose little of the user data. For rich interfaces (when it may be too optimistic to hope for a secure preserver implementation) or interfaces that may expose the user's data directly, information flow control mechanisms may be preferable [12, 24, 27, 33]. Second, we do not aim to *find* appropriate interfaces for applications; we limit ourselves to the easier task of helping a large class of applications that already have such interfaces. Third, we recognize that a service will have to be refactored to accommodate strong interfaces to user data, possibly executing locally preserver code written elsewhere. We believe this is realistic given the increasing support of web services for third-party applications.

The rest of the paper is structured as follows. Section 2 refines our problem statement, while Sections 3 and 4 present an architectural and design overview respectively of our preserver framework. We discuss our implementation in Section 5 and evaluate its performance and security in Section 6. We then present related work in Section 7 and conclude in Section 8.

2 The User-Data Encapsulation Problem

In this section, we present a problem statement, including assumptions, threat models we target, and the deployment scenarios we support. We then present motivating application scenarios followed by our design goals.

2.1 Problem Statement

Our goal is to rearchitect web services so as to isolate user data from service code and data. Our top priority is to protect user data from *abusive access*; we define as abusive any access that violates a well-defined, well-known interface, including unauthorized disclosure or update. We aim for the interface to be such that the data need never be revealed directly and policies can be meaningfully specified on its invocation; we will show such interfaces can be found for a variety of services. A secondary priority is to provide reasonable performance and scalability. Given a chosen interface, our security objective is to ensure that any information revealed to a service is obtainable only via a sequence of legal interface invocations. This significantly limits an adversary's impact since the interface does not expose raw data; at best, she can exercise the interface to the maximum allowed by the policies.

It is important to clarify the scope of our work. First, we assume that an interface has been arrived at for a service that provides the desired privacy to the user; we do not verify that such an interface guarantees the desired privacy, or automate the process of refactoring existing code. Second, we assume that a preserver implementation carries out this interface correctly without any side channels or bugs. We aim for simple narrow interfaces to make correct implementation feasible.

2.2 Threat Model

The threat model we consider is variable, and is chosen by the user. We distinguish three choices, of increasing threat strength. A *benign threat* is one caused by service misconfiguration or buggy implementation (e.g., missing access policy on an externally visible database, insecure cookies, etc.), in the absence of any malice. An *external adversarial threat* corresponds to an Internet attacker who exploits software vulnerabilities at an otherwise honest service. An *internal adversarial threat* corresponds to a malicious insider with physical access to service infrastructure.

We make standard security assumptions. First, trusted hardware such as Trusted Platform Modules (TPM [5]) and Tamper Resistant Secure Coprocessors (e.g., IBM 4758 [11]) provide their stated guarantees (software attestation, internal storage, data sealing) despite software attacks; tamper-resistant secure coprocessors can also withstand physical attacks (e.g., probes). Second, trusted hypervisors or virtual machine monitors (e.g., Terra [13], SecVisor [25]) correctly provide software isolation among virtual machines. Third, a trusted hosting service can be guaranteed to adhere to its stated interface, and is resistant to internal or external attacks. Finally, we assume that standard cryptography works as specified (e.g., adversaries cannot obtain keys from ciphertext).

2.3 Deployment Scenario

We aim to support three main deployment scenarios corresponding to different choices on the performance-isolation trade-off.

The closest choice in performance to the status quo with an increase in isolation keeps user data on the same service infrastructure, but enforces the use of an access interface; we refer to this as *colocation*. Typical software encapsulation is the simplest version of this, and protects from benign threats such as software bugs. Virtualization via a secure hypervisor can effectively enforce that isolation even for external attacks that manage to execute at the privilege level of the service. Adding attestation (as can be provided by TPMs for instance) allows an honest service to prove to clients that this level of isolation is maintained. However, internal attacks can only be tolerated with the help of tamper-resistant secure co-processors. Though our design focuses primarily on the colocation scenario, we support two other deployment options.

The second deployment option is the *trusted third party (TTP)* option; placing user data in a completely separate administrative domain provides the greatest isolation, since even organizational malfeasance on the part of the service cannot violate user-data interfaces. However, administrative separation implies even further reduced interface bandwidth and timeliness (Internet-wide connections are required). This scenario is the most expensive for the user, who may have to settle for existing types of data hosting already provided (e.g., use existing charging services via their existing interfaces), rather than springing for a fully customized solution. The third deployment option offers a similar level of isolation and is the cheapest (for the user); the user can host her data on her own machine (the *client-side* option). Performance and availability are the lowest due to end-user connectivity.

2.4 Usage Idioms

The requirement of a suitable interface is fundamental to our architecture; we now present three basic application *idioms* for which such interface-based access is feasible to delineate the scope of our work.

Sensitive Query Idiom: Applications in this idiom are characterized by a large data stream offered by the service (possibly for a fee), on which the user wishes to evaluate a sensitive query. Query results are either sent back to the user and expected to be limited in volume, or may result in service operations. For example, in Google Health, the user's data is a detailed list of her prescriptions and diseases, and the service notifies her of any information relating to these (e.g., a product recall, conflict of medicines). In applications of this idiom, an interface of the form *ReportNewDatum()* is exported by the user to the service; the service invokes this interface upon arrival of a new datum, and the preserver is responsible for notifying the user of any matches or initiating corresponding actions. Notifications are encrypted and the preserver can batch them up, push them to the user or wait for pulls by the user, and even put in fake matches. Other examples include stock-trading, Google News Alerts etc.

Analytics on Sensitive Data Idiom: This idiom is characterized by expensive computations on large, sensitive user datasets. The provider executes a public algorithm on large datasets of a single user or multiple users, returning results back to the user, or using the results to offer a particular targeted service. Examples include targeted advertising and recommendation services. In targeted advertising, an activity log of a set of users (e.g., web visit logs, search logs, location trajectories) are mined to construct a prediction model. A slim interface for this scenario is a call of the form *SelectAds(ListOfAds)* which the preserver implements via statistical analysis.

Proxying Idiom: This idiom captures functionality whose purpose is to obtain the help of an external service. The user data in this case has the flavor of a capability for accessing that external service. For instance, consider the case when a user hands her credit card number (CCN) to a web service. If we restructure this application, an interface of the form *Charge(Amount, MerchantAccount)* would suffice; the preserver is responsible for the actual charging, and returning the confirmation code to the merchant. Other examples include confiding email addresses to websites, granting Facebook access to Gmail username/password to retrieve contacts, etc.

Data Hosting Non-idiom: For the idioms we discussed, the interface to user data is simple and narrow. To better characterize the scope of our work, we describe here a class of applications that do not fit within our work. Such applications rely on the service reading and updating the user's data at the finest granularity. Examples are collaborative document-editing services (e.g., Google Docs) or social networking services (e.g., Facebook itself, as opposed to Facebook Apps, which we described above). Applications in the hosting idiom are better handled by data-centric—instead of interface-centric—control mechanisms such as information flow control (e.g., XBook [27]) or end-to-end encryption (e.g., NOYB [14]).

2.5 Design Goals

Based on the applications we have discussed so far, we present our basic design goals:

Simple Interface: A user's data should be accessed only via a *simple, narrow interface*. The use of such interfaces is the key for allowing the user the flexibility to choose code from any provider that implements an interface, while offering confidence that said code is correct.

Flexible Deployment: *Flexible interposition* of the boundary between the user's data and the web service is necessary so that users and services can choose a suitable deployment option to match isolation requirements, at a given performance and cost budget.

Fine-grained Use Policy: Even given an interface, different uses may imply different permissions, with different restrictions. For example, a user may wish to use the Google Checkout service but impose restrictions on time-to-use and budget. As a result, we aim to allow *fine-grained and flexible user control* over how the interface to a user's data is exercised.

Trust But Mitigate Risk: Enforced interfaces are fine, but even assumptions on trusted mechanisms occasionally fail—in fact, we would not be discussing this topic without this painful truth. As a result, *risk mitigation* on what may be exposed even if a trusted encapsulation of a user's data were to be breached is significant. This is particularly important for multi-service workflows, where typically only a small subset of the raw user data or an anonymized form of raw user data need be provided to a subordinate service. For example, consider an advertising service that has access to all of a user's purchase history; if that service delegates the task of understanding the user's music tastes to a music recommendation service, there is no reason to provide the recommendation service with *all* history, but only music-related history.

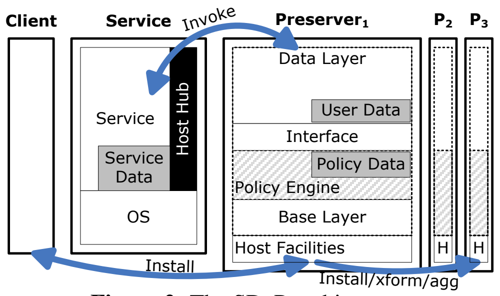


Figure 2: The SDaP architecture.

3 Preserver Architecture

In this section, we present the software components of the architecture, followed by an operational overview that shows their interaction.

3.1 Components

Figure 2 shows a preserver ($Preserver_1$) that serves a user's data to the service via a chosen interface; P_2, P_3 are preservers derived from $Preserver_1$ through hosting transfers or transformations. The three main components of a preserver are shown within $Preserver_1$. The *data layer* is the data-specific code that implements the required interface. The *policy engine* vets interface invocations per the user's policy stored alongside the user's data as *policy data*. Finally, the *base layer* coordinates these two components, and is responsible for: (a) interfacing with the service, (b) implementing the hosting protocol, and (c) invoking any required data transformations alongside hosting transfers. The base layer relies on *host facilities* for its interactions with the external world (such as network access). At the service side, the *host hub* module runs alongside the service. It serves as a proxy between the service code and the preserver, and allows us to decouple the particular application from the details of interactions with the preserver.

3.2 Operational View

The lifecycle of a preserver consists of installation, followed by any combination of invocations, hosting transfers, and transformations. We give an overview of these; our design section discusses them in detail.

A user who wishes to use preservers to protect her data when stored at a service, picks a preserver implementation from third party security companies (e.g., Symantec) that exports an interface suitable for that kind of data. Or, she may pick one from an open-source repository of preserver implementations, or purchase one from an online app-store. Another option is for the service itself to offer a third-party audited preserver which the user may trust. For such a rich ecosystem of preserver implementations, we envision that APIs suitable for specific kinds of data will eventually be well-defined for commonly used sensitive information such as credit card numbers (CCNs), email addresses, web histories, and trading strategies. Services that require use of a specific kind of data can support such an API, which can then be implemented by open-source reference implementations and security companies. The evolution of APIs like OpenSocial [3] are encouraging in this regard. Once the user picks a preserver implementation, she customizes it with policies that limit where her data may be stored and who may invoke it. We envision that users will use visual interfaces for this purpose, which would then be translated to our declarative policy language. Once customization is done, the user initiates an installation process by which the preserver is hosted (at TTP/client/service as desired) and the association between the service and the preserver established.

We discussed needed changes from the user's perspective; we now turn to the service's side. First, a service has to modify its code to interact with the preserver via a programmatic interface, instead of accessing raw data. Since web service code is rewritten much more frequently than desktop applications and is usually modular, we believe this is feasible. For instance, in the case of our charging preserver, the required service-side modifications took us only a day. Second, services need to run third-party preserver code for certain deployment options (colocation, preserver hosted on isolated physical infrastructure within the service). We believe this does not present serious security concerns since preserver functionality is very limited; preservers can be sandboxed with simple policies (e.g., allow network access to only the payment gateway, allow no disk access). A service can also insist that the preserver be signed from a set of well-known security companies. The overhead of preserver storage and invocation may also hinder adoption by services. Our evaluation section (Section 6) measures and discusses the implications of this overhead; this overhead amounts to the *cost of data isolation* in our framework.

Once the association between a service and a preserver is established, the service can make invocation requests via its host hub; these requests are dispatched to the base layer, which invokes the policy engine to determine whether the invocation should be allowed or not. If allowed, the invocation is dispatched to the data layer and the result returned. A service makes a hosting transfer request in a similar fashion; if the policy engine permits the transfer, the base layer initiates a hosting transfer protocol and initiates any policy-specified transformations alongside the transfer.

Interaction between Base Layer and Host Hub: The mechanics of interaction between the host hub (running alongside the service code) and the base layer (at the preserver) depends on the deployment scenario. If the preserver is hosted at a TTP or at the client, then this communication is over the network, and trust is guaranteed by the physical isolation between the preserver and the service. If the preserver is co-located at the service, then communication is achieved via the trusted module at the service site (e.g., Xen hypercalls, TPM late launch, secure co-processor invocation). The base layer requires the following functionality from such a trusted module: (a) isolation, (b) non-volatile storage, which can be used for anti-replay protection, (c) random number generation, and (d) remote attestation (we will argue in Section 6.3 that these four properties suffice for the security of our framework; for now, we note they are provided by all three trust modules we consider).

Details about base layers, policy engines, and aggregation modules follow in the next section; here we focus on the two roles of the host hub. The first is to serve as a proxy for communication to/from the preserver from/to the service. The second role applies only to colocation; it provides host facilities, such as network access, to the preserver. This lets the preserver leverage such functionality without having to implement it, considerably simplifying the implementation. The host hub runs within the untrusted service, but this does not violate our trust, since data can be encrypted if desired before any network communication via the host hub. The host hub currently provides three services: network access, persistent storage, and access to current time from a trusted remote time server (useful for time-bound policies).

4 Design

This section presents the design of two key components of the preserver: the policy engine and the data transformation mechanisms. We discuss the policy engine in two parts: the hosting policy engine (Section 4.1) and the invocation policy engine (Section 4.2). We then discuss data transformations (Section 4.3).

The main challenge in designing the policy layer is to design a policy language that captures typical kinds of constraints (based on our application scenarios), whilst enabling a secure implementation. We make the design decision of using a declarative policy language that precisely defines the set of allowed invocations. Our language is based on a simplified version of SecPAL [7] (a declarative authorization policy language for distributed systems); we reduced SecPAL's features to make simpler its interpreter, since it is part of our TCB.

4.1 Preserver Hosting

We introduce some notation before discussing our hosting protocol. A preserver C storing user U's data and resident on a machine M owned by principal S is denoted as $C_U@[M,S]$; the machine M is included in the notation since whether C can be hosted at M can depend on whether M has a trusted hardware/software module. Once a preserver C (we omit U,S,M when the preserver referred to is clear from the context) has been associated with S, it will respond to invocation requests per any configured policies (denoted by $P(C)$; policies are sent along with the preserver during installation). S can also request hosting transfers from C's current machine to one belonging to another service (for inter-organizational transfers) or to one belonging to S (say, for replication). If this transfer is concordant with the preserver's *hosting policy*, then the *hosting protocol* is initiated; we discuss these next.

Hosting Policy: We wish to design a language flexible enough to allow a user to grant a service S the right to host her data based on: (a) white-lists of services, (b) trusted hardware/software modules available on that service, and (c) delegating the decision to a service. We show an example below to illustrate the flavor of our language.

(1) `alice` SAYS CanHost(`M`) IF OwnsMachine(`amazon`,`M`)
(2) `alice` SAYS CanHost(`M`) IF TrustedService(`S`), OwnsMachine(`S`,`M`), HasCoprocessor(`M`)
(3) `alice` SAYS `amazon` CANSAY TrustedService(`S`)

We use a lower-case typewriter font to indicate principals like Alice (the user; a principal's identity is established by a list of public keys sent by the user or by a certificate binding its name to a public key), capitals to indicate language keywords, and mixed-case for indicating predicates. A predicate can either be built-in or user-defined. The predicate OwnsMachine*(S,M)* is a built-in predicate that is true if S acknowledges M as its own (using a certificate). The predicates HasCoprocessor*(M)*, HasTPM*(M)*, HasVMM*(M)* are built-in predicates that are true if M can prove that it has a co-processor / TPM / VMM respectively using certificates and suitable attestations. The user-defined predicate CanHost*(M)* evaluates to true if machine M can host the user's preserver. The user-defined predicate TrustedService*(S)* is a helper predicate.

The first rule says that `alice` allows any machine M to host her preserver, provided `amazon` certifies such a machine. The second rule indicates `alice` allows machine M to host her preserver if (a) S is a trusted service, (b) S asserts that M is its machine, and (c) M has a secure co-processor. The third rule allows `amazon` to recommend any service S as a trusted service.

A hosting request received at $C_U@[M,S]$ has three parameters: the machine M' to which the transfer is requested, the service S' owning machine M', and a set of assertions P_{HR}. The set of assertions P_{HR} are presented by the principal in support of its request; it may include delegation assertions (such as "`S` SAYS `S'` CanHost*(X)*") and capability assertions (such as "`CA` SAYS HasTPM*(M')*"). When such a request is received by $C_U@[M,S]$, it is checked against its policy. This involves checking whether the fact "`U` SAYS CanHost*(M')*" is derivable from $P(C) \cup P_{HR}$ per SecPAL's inference rules. If so, then the hosting protocol is initiated.

Hosting Protocol: The hosting protocol forwards a preserver from one machine M to another M'. The transfer of a preserver to a TTP is done over SSL; for colocation, the transfer is more complex since the protocol should first verify the presence of the trusted module on the service side. We rely on the attestation functionality of the trusted module in order to do so.

The goal of the hosting protocol is to maintain the confidentiality of the preserver during its transfer to the service side, whilst verifying the presence of the trusted module. We achieve this by extending the standard Diffie-Hellman key-exchange protocol:

- Step 1: $M \rightarrow M'$: (g,p,A), N
- Step 2: $M' \rightarrow M$: B, $Attestation[\ M'$, BaseLayer, $N, (g,p,A), B\ ]$
- Step 3: $M \rightarrow M'$: $\{C_U@[M,S]\}_{DHK}$

Here, $(g,p,A = g^a \ mod\ p)$ and $B = g^b \ mod\ p$ are from the standard Diffie-Hellman protocol, and DHK is the Diffie-Hellman key (generated as $A^b \ mod\ p = B^a \ mod\ p$). This protocol only adds two fields to the standard protocol: the nonce N and the attestation (say, from a Trusted Platform Module) that proves that the base layer generated the value B in response to $((g,p,A),N)$. Thus, the security properties of the original protocol still apply. The attestation is made with an attestation identity key M'; this key is vouched for by a certification authority as belonging to a trusted module (e.g., TPM). The attestation guarantees freshness (since it is bound to N), and rules out tampering on both the input and output.

At the completion of the exchange, the base layer at M' de-serializes the transferred preserver $C'_U@[M',S']$. At this point, the preserver C' is operational; it shares the same data as C and is owned by the same user U. After the transfer, we do not enforce any data consistency between C and C' (much like the case today; if Amazon hands out a user's purchase history to a third party, it need not update it automatically). A user or service can notify any such derived preservers and update them, but this is not automatic since synchronous data updates are typically not required across multiple web services.

4.2 Preserver Invocation Policy

An invocation policy allows the user to specify constraints on the invocation parameters to the preserver interface. Our policy language supports two kinds of constraints: stateless and stateful.

Stateless constraints specify conditions that must be satisfied by arguments to a single invocation of a preserver, e.g., "never charge more than 100 dollars in a single invocation". We support predicates based on comparison operations, along with any conjunction or disjunction operations. Stateful constraints apply across several invocations; for example, "no more than 100 dollars during the lifetime of the preserver"; such constraints are useful for specifying cumulative constraints. For instance, users can specify a CCN budget over a time window. We present an excerpt below.

```
(1) alice SAYS CanInvoke(amazon, A) IF LessThan(A,50)
(2) alice SAYS CanInvoke(doubleclick, A)
IF LessThan(A,Limit), Between(Time, "01/01/10","01/31/10")
STATE (Limit = 50, Update(Limit, A))
(3) alice SAYS amazon CANSAY CanInvoke(S,A)
IF LessThan(A,Limit)
STATE (Limit = 50, Update(Limit, A))
```

The first rule gives the capability "can invoke up to amount A" to Amazon as long as $A < 50$. The second rule shows a stateful example; the semantics of this rule is that DoubleClick is allowed to charge up to a cumulative limit of 50 during Jan 2010. The syntax for a stateful policy is to annotate state variables with the STATE keyword. This policy has a state variable called *Limit* set to 50 initially. The predicate Update*(Limit,A)* is a built-in update predicate that indicates if this rule is matched, then the *Limit* should be updated with the amount A. When a rule is matched with a *state* keyword, it is removed from the policy database, any state variables (e.g., *Limit*) suitably updated, and the new rule inserted into the database. This usage idiom is similar to SecPAL's support for RBAC dynamic sessions. The alternative is to move this state outside the SecPAL policy, and house it within the preserver functionality; we avoid this so that the policy implementation is not split across SecPAL and the preserver implementation. The third rule is very similar to the second rule; however, this rule is matched for any principal to which Amazon has bestowed invocation rights. This means that the limit is enforced across all those invocations; this is exactly the kind of behavior a user would expect.

Transfer of Invocation Policies: We now discuss how the invocation protocol interacts with the hosting protocol. During a hosting transfer initiated from $C_U@[M, S]$ to $C'_U@[M', S']$, C should ensure that C' has suitable policy assertions $P(C')$ so that the user's policy specified in $P(C)$ is not violated. To ensure this, any policies P_{HR} specified by S' during the hosting request are added to $P(C')$ to record the fact that C' operates under that context. Second, any stateful policies need to be specially handled, e.g., consider our third invocation policy: the total budget across all third parties that are vouched for by Amazon is 100 dollars. If this constraint is to hold across both C' and any future C'' that might be derived from C, then one option is to use C as a common point during invocation to ensure that this constraint is never violated. However, this requires any transferred preserver C' to communicate with C upon invocations. This is undesirable, and further, such synchronization is not required in most web service applications. Instead, we leverage the concept of exo-leasing [26]. *Decomposable constraints* (such as budgets, number of queries answered) from $P(C)$ are split into two sub-constraints; the original constraint in $P(C)$ is updated with the first, and the second is added to $P(C')$. For instance, a budget is split between the current preserver and the transferred preserver. We currently only support additive constraints which can be split in any user-desired ratio; other kinds of constraints can be added if required.

4.3 Preserver Data Transformation

This section discusses how to provide users control over data transformations. This is different from providing invocation control; the latter controls operations invoked over the data, and the former controls the data itself. We refer to a preserver whose data is derived from a set of existing preservers as a derivative preserver. We support two data transformations towards *aiding risk mitigation*: filtering and aggregation.

Filtering: A derivative preserver obtained by filtering has a subset of the original data; for instance, only the web history in the last six months. A preserver that supports such transformations on its data exports an interface call for this purpose; this is invoked alongside a hosting protocol request so that the forwarded preserver contains a subset of the originating preserver.

Aggregation: This allows the merging of raw data from mutually trusting users of a service, so that the service can use the aggregated raw data, while the users still obtain some privacy guarantees due to aggregation. A trusted aggregator preserver can also improve efficiency in the sensitive query idiom, since it enables a (private) index across preservers, sparing them from irrelevant events. We refer to the set of aggregated users as "data crowds" (inspired by the Crowds anonymity system [23]). We describe what user actions are necessary for enabling aggregation, and then discuss how the service carries it out.

To enable aggregation, a user U instructs her preserver $C_U@[M, S]$ to aggregate her data with a set of preservers

$C_{U'}[M', S]$ where U' is a set of users that she trusts. The set of users U' form a data crowd. We envision that a user U can discover such a large enough set of such users U' by mining her social network (for instance). During preserver installation, each member U of the crowd C confides a key K_C shared among all members in the crowd to their preserver. During installation, a user $U \in C$ also notifies the service of her willingness to be aggregated in a data crowd identified by $H(K_A)$ (H is a hash function). S can then identify the set of preservers C_A belonging to that particular data crowd using $H(K_A)$ as an identifier.

To aggregate, the service expects the preserver to support an aggregation interface call. This call requests the preserver C_U to merge with $C_{U'}$ and is a simple pair-wise operation. These mergers are appropriately staged by the service that initiates the aggregation. During the aggregation operation of C_U with $C_{U'}$, preserver C_U simply encrypts its sensitive data using the shared key K_A and hands it off to U' along with its owner's key. During this aggregation, the resultant derivative preserver also maintains a list of all preservers merged into the aggregate so far; preservers are identified by the public key of the owner sent during installation. This list is required so as to prevent duplicate aggregation; such duplicate aggregation can reduce the privacy guarantees. Once the count of source preservers in an aggregated preserver exceeds a user-specified constraint, the aggregate preserver can then reveal its data to the service S. This scheme places the bulk of the aggregation functionality upon the service giving it freedom to optimize the aggregation.

5 Implementation

Our implementation supports three deployments: TTP, client-side, and Xen-based colocation. We plan to support TPMs and secure co-processors using standard implementation techniques such as late launch (e.g., Flicker [18]). We implement three preservers, one per idiom: a stock trading preserver, a targeted ads preserver, and a CCN-based charging preserver (we only describe the first two here due to space constraints). We first describe our framework, and then these preservers (details are presented in our technical report [16]).

Preserver Framework: For TTP deployment, the network isolates the preserver from the service. The colocation deployment relies on Xen. We ported and extended XenSocket [34] to our setup (Linux 2.6.29-2 / Xen 3.4-1) in order to provide fast two-way communication between the web service VM and the preserver VM using shared memory and event channels. The base layer implements policy checking by converting policies to DataLog clauses, and answers queries by a simple top-down resolution algorithm (described in SecPAL [7]; we could not use their implementation since it required .NET libraries). We use the PolarSSL library for embedded systems for light-weight cryptographic functionality, since it is a significantly smaller code base (12K) compared to OpenSSL (over 200K lines of code); this design decision can be revisited if required. We use a TPM, if available, to verify that a remote machine is running Xen using the Trousers library. Note that we only use a TPM to verify the execution of Xen; we still assume that Xen isolates correctly.

Stock Trading Preserver: We model our stock preserver after a feature in a popular day trading software, Sierra Chart [4] that makes trades automatically when an incoming stream of ticker data matches certain conditions. This preserver belongs to the query idiom and exports a single function call *TickerEvent (SYMBOL, PRICE)* that returns an *ORDER("NONE" /"BUY" / "SELL", SYMBOL, QUANTITY)* indicating whether the preserver wishes to make a trade and of what quantity. The preserver allows the user to specify two conditions (which can be arbitrarily nested boolean predicates with operations like AND, OR, and NOT, and base predicates that consist of the current price, its moving average, current position, and comparison operations): a "BUY" and a "SELL" condition. Our implementation of predicate matching is straightforward; we apply no query optimizations, so our results are only meaningful for comparison.

Targeted Advertising Preserver: We implemented two preservers for targeted advertising (serving targeted ads and building long-term models), both in the analytics idiom. They store the user's browsing history and are updated periodically (say, daily).

The first preserver is used for targeted advertising for which we implemented two possible interfaces: *ChooseAd(List of Ads, Ad Categories)* and *GetInterestVector()*. In the first, the preserver selects the ad to be displayed using a procedure followed by web services today (described in Adnostic [29]). In the second, the preserver extracts the user's interest vector from her browsing history, and then perturbs it using differential privacy techniques (details are in our technical report [16]). This preserver uses a stateful policy to record the number of queries made (since information leak in interactive privacy mechanisms increases linearly with the number of queries). The second preserver allows the service to use any proprietary algorithm in serving ads since the feature vector, which summarizes a user's detailed history, is itself revealed after being appropriately perturbed with noise. This preserver is used by the service to build long-term models related to the affinity of users with specific profiles to specific advertisements; our aggregation functionality is of use here.

Based on these preservers, we estimate the typical refactoring effort for a web service in adopting the preserver architecture. We do not have access to a stock broker's code base, so our experience is based on the targeted ads preserver and the CCN case. For the targeted ads case, given the availability of client-side plugins like Adnostic and PrivAd, we believe the refactoring effort will be minimal. In the CCN case, modifying the Zen shopping cart [6] to interact with the preserver took us only a day.

6 Evaluation

In this section, we first present the performance evaluation of our framework, followed by a security analysis. Our performance evaluation has two goals. First, evaluate *the cost of isolation in our framework* as the performance overhead and provisioning cost of client-side / TTP / colocation deployment, relative to the base case (no isolation). Second compare *various deployment options* to determine the ideal deployment for different workloads.

Experimental Scenarios: We consider three scenarios: (a) the base case, (b) the TTP case, and (c) the Xen-based colocation case. Since a TTP deployment is equivalent in most respects to a client-side preserver, we discuss them together. We compare these scenarios along three dimensions of performance: setup cost, invocation cost, data transformation cost. The term *cost* includes latency, network bandwidth consumption, and storage cost. Of these, the invocation cost is borne every time the user's data is accessed, and is thus the primary metric of comparison. The setup cost is incurred only during the initial and subsequent transfers of the user's data (since this is an infrequent event, the details are omitted from this paper; they are in our technical report [16]), while the transformation cost, though not as frequent as invocation, may be significant if the aggregation involves data belonging to large numbers of users. All results are reported over 100 runs. For latency, we report the median and the 95% confidence interval for the median; we report only the median for bandwidth since it is much less variable.

Hardware Configuration: Our test server is a 2.67 GHz quad core Intel Xeon with 5 GB memory. A desktop (Intel Pentium4 1.8 GHz processor, 1 GB memory), on the same LAN as the server, served as a TTP (or client). The bandwidth between the server and this desktop was 10 Gbps and the round-trip about 0.2 ms. To simulate a wide-area network between the client/TTP and the service, we used DummyNet [2] to artificially delay packets by a configurable parameter; the default round-trip is 10 ms.

6.1 The Cost of Isolation

We measured the performance metrics during invocation and setup, and then examine provisioning requirements. For these measurements, we used a dummy data layer that accepts an invocation payload of a specific size and returns a response of a specific size.

Invocation Costs: To examine the invocation cost across different payload sizes, we plotted the latency as a function of payload size (varied in multiples of 2 from 256 bytes to 32 KB) in Figure 3(left). At 1 KB invocation size, though the latency via Xen (1237μs) is about 800 μs worse compared to the base case (415μs), it is still significantly lower compared to the TTP case (24.37ms). We found considerable variation ranging from 900 to 4000μs with a median of 1237μs; we believe this is related to the Xen scheduling algorithm which may delay the execution of the preserver VM. This plot shows that the overhead added by Xen as a percentage of the base case declines, while the network transfer time increases the latency for the TTP case. In the TTP case, the latency is due to two round-trips (one each for exchange of TCP SYN and SYN ACKs, and the invocation), and network transfer time. The traffic for the TTP case varies roughly linearly with the size of the payload: 1.7 KB (for invocations of 256 bytes), 10 KB (4 KB), and 72 KB (32 KB); of course, in the Xen case and the base case, no network communication is involved for invocation.

Provisioning Cost: We first estimate the colocation provisioning cost and the monetary/provisioning costs for TTP/client-side preservers.

Under colocation, preserver storage requires memory and disk; our preserver code is about 400 KB (120 KB compressed) from 100 SLOC. The overhead is due to glibc which we plan to remove. We used the Difference Engine [15] which finds similar pages across VMs to reduce memory costs; thus, the memory requirements for the VMs of users who share the preserver code from the same provider are significantly reduced. In our experiments, we initially allocated 64 MB of memory to each preserver VM, and then invoked a varying number of client VMs with the same preserver. The Difference Engine saves us about 85% overall memory; the memory requirement per preserver VM is about 10 MB (this estimate was obtained by invoking 10 VMs, and then allowing time for the detection of identical and similar memory pages). At the current estimate of 10 MB per preserver, every 100 users require about 1 GB memory. We believe that this can be reduced further since we do not use any kernel facilities; thus, the preserver can run directly on the VMM.

The trade-offs are different for TTP/client-side preservers. The TTP shoulders the cost of availability and performance. Web-service hosting (with unlimited bandwidth) are about $10 - 15$ dollars per month today; expectations are higher from a *trusted* service because of the security risks, so one may expect a higher cost. In the client-side preserver, availability and performance fall on the user instead.

6.2 Performance of Various Preserver Deployments

We evaluated the stock trading preserver and targeted ads preserver for comparing the deployments since they represent different workloads (frequent small invocation versus low-frequency invocation with larger payload). The CCN preserver's results are in our technical report [16].

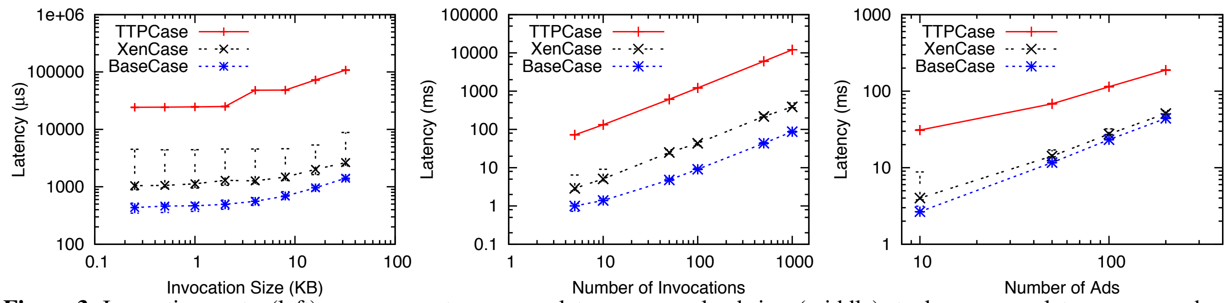


Figure 3: Invocation costs: (left) measurement preserver; latency vs. payload size, (middle) stock preserver; latency vs. number of ticker events, (right) targeted ads preservers; latency vs number of ads.

Stock Preserver: To reflect a typical trading scenario with hundreds of ticker events per second, we plotted latency versus numbers of back-to-back invocations (stock ticker events) in Figure 3(middle). As expected, the TTP case requiring network access is substantially slower as compared to the Xen case and the base case. Comparing the Xen case and the base case for a sequence of 100 back-to-back invocations, the latencies are 43.4ms and 9.05ms respectively; though the overhead due to Xen is substantial, it is still a significant improvement over the TTP case, which requires 12 seconds. Regarding the traffic, in the TTP case, the traffic generated for a sequence of $5, 10, 50, 100, 500$ and 1000 invocations are respectively $2.8, 4.8, 20.2, 39.4, 194, 387$ KB respectively. Thus, the bandwidth required for say, 500 events per second, is about 1.6 MB/s per user per stock symbol (and a single user typically trades multiple symbols). These results reflect that server site colocation can offer significant benefit in terms of network bandwidth.

Targeted Ads Preserver: We present two results for the targeted ads scenario. Figure 3(right) shows the latency per invocation for the preserver to serve targeted ads using the *ChooseAd* interface (we do not show results for the *GetInterestVector* interface due to lack of space). This graph shows that the Xen preserver has nearly the same overhead as the Base case preserver once the number of ads out of which the preserver selects one exceeds 5. This is mainly because the payload size dominates the transfer time; the context switch overhead is minimal. In the TTP case, the latency is clearly impacted by the wide-area network delay. The traffic generated in the TTP case was measured to be $2.6, 8.3, 15.8, 29.2$ KB for $10, 50, 100, 200$ ads respectively. These reflect the bandwidth savings of colocation; since this bandwidth is incurred for every website the user visits, this could be significant. Our technical report [16] has performance results on two other operations: allowing a user to update her web history and the data aggregation transformation.

6.3 Security Analysis

We now discuss the desirable security properties of our preserver, determine the TCB for these to hold, and argue that our TCB is a significant improvement over the status quo. Our security goal is that any data access or preserver transfer obeys the user-sanctioned interface and policies. Depending on the deployment, the adversary either has control of the service software stack (above the VMM layer) or has physical access to the machine hosting the preserver; the first applies to VMMs, TPMs, and the second to TTPs, client-side preservers, secure co-processors. In the TTP/client case, we rely on physical isolation and the preserver implementation's correctness; the rest of this section is concerned with the security goal for colocation.

Colocation has one basic caveat: the service can launch an availability attack by refusing to service requests using its host hub or by preventing a user from updating her preserver. We assume that the trusted module provides the four security properties detailed in Section 3: isolation, anti-replay protection, random number generation, and remote attestation. We also assume that any side-channels (such as memory page caching, CPU usage) have limited bandwidth (a subject of ongoing research [30]). We first analyze the installation protocol and then examine the invocation protocol.

Installation Protocol: The installation protocol is somewhat complex since it involves a Diffie-Hellman key exchange along with remote attestation. In order to argue its correctness, we now present an informal argument (our technical report [16] has a formal argument in LS^2 [10], a formal specification language). Our informal argument is based on three salient features of the installation protocol (Section 4.1). First, the attestation guarantees that the public key is generated by the base layer. Second, the Diffie-Hellman-based installation protocol ensures confidentiality; only the party that generated the public key itself can decrypt the preserver. From the first observation, this party can only be the base layer. This relies on the random number generation ability of the trusted module to ensure that the generated public key is truly random. Third, since attestation verifies the purported input and output to the code which generated the public key, man-in-the-middle attacks are ruled out; an adversary cannot tamper with the attestation without rendering it invalid. These three properties together ensure that the preserver can only be decrypted by the base layer, thus ruling out any leakage during installation. The correctness of the base layer's implementation helps argue that the preserver is correctly decrypted and instantiated at the service.

Invocation Protocol: Upon invocation, the base layer verifies: (a) the invoking principal's identity, (b) any supporting policies, and (c) that user-specified policies along with supporting policies allow the principal the privilege to make the particular invocation. The principal's identity is verified either against a list of public keys sent during installation (which binds service names to their public keys or offloads trust to a certification authority). In either case, the correctness of installation ensures that the identity cannot be spoofed. The base layer verifies the supporting policies by verifying each statement of the form $X\ SAYS\ \cdots$ against the signature of X. The policy resolver takes two sets of policies as input: user-specified policies and the invoker's supporting policies. The latter, we have argued, is correct; for the former, we rely on the anti-replay property provided by the trusted module to ensure that the preserver's policies (which can be updated over time) are up-to-date and reflects the outcome of all past invocations. This ensures that any stateful constraints are correctly ensured. Once invocation is permitted by the base layer, it is passed on to the data layer which implements the interface. In cases such as a query preserver or an analytics preserver, this functionality is carried out entirely within the data layer, which we assume to be correct. For the proxy preserver, which requires network access, we note that though the network stack is itself offloaded to the host hub, the SSL library resides in the preserver; thus the host hub cannot compromise confidentiality or integrity of network communication.

6.3.1 TCB Estimate

From the preceding discussion, it is clear that our TCB includes: (A) the trust module, (B) the data layer interface and implementation, and (C) the base layer protocols and implementation. In our colocation implementation, (A) is the Xen VMM and Dom0 kernel; we share these with other VMM-based security architectures (e.g., Terra [13]). Mechanisms to improve VMM-based security (e.g., disaggregation [19] removes Dom0 from the TCB) also apply to our framework. Regarding the base layer and the data layer, their per-module LOC estimates are: Base Layer (6K), PolarSSL (12K), XenSocket (1K), Trousers (10K), Data Layers (Stock Preserver: 340, Ads: 341, CCN: 353). This list omits two implementation dependencies we plan to remove. First, we boot up the preserver atop Linux 2.6.29-2; however, our preservers do not utilize any OS functionality (since device-drivers, network stack, etc., are provided by the host hub), and can be ported to run directly atop Xen or MiniOS (a barebones OS distributed with Xen). Second, the preservers use *glibc*'s memory allocation and a few STL data structures; we plan to hand-implement a custom memory allocator to avoid these dependencies. We base our trust in the data layer interface and implementation in the interface's simplicity. Currently, despite our efforts at a simple design, the base layer is more complex than the data layer, as reflected in the LOC metric. In the lack of a formal argument for correctness, for now, our argument is that even our complex base layer offers a significant improvement in security to users who have no choice today but to rely on unaudited closed source service code.

7 Related Work

Before examining broader related work, we discuss three closely related papers: Wilhelm's thesis [31], CLAMP [21], and BStore [9]. Work in the mobile agent security literature, such as Wilhelm's thesis [31], leverages mobile agents (an agent is code passed around from one system to another to accomplish some functionality) to address data access control in a distributed system. Our main differences are: (a) our interface is data dependent and its invocation can be user-controlled, and (b) preservers return some function of the secret data as output; this output provides some useful secrecy guarantees to the user. CLAMP [21] rearchitects a web service to isolate various clients by refactoring two security-critical pieces into stand-alone modules: a query restrictor (which guards a database) and a dispatcher (which authenticates the user). Our goal is different: it is to protect an individual user's data (as opposed to a shared database), both from external and internal attacks. BStore [9] argues for the decoupling of the storage component from the rest of the web service: users entrust their files to a third party storage service which enforces policies on their behalf. Our preserver architecture is in a similar spirit, except that it pertains to *all* aspects of how data is handled by the web service, not just storage; the enforcement of an interface means that user data is never directly exposed to the web service. Our work is also related to the following areas:

Information Flow Control (IFC): The principle of IFC has been implemented in OSes (e.g., Asbestos [12]) and programming languages (e.g., JIF [24]), and enables policy-based control of information flow between security compartments. IFC has also been used to build secure web service frameworks, e.g., W5 [17], xBook [27]. Preservers provide data access control; this is complementary to IFC's data propagation control. Preservers rely on an interface that offers sufficient privacy to the user and is usable to the service. The advantage of interface-based access control is that we can rely on a variety of isolation mechanisms without requiring a particular OS or programming language. Further, the interface's simplicity makes it feasible to envision proving a preserver's correctness; doing so in the IFC case requires one to prove the correctness of the enforcement mechanism (OS or compiler) which can be significantly more complex.

Decentralized Frameworks For Web Services: Privacy frameworks that require only support from users have been proposed as an alternative to web services. VIS [8] maintains a social network in a completely decentralized fashion by users hosting their data on trusted parties of their own choice; there is no centralized web service. Preservers are more compatible with the current ecosystem of a web service storing users' data. NOYB [14] and LockR [28] are two recent proposals that use end-to-end encryption in social network services; both approaches are specific to social networks, and their mechanisms can be incorporated in the preserver framework, if so desired.

8 Conclusion

Our preserver framework rearchitects web services around the principle of giving users control over the code and policies affecting their data. This principle allows a user to decouple her data privacy from the services she uses. Our framework achieves this via *interface-based access control*, which applies to a variety of web services (though not all). Our *colocation* deployment model demonstrates that this property can be achieved with moderate overhead, while more stringent security can be obtained with other deployment models. In the future, we hope to formalize two aspects of our framework. First, we wish to prove the correctness of our interface implementation. Second, we hope to define and prove precise guarantees on information leak via interfaces.

Acknowledgments: We thank Gautam Altekar, Kevin Fall, Jon Howell, Eddie Kohler, Jay Lorch, Radia Perlman, and the anonymous reviewers for their careful feedback on our work, as well as the authors of Adnostic, in particular Arvind Narayanan, for sharing their data.

References

[1] DataLoss DB: Open Security Foundation. `http://datalossdb.org`.

[2] DummyNet Homepage. `http://info.iet.unipi.it/~luigi/dummynet/`.

[3] OpenSocial. `http://code.google.com/apis/opensocial/`.

[4] Sierra Chart: Financial Market Charting and Trading Software. `http://sierrachart.com`.

[5] TPM Main Specification Level 2 Version 1.2, Revision 103 (Trusted Computing Group). `http://www.trustedcomputinggroup.org/resources/tpm_main_specification/`.

[6] Zen E-Commerce Solution. `http://www.zen-cart.com/`.

[7] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and Semantics of a Decentralized Authorization Language. In *Proc. IEEE Computer Security Foundations Symposium*, 2006.

[8] R. Cáceres, L. Cox, H. Lim, A. Shakimov, and A. Varshavsky. Virtual Individual Servers as Privacy-Preserving Proxies for Mobile Devices. In *Proc. MobiHeld*, 2009.

[9] R. Chandra, P. Gupta, and N. Zeldovich. Separating Web Applications from User Data Storage with BStore. In *WebApps*, 2010.

[10] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A Logic of Secure Systems and its Application to Trusted Computing. In *IEEE Security and Privacy*, 2009.

[11] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S. Smith. Building the IBM 4758 Secure Coprocessor. *Computer*, 34(10), 2001.

[12] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. In *SOSP*, 2005.

[13] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *SOSP*, 2003.

[14] S. Guha, K. Tang, and P. Francis. NOYB: Privacy in Online Social Networks. In *Proc. WOSP*, 2008.

[15] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *OSDI*, 2008.

[16] J. Kannan, P. Maniatis, and B.-G. Chun. A Data Capsule Framework For Web Services: Providing Flexible Data Access Control To Users. arXiv:1002.0298v1 [cs.CR].

[17] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A World Wide Web Without Walls. In *Proc. HotNets*, 2007.

[18] J. M. Mccune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys*, 2008.

[19] D. G. Murray, G. Milos, and S. Hand. Improving Xen Security through Disaggregation. In *VEE*, 2008.

[20] G. C. Necula. Proof-carrying code: Design, Implementation, and Applications. In *Proc. PPDP*, 2000.

[21] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *IEEE Security and Privacy*, 2009.

[22] L. Ponemon. Fourth Annual US Cost of Data Breach Study. `http://www.ponemon.org/local/upload/fckjail/generalcontent/18/file/2008-2009 US Cost of Data Breach Report Final.pdf`, 2009. Retrieved Feb 2010.

[23] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for Web Transactions. *ACM Transactions on Information and System Security*, 1(1), 1998.

[24] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE JSAC*, 21, 2003.

[25] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP*, 2007.

[26] L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow Synchronization for Mobile Clients of Commodity Storage Servers. In *Middleware*, 2008.

[27] K. Singh, S. Bhola, and W. Lee. xBook: Redesigning Privacy Control in Social Networking Platforms. In *USENIX Security*, 2009.

[28] A. Tootoonchian, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: Better Privacy for Social Networks. In *CoNEXT*, 2009.

[29] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *NDSS*, 2010.

[30] E. Tromer, A. Chu, T. Ristenpart, S. Amarasinghe, R. L. Rivest, S. Savage, H. Shacham, and Q. Zhao. Architectural Attacks and their Mitigation by Binary Transformation. In *SOSP*, 2009.

[31] U. G. Wilhelm. *A Technical Approach to Privacy based on Mobile Agents Protected by Tamper-Resistant Hardware*. PhD thesis, Lausanne, 1999.

[32] B. S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.

[33] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving Application Security with Data Flow Assertions. In *SOSP*, 2009.

[34] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. XenSocket: A High-Throughput Interdomain Transport for Virtual Machines. In *Middleware*, 2007.

BenchLab: An Open Testbed for Realistic Benchmarking of Web Applications

Emmanuel Cecchet, Veena Udayabhanu, Timothy Wood, Prashant Shenoy
University of Massachusetts Amherst
{cecchet,veena,twood,shenoy}@cs.umass.edu

Abstract

Web applications have evolved from serving static content to dynamically generating Web pages. Web 2.0 applications include JavaScript and AJAX technologies that manage increasingly complex interactions between the client and the Web server. Traditional benchmarks rely on browser emulators that mimic the basic network functionality of real Web browsers but cannot emulate the more complex interactions. Moreover, experiments are typically conducted on LANs, which fail to capture real latencies perceived by users geographically distributed on the Internet. To address these issues, we propose BenchLab, an open testbed that uses real Web browsers to measure the performance of Web applications. We show why using real browsers is important for benchmarking modern Web applications such as Wikibooks and demonstrate geographically distributed load injection for modern Web applications.

1. Introduction

Over the past two decades, Web applications have evolved from serving primarily static content to complex Web 2.0 systems that support rich JavaScript and AJAX interactions on the client side and employ sophisticated architectures involving multiple tiers, geographic replication and geo-caching on the server end. From the server backend perspective, a number of Web application frameworks have emerged in recent years, such as Ruby on Rails, Python Django and PHP Cake, that seek to simplify application development. From a client perspective, Web applications now support rich client interactivity as well as customizations based on browser and device (e.g., laptop versus tablet versus smartphone). The emergence of cloud computing has only hastened these trends---today's cloud platforms (e.g., Platform-as-a-Service clouds such as Google AppEngine) support easy prototyping and deployment, along with advanced features such as autoscaling.

To fully exploit these trends, Web researchers and developers need access to modern tools and benchmarks to design, experiment, and enhance modern Web systems and applications. Over the years, a number of Web application benchmarks have been proposed for use by the community. For instance, the research community has relied on open-source benchmarks such as TPC-W [16] and RUBiS [13] for a number of years; however these benchmarks are outdated and do not fully capture the complexities of today's Web 2.0 applications and their workloads. To address this limitation, a number of new benchmarks have been proposed, such as TPC-E, SPECWeb2009 or SPECjEnterprise2010. However, the lack of open-source or freely available implementations of these benchmarks has meant that their use has been limited to commercial vendors. CloudStone [15] is a recently proposed open-source cloud/Web benchmark that addresses some of the above issues; it employs a modern Web 2.0 application architecture with load injectors relying on a Markov model to model user workloads. Cloudstone, however, does not capture or emulate client-side JavaScript or AJAX interactions, an important aspect of today's Web 2.0 applications and an aspect that has implications on the server-side load.

In this paper, we present BenchLab, an open testbed for realistic Web benchmarking that addresses the above drawbacks. BenchLab's server component employs modern Web 2.0 applications that represent different domains; currently supported server backends include Wikibooks (a component of Wikipedia) and CloudStone's Olio social calendaring application, with support for additional server applications planned in the near future. BenchLab exploits modern virtualization technology to package its server backends as virtual appliances, thereby simplifying the deployment and configuration of these server applications in laboratory clusters and on public cloud servers. BenchLab supports Web performance benchmarking "at scale" by leveraging modern public clouds---by using a number of cloud-based client instances, possibly in different geographic regions, to perform scalable load injection. Cloud-based load injection is cost-effective, since it does not require a large hardware infrastructure and also captures Internet round-trip times. In the design of BenchLab, we make the following contributions:

- We provide empirical results on the need to capture the behavior of real Web browsers during Web load injection. Our results show that traditional trace replay methods are no longer able to faithfully emulate modern workloads and exercise client and server-side functionality of modern Web applications. Based on this insight, we design BenchLab to use real Web browsers, in conjunction with automated tools, to inject Web requests to the server application. As noted above, we show that our load injection process can be scaled by leveraging inexpensive client instances on public cloud platforms.

- Similar to CloudStone's Rain [2], BenchLab provides a separation between the workload modeling/generation and the workload injection during benchmark execution. Like Rain, BenchLab supports the injection of real Web traces as well as synthetic ones generated from modeling Web user behavior. Unlike Rain, however, BenchLab uses real browsers to inject the requests in these traces to faithfully capture the behavior of real Web users.
- BenchLab is designed as an open platform for realistic benchmarking of modern Web applications using real Web browsers. It employs a modular architecture that is designed to support different backend server applications. We have made the source code for BenchLab available, while also providing virtual appliance versions of our server application and client tools for easy, quick deployment.

The rest of this document is structured as follows. Section 2 explains why realistic benchmarking is an important and challenging problem. Section 3 introduces BenchLab, our approach to realistic benchmarking based on real Web browsers. Section 4 describes our current implementation that is experimentally evaluated in section 5. We discuss related work in section 6 before concluding in section 7.

2. Why Realistic Benchmarking Matters

A realistic Web benchmark should capture, in some reasonable way, the behavior of modern Web applications as well as the behavior of end-users interacting with these applications. While benchmarks such as TPC-W or RUBiS were able to capture the realistic behavior of Web applications from the 1990s, the fast paced technological evolution towards Web 2.0 has quickly made these benchmarks obsolete. A modern Web benchmark should have realism along three key dimensions: (i) a realistic server-side application, (ii) a realistic Web workload generator that faithfully emulates user behavior, and (iii) a realistic workload injector that emulates the actual "browser experience." In this section, we describe the key issues that must be addressed in each of these three components when constructing a Web benchmark.

2.1. Realistic applications

The server-side component of the benchmark should consist of a Web application that can emulate common features of modern Web applications. These features include:

Multi-tier architecture: Web applications commonly use a multi-tier architecture comprising at least of a database backend tier, where persistent state is stored, and a front-end tier, where the application logic is implemented. In modern applications, this multi-tier architecture is often implemented in the form of a Model-View-Controller (MVC) architecture, reflecting a similar partitioning. A number of platforms are available to implement such multi-tier applications. These include traditional technologies such as JavaEE and PHP, as well as a number of newer Web development frameworks such as Ruby on Rails, Python Django and PHP Cake. Although we are less concerned about the idiosyncrasies of a particular platform in this work, we must nevertheless pay attention to issues such as the scaling behavior and server overheads imposed by a particular platform.

Rich interactivity: Regardless of the actual platform used to design them, modern Web applications make extensive use of JavaScript, AJAX and Flash to enable rich interactivity in the application. New HTML5 features confirm this trend. In addition to supporting a rich application interface, such applications may incorporate functionality such as "auto complete suggestions" where a list of completion choices is presented as a user types text in a dialog or a search box; the list is continuously updated as more text is typed by the user. Such functions require multiple round trip interactions between the browser and the server and have an implication on the server overheads.

Scaling behavior: To scale to a larger number of users, an application may incorporate techniques such as replication at each tier. Other common optimizations include use of caches such as memcached to accelerate and scale the serving of Web content. When deployed on platforms such as the cloud, it is even feasible to use functions like auto-scaling that can automatically provision new servers when the load on existing ones crosses a threshold.

Domain: Finally, the "vertical" domain of the application has a key impact on the nature of the server-side workload and application characteristics. For example, "social" Web applications incorporate different features and experience a different type of workload than say, Web applications in the financial and retail domains. Although it is not feasible for us to capture the idiosyncrasies of every domain, our open testbed is designed to support any application backend in any domain. We presently support two backends: Wikibooks [20] (a component of Wikipedia [21]) and CloudStone's Olio [12] social calendaring application, with support for additional server applications planned in the future.

2.2. Realistic load generation

Realistic load generation is an important part of a benchmark. The generated workload should capture real user behavior and user interactions with the application. There are two techniques to generate the workload for a benchmark. In the first case, we can use real workload data to seed or generate the workload for the benchmark; in the simplest case, the real workload is replayed during benchmark execution. The advantage of this approach is that it is able to capture real user behavior. However, real workload data may not always be available. Further, the data may represent a particular set of benchmark parameters and it is not always easy to change these parameters (e.g., number of concurrent users, fraction of read and write requests, etc) to suit the benchmarking needs. Consequently many benchmarks rely on synthetic workload generators. The generators model user behavior such as think times as well as page popularities and other workload characteristics. Cloudstone, for instance, uses a sophisticated Markov model to capture user behavior [15]. The advantage of synthetic workload generation is that it allows fine-grain control over the parameters that characterize the workload [8].

BenchLab does not include a custom Web workload generation component. Rather it is designed to work with any existing workload generator. This is done by decoupling the workload generation step from the workload injection. In many benchmarks, workload generation and injection are tightly coupled—a request is injected as soon as it is generated. BenchLab assumes that workload generation is done separately and the output is stored as a trace file. This trace data is then fed to the injection process for replay to the server application. This decoupling, which is also used by Rain [2], allows the flexibility of using real traces for workload injection (as we do for our Wikibooks backend) as well as the use of any sophisticated synthetic workload generator.

2.3. Realistic load injection

Traditionally Web workload injection has been performed using trace replay tools such as *httperf* that use one or a small number of machines to inject requests at a high rate to the application. The tools can also compute client-side statistics such as response time and latency of HTTP requests. This type of workload injection is convenient since it allows emulating hundreds of virtual users (sometimes even more) from a single machine, but it has limited use for many applications that adjust behavior based on a client's IP address. In some scenarios, such as testing real applications prior to production deployment, this can be problematic since many requests originating from the same IP address can trigger the DDoS detection mechanisms if any. More importantly, this approach does not realistically test IP-based localization services or IP-based load balancing.

An important limitation of trace replay-based techniques is that they fall short of reproducing real Web browser interactions as they do not execute JavaScript or perform AJAX interactions. As a result, they may even fail to generate requests that would be generated by a real browser. Even the typing speed in a text field can have an impact on the server load since each keystroke can generate a request to the server like with Google Instant. Such interactions are hard to capture using trace replay tools.

Modern applications also include browser-specific customizations; they may send out custom style sheets and custom JavaScript depending on the browser type. The same application may also send a vastly different version of a page to a mobile or a tablet browser than a traditional desktop-class browser.[1] Moreover, each browser has different optimizations to fetch the content of Web pages in parallel and to render them quickly. Thus, the browser mix can impact the load seen by the server applications, even for a fixed number of users.

Finally, the replay tools typically report the response time of individual requests, rather than page load times seen by a browser—typically a Web page can include tens of components, including style sheets, images, ads and others components, and the response time for a page should include the time to load and render all of these components from a browser standpoint.

To capture these subtleties, we argue for the use of real Web browsers to drive the load injection. This is achieved by using automated tools that interact with a browser UI like a real user would and to issue requests from the browser, using the traces generated by the workload generation process. Having a variety of real Web browsers with various configurations and plugins improves the accuracy of benchmarking the real user experience of a Web application.

3. BenchLab

BenchLab is an open testbed for Web application benchmarking. It can be used with any standard benchmark application as well as real Web applications (section 3.2). Applications can have multiple datasets and workloads (section 3.3), and load injection is performed by real Web browsers (section 3.4).

3.1. Overview

Figure 1 gives an overview of the BenchLab components and how they interact to run an experiment. The

[1] Typically web applications redirect users from mobile devices to a separate mobile version of the application. However some recent applications have embedded support for mobile browsers within the main application.

BenchLab WebApp is the central piece that controls experiments. It is a Java Web application that can be deployed in any Java Web container such as Apache Tomcat. The BenchLab WebApp provides a Web interface to interact with experimenters that want to manage experiments and automated Web browsers that are executing experiments.

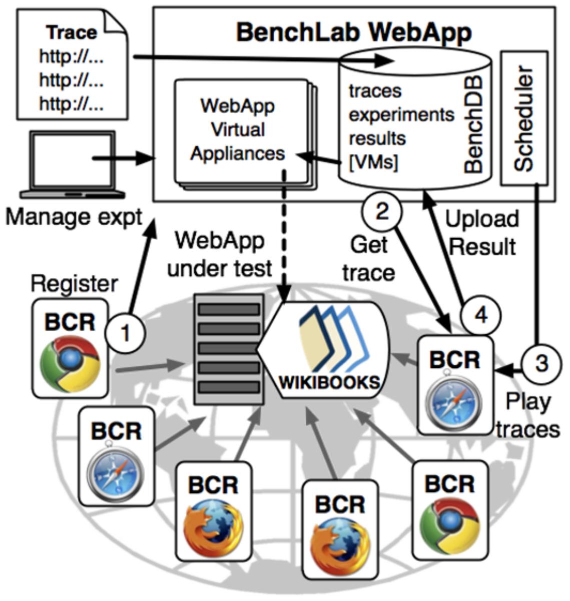


Figure 1. BenchLab experiment flow overview.

Web traces can be recorded from a live system or generated statically (see section 3.3). Trace files are uploaded by the experimenter through a Web form and stored in the BenchLab database. Virtual machines of the Web Application under test can also be archived so that traces, experiments and results can be matched with the correct software used. However BenchLab does not deploy, configure or monitor any server-side software. There are a number of deployment frameworks available that users can use depending on their preferences (Gush, WADF, JEE, .Net deployment service, etc) Server side monitoring is also the choice of the experimenter (Ganglia and fenxi are popular choices). It is the responsibility of the user to deploy the application to be tested. Note that anyone can deploy a BenchLab WebApp and therefore build his or her own benchmark repository.

An experiment defines what trace should be played and how. The user defines how many Web browsers and eventually which browsers (vendor, platform, version …) should replay the sessions. If the trace is not to be replayed on the server it was recorded, it is possible to remap the server name recorded in the URLs contained in the trace to point to another server that will be the target of the experiment.

The experiment can start as soon as enough browsers have registered to participate in the experiment or be scheduled to start at a specific time. The BenchLab WebApp does not deploy the application nor the client Web browsers, rather it waits for browsers to connect and its scheduler assigns them to experiments.

The BenchLab client runtime (BCR) is a small program that starts and controls a real Web browser on the client machine. The BCR can be started as part of the booting process of the operating system or started manually on-demand. The BCR connects the browser to a BenchLab WebApp (step 1 in Figure 1). When the browser connects to the WebApp, it provides details about the exact browser version and platform runtime it currently executes on as well as its IP address. If an experiment needs this browser, the WebApp redirects the browser to a download page where it automatically gets the trace for the session it needs to play (step 2 in Figure 1). The BCR stores the trace on the local disk and makes the Web browser regularly poll the WebApp to get the experiment start time. There is no communication or clock synchronization between clients, they just get a start time as a countdown in seconds from the BenchLab WebApp that informs them 'experiment starts in x seconds'. The activity of Web browsers is recorded by the WebApp and stored in the database for monitoring purposes.

When the start time has been reached, the BCR plays the trace through the Web browser monitoring each interaction (step 3 in Figure 1). If Web forms have to be filled, the BCR uses the URL parameters stored in the trace to set the different fields, checkboxes, list selections, files to upload, etc. Text fields are replayed with a controllable rate that emulates human typing speed. The latency and details about the page are recorded (number of div sections, number of images, size of the page and title of the page) locally on the client machine. The results are uploaded to the BenchLab WebApp at the end of the experiment (step 4 in Figure 1).

Clients replay the trace based on the timestamps contained in the trace. If the client happens to be late compared to the original timestamp, it will try to catch up by playing requests as fast as it can. A global timeout can be set to limit the length of the experiment and an optional heartbeat can also be set. The heartbeat can be used for browsers to report their status to the BenchLab WebApp, or it can be used by the WebApp to notify browsers to abort an experiment.

3.2. Application backends

Our experience in developing and supporting the RUBiS benchmark for more than 10 years, has shown that users always struggle to setup the application and the different tools. This is a recurrent problem with benchmarks where more time is spent in installation and configuration rather than experimentation and measurement. To address this issue, we started to release RUBiSVA, a Virtual Appliance of RUBiS [13], i.e., a virtual machine with the software stack already configured and ready to use. The deployment can be automated on any platform that supports virtualization.

Virtualization is the de-facto technology for Web hosting and Web application deployment in the cloud. Therefore, we have prepared virtual appliances of standard benchmarks such as RUBiS, TPC-W [16] and CloudStone [15] for BenchLab. This allows reproducing experiments using the exact same execution environment and software configuration and will make it easier for researchers to distribute, reproduce and compare results.

As BenchLab aims at providing realistic applications and benchmarks, we have also made virtual appliances of Wikibooks [20]. Wikibooks provides a Web application with a similar structure to Wikipedia [21], but a more easily managed state size (GBs instead of TBs). More details about our Wikibooks application backend are provided in section 4.4.2.

3.3. Workload definitions

Most benchmarks generate the workload dynamically from a set of parameters defined by the experimenter such as number of users, mix of interactions, and arrival rate. Statistically, the use of the same parameters should lead to similar results between runs. In practice the randomness used to emulate users can lead to different requests to the Web application that require very different resources depending on the complexity of the operations or the size of the dataset that needs to be accessed. Consequently, the variance in the performance observed between experiments using the same parameters can be large. Therefore, it is necessary to decouple the request generation from the request execution so that the exact same set of requests can be replayed at will.

This can be done with little effort by instrumenting existing load generators and logging the requests that are made to the server. The resulting trace file can then be replayed by a generic tool like httperf or a more realistic injector like a real Web browser.

The traces used in BenchLab are based on the standard HTTP archive (HAR) format [9]. This format captures requests with their sub-requests, post parameters, cookies, headers, caching information and timestamps. Parameters include text to type in text fields, files to upload, boxes to check or buttons to click, etc. In the case of real applications, these traces can also be generated from an HTTP server access log to reproduce real workloads. As Web browsers automatically generate sub-requests to download the content of a page (cascading style sheets (.css), JavaScript code (.js), image files, etc), only main requests from the trace file are replayed.

Defining and generating workloads are beyond the scope of BenchLab. BenchLab focuses on the execution and replay of existing traces. Traces are stored in a database to be easily manipulated and distributed to injectors. Traces cannot be scaled up, they are either replayed completely or partially (a subset of the sessions in the trace). This means that if a trace contains 100 user sessions, it can be replayed by at most 100 clients. If a trace needs to be scaled, the user must use her workload generator to generate a scaled trace.

3.4. Web browser load injection

A central contribution of BenchLab is the ability to replay traces through real Web browsers. Major companies such as Google and Facebook already use new open source technologies like Selenium [14] to perform functional testing. These tools automate a browser to follow a script of actions, and they are primarily used for checking that a Web application's interactions generate valid HTML pages. We claim that the same technology can also be used for performance benchmarking.

BenchLab client runtime can be used with any Web browser supported by Selenium: Firefox, Internet Explorer, Chrome and Safari. Support for mobile phones with Webkit-based Web browsers is also under development. The functionality of BenchLab on the client side is limited to downloading a trace, replaying it, recording response times and uploading response times at the end of the replay. This small runtime is deployable even on devices with limited resources such as smartphones. Unlike commercial offerings, BenchLab clients can be deployed on public clouds or any other computing resource (e.g., desktop, smartphone).

Unlike traditional load injectors that work at the network level, replaying through a Web browser accurately performs all activities such as typing data in Web forms, scrolling pages and clicking buttons. The typing speed in forms can also be configured to model a real user typing. This is particularly useful when inputs are processed by JavaScript code that can be triggered on each keystroke. Through the browser, BenchLab captures the real user perceived latency including network transfer, page processing and rendering time.

4. Implementation

BenchLab is implemented using open source software and is also released as open source software for use by the community. The latest version of the software and documentation can be found on our Web site [3].

4.1. Trace recorder

We have implemented a trace recorder for Apache httpd that collects request information from a live system using the standard httpd logging mechanisms (mod_log_config and mod_log_post). We then process these logs to generate traces in HAR format. We have contributed a new Java library called HarLib to manage HAR traces in files and databases.

Additionally we can record HTML pages generated using mod_dumpio. This is useful to build tools that will check the consistency of Web pages obtained during replay against the originally captured HTML.

4.2. Browser based load injection

We use the Selenium/Webdriver [14] framework that provides support for Firefox, Internet Explorer and Chrome on almost all the platforms (Linux, Windows, MacOS) where they are available. Safari support is experimental as well as Webkit based browsers for Android and iPhone. The BenchLab client runtime (BCR) is a simple Java program interfacing with Selenium. We currently use Selenium 2.0b3 that includes Webdriver. The BCR can start any Firefox, IE or Chrome browser installed on the machine and connect it to a BenchLab WebApp. On Linux machines that do not have an X server environment readily available, we use X virtual frame buffer (Xvfb) to render the browser in a virtual X server. This is especially useful when running clients in the cloud on machines without a display.

When a browser is assigned to an experiment, the BCR downloads the trace it has to replay through the browser and stores it in a local file. The information about the experiment, trace and session being executed by the browser is encoded by the BenchLab WebApp in cookies stored in the Web browser.

The BCR parses the trace file for the URLs and encoded parameters that are then set in the corresponding forms (text fields, button clicks, file uploads, etc.). When a URL is a simple “GET” request, the BCR waits according to the timestamp before redirecting the browser to the URL. When a form has to be filled before being submitted, the BCR starts filling the form as soon as the page is ready and just waits before clicking the submit button. As we emulate the user typing speed it can take multiple minutes to fill some forms like edits to a wiki page with Wikipedia.

The BCR relies on the browser’s performance profiling tools to record detailed timings in HAR format. This includes network level performance (DNS resolution, send/wait/receive time…) and browser level rendering time. The entire HTML page and media files can be recorded for debugging purposes if the client machine has enough storage space. An alternative compressed CSV format is also available to record coarser grain performance metrics on resource constrained devices.

We have built Xen Linux virtual machines with the BCR and Firefox to use on private clouds. We also built Amazon EC2 AMIs for both Windows and Linux with Firefox, Chrome and Internet Explorer (Windows only for IE). These AMIs are publicly available.

4.3. BenchLab WebApp

The BenchLab WebApp is a Java application implemented with JSP and Servlets. It uses an embedded Apache Derby database. Each trace and experiment is stored in separate tables for better scalability. Virtual machines of Web applications are not stored in the database but we store a URL to the image file that can point to the local file system or a public URL such as an S3 URL if the images are stored in the Amazon Simple Storage Service.

The user interface is intentionally minimalist for efficiency and scalability allowing a large number of browsers to connect. BenchLab makes a minimal use of JavaScript and does not use AJAX to keep all communications with clients purely asynchronous. Similarly no clock synchronization is needed nor required.

As the BenchLab WebApp is entirely self-contained, it can easily be deployed on any Java Web application server. We currently use Apache Tomcat 6 for all our experiments. We have tested it successfully on Linux and Windows platforms, but it should run on any platform with a Java runtime.

The BenchLab WebApp acts as a repository of traces, benchmark virtual machines and experiments with their results. That data can be easily downloaded using any Web browser or replicated to any other BenchLab WebApp instance.

4.4. Application backends

We provide Xen virtual machines and Amazon AMIs of the CloudStone benchmark and the Wikibooks application on our Web site [3]. As BenchLab does not impose any deployment or configuration framework, any application packaged in a VM can be used as a benchmark backend.

4.4.1. CloudStone

CloudStone [15] is a multi-platform, multi-language benchmark for Web 2.0 and Cloud Computing. It is composed of a load injection framework called Faban, and a social online calendar Web application called Olio [12]. A workload driver is provided for Faban to emulate users using a Markov model.

We have chosen the PHP version of Olio and packaged it in a virtual machine that we will refer to as *OlioVM*. OlioVM contains all the software dependencies to run Olio including a MySQL database and the Java Webapp implementing a geocoding service.

Faban is packaged in another VM with the load injection driver for Olio. We refer to this VM as *FabanVM*. Faban relies on the Apache HttpClient v3 (HC3) library [1] for the HTTP transport layer to interact with the Web application. We have instrumented Faban to record the requests sent to HC3 in order to obtain trace files with all needed parameters for interactions that require user input in POST methods. FabanVM is not used for load injection in our experiments but only to generate traces that can be replayed using our replay tool. The replay tool is a simple Java program replaying HTTP requests using the HC3 library.

As part of this work, we fixed a number of issues such as the workload generator producing invalid inputs for the Olio calendaring applications (e.g., invalid phone numbers, zip codes, state name). We process trace files to fix erroneous inputs and use these valid input traces in all experiments except in section 5.3.3 where we evaluate the impact of invalid inputs.

4.4.2. Wikibooks

Wikibooks [20] is a wiki of the Wikimedia foundation and provides free content textbooks and annotated texts. It uses the same Wikimedia wiki software as Wikipedia which is a PHP application storing its data in a MySQL database. Our Wikibooks application backend includes all Wikimedia extensions necessary to run the full Web site including search engine and multimedia content.

The Wikibooks virtual appliance is composed of two virtual machines. One virtual machine contains the Wikimedia software and all its extensions and the other VM runs the database. Database dumps of the Wikibooks content are freely available from the Wikimedia foundation in compressed XML format. We currently use a dump from March 2010 that we restored into a MySQL database. Real Wikibooks traces are available from the Wikibench Web site [19].

Due to copyright issues, the multimedia content in Wikibooks cannot be redistributed, and therefore, we use a multimedia content generator that produces images with the same specifications as the original content but with random pixels. Such multimedia content can be either statically pre-generated or produced on-demand at runtime.

4.5. Limitations

Our current implementation is limited by the current functionality of the Selenium/Webdriver tools we are using. Support for Firefox on all platforms and Internet Explorer on Windows are overall stable though performance may vary on different OS versions. The Chrome driver does not support file upload yet but it provides experimental access to Webkit browsers such as Safari and Android based browsers.

Our prototype does not support input in popup windows but we are able to discard JavaScript alert popups when erroneous input is injected into forms.

The current BenchLab WebApp prototype does not implement security features such as browser authentication, data encryption or result certification.

5. Experimental Results

5.1. Experimental setup and methodology

For all our experiments, the Web applications run on an 8-core AMD Opteron 2350 server, 4GB RAM with a Linux 2.6.18-128.1.10.el5xen 64 bit kernel from a standard CentOS distribution. We use the Xen v3.3.0 hypervisor. The server is physically located in the data center of the UMass Amherst campus.

CloudStone is configured with 1 virtual CPU (vCPU) and 512MB of memory for OlioVM. The Olio database is initialized for 500 users. FabanVM is allocated 1 vCPU and 1024MB of memory and runs on a different physical machine. Wikibooks VMs are both allocated 4 vCPUs and 2GB of RAM.

Experiments using Amazon EC2 resources use Linux small instances with a CentOS distribution and the BenchLab client runtime controlling Firefox 3.6.13. The BenchLab Web application runs in Tomcat 6 on a laptop located on the UMass Amherst campus.

We have written a Java replay tool similar to httperf that can replay Faban traces through the Apache HttpClient 3 library. We have validated the tool by replaying traces generated by Faban and comparing the response time and server load with the ones obtained originally by Faban.

5.2. Realistic application data sets

In this experiment we illustrate the importance of having benchmark applications with realistic amounts of application state. The CloudStone benchmark populates the database and the filestore containing multimedia content according to the number of users to emulate. The state size grows proportionally to the number of users. Table 1 shows the dataset state size from 3.2GB for 25 users to 44GB for 500 users.

Table 1. CloudStone Web application server load observed for various dataset sizes using a workload trace of 25 users replayed with Apache HttpClient 3.

Dataset size	State size (in GB)	Database rows	Avg CPU load with 25 users
25 users	3.2	173745	8%
100 users	12	655344	10%
200 users	22	1151590	16%
400 users	38	1703262	41%
500 users	44	1891242	45%

We generated a load for 25 users using the Faban load generator and recorded all the interactions with their timestamps. We then replayed the trace using 25 emulated browsers and observed the resource usage on the CloudStone Web application (Olio) when different size data sets were used in the backend. The results in Table 1 show the CPU load observed in the Web Application VM. Note that in this experiment the trace is replayed through the Apache HttpClient 3 library and not using a real Web browser. The average CPU load on the server is 8% with the 25 user dataset but it reaches 45% for the exact same workload with a 500 user dataset. This is mainly due to less effective caching and less efficient database operations with larger tables.

Real applications like Wikipedia wikis have databases of various sizes with the largest being the English Wikipedia database which is now over 5.5TB. This experiment shows that even for a modest workload accessing the exact same working set of data, the impact on the server load can vary greatly with the dataset size. It is therefore important for realistic benchmarks to provide realistic datasets.

5.3. Real browsers vs emulators

5.3.1. Complexity of Web interactions

Real Web applications have complex interactions with the Web browser as shown in Table 2. While accessing the home page of older benchmarks such as RUBiS or TPC-W only generates 2 to 6 requests to fetch the page content. Their real life counterpart, eBay.com and amazon.com require 28 and 141 browser-server interactions, respectively. A more modern benchmark application such as CloudStone's Olio requires 28 requests which is still far from the 176 requests of the most popular social network Web site Facebook. When the user enters http://en.wikibooks.org/ in his favorite Web browser, 62 requests are generated on his behalf by the Web browser to fetch the content of the Wikibooks home page. Even if modern HTTP client libraries such as Apache HttpComponents Client [1] provide a good implementation of HTTP transport very similar to the one used in Web browsers, other functionalities such as caching, JavaScript execution, content type detection, request reformatting or redirection may not be accurately emulated.

Table 2. Browser generated requests per type when browsing the home page of benchmarks and Web sites.

Benchmark	HTML	CSS	JS	Multimedia	Total
RUBiS	1	0	0	1	2
eBay.com	1	3	3	31	28
TPC-W	1	0	0	5	6
amazon.com	6	13	33	91	141
CloudStone	1	2	4	21	28
facebook.com	6	13	22	135	176
wikibooks.org	1	19	23	35	78
wikipedia.org	1	5	10	20	36

To further understand how real browsers interact with real applications, we investigate how Firefox fetches a page of the Wikipedia Web site and compare it to an HTTP replay. The workflow of operations and the corresponding timings are shown in Figure 2. The times for each block of GET operations correspond to the network time measured by our HTTP replay tool (on the left) and Firefox (on the right). Times between blocks correspond to processing time in Firefox.

First we observe that the complexity of the application forces the browser to proceed in multiple phases. After sending the requested URL to the Web application, the browser receives an HTML page that it analyzes (step 1 on Figure 2) to find links to JavaScript code and additional content to render the page (.css, images…). Firefox opens six connections and performs the content download in parallel. It then starts to render the page and execute the JavaScript onLoad operations (step 2). This requires additional JavaScript files to be downloaded and another round of code execution (step 3).

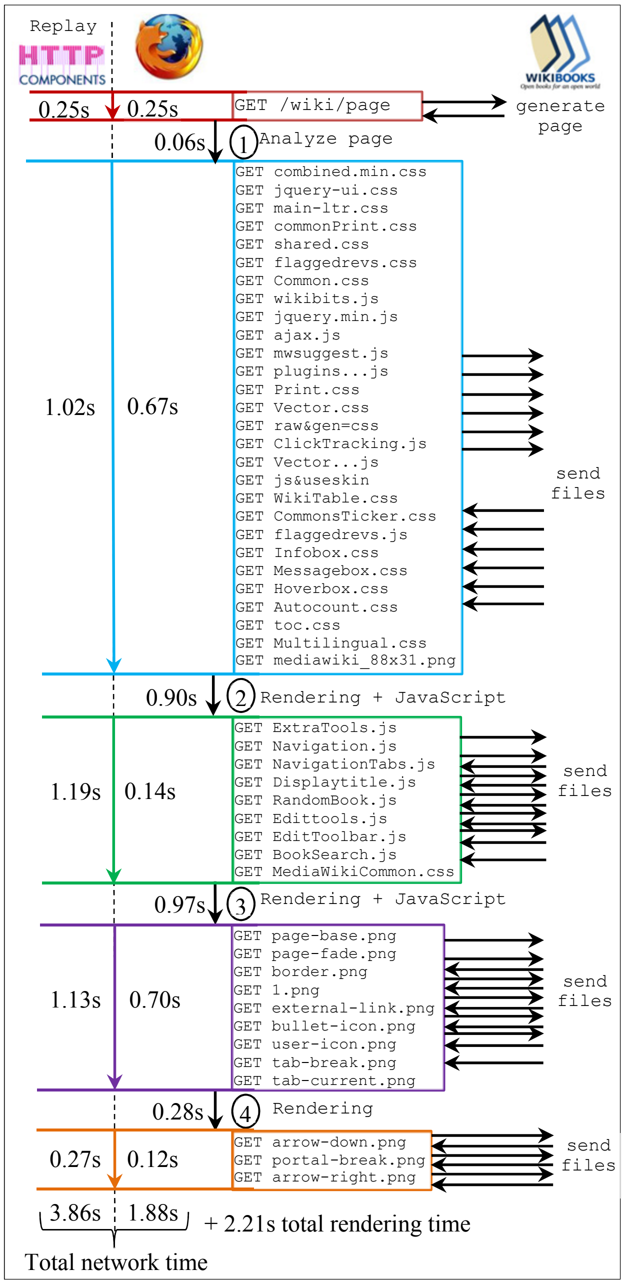


Figure 2. Time breakdown of a Wikibooks page access with Firefox 3.6.13 and HTTP replay.

Finally images are downloaded reusing the same six connections and a final rendering round (step 4) triggers the download of the 3 final images at the bottom of the page. The total page loading time in Firefox is 4.09s with 1.88s for networking and 2.21s for processing and rendering. The single threaded HTTP replay tool is not

able to match Firefox's optimized communications and does not emulate processing, thus generating different access patterns on the server leading to a different resource usage.
Finally, Table 3 shows how typing text in a Web page can result in additional requests to a Web application. When the user enters text in the search field of Wikibooks, a request is sent to the server on each keystroke to provide the user with a list of suggestions. The typing speed and the network latency influence how many requests are going to be made to the server.

Table 3. JavaScript generated requests when typing the word 'Web 2.0' in Wikibooks' search field.

```
GET /api.php?action=opensearch&search=W
GET /api.php?action=opensearch&search=Web
GET /api.php?action=opensearch&search=Web+
GET /api.php?action=opensearch&search=Web+2
GET /api.php?action=opensearch&search=Web+2.
GET /api.php?action=opensearch&search=Web+2.0
```

In Table 3's example, the user starts by typing 'W' causing a request to the server. She then quickly types the letter 'e' before the response has come back. When she types the next letter 'b' a second request goes to the server. Each following keystroke is followed by another request. This shows that even replaying in a Web browser needs to take in to consideration the speed at which a user performs operations since this can have an impact on how many requests are issued to the server directly affecting its load.

5.3.2. Latencies and server load

In this experiment, we inject the same 25 user CloudStone workload from the Amazon EC2 East coast data center to our server running at UMass Amherst. The emulator runs the 25 users from one virtual machine whereas 25 server instances each running one Firefox Web browser inject the load for the realistic injection. Figure 3 shows the latencies observed by the emulator and by the Web browsers.

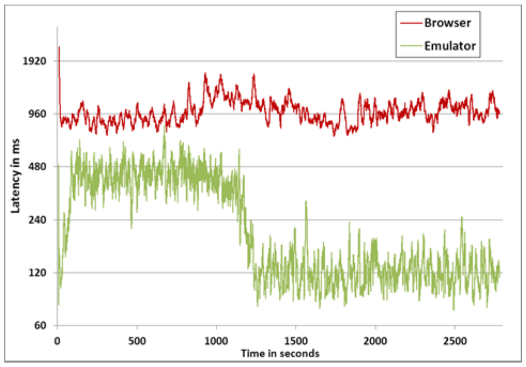


Figure 3. Browser vs Emulator measured latencies for the same load of 25 users using CloudStone between EC2 East coast and UMass Amherst.

The emulator has to mimic all the requests that a real Web browser would issue. Therefore a lot of small queries to fetch style sheets (.css) or JavaScript (.js) have small latencies. Some more complex pages are not fetched as efficiently on multiple connections and result in much higher latencies when the latencies of sub-requests are added up.
The latencies observed by the Web browsers vary significantly from the ones observed by the emulator because not only do they account for the data transfer time on the network, but also because they include the page rendering time that is part of the user perceived latency. Another key observation is that the showEvent interaction that displays information about an event in Olio's online calendar makes use of the Yahoo map API. As the emulator does not execute any JavaScript, all interactions with the real Yahoo Web site and its map API are not accounted in the interaction time. When a real browser is used, it contacts the Yahoo map Web site and displays the map with the location of the event. The page rendering time is then influenced not only by the response time of the Olio Web application but also with the interactions with Yahoo's Web site.

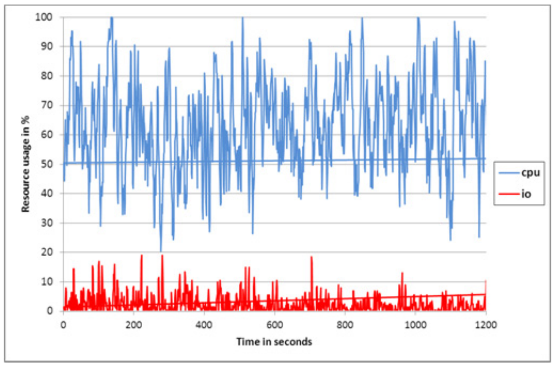


Figure 4. Server side CPU and disk IO usage with an emulator injecting a 25 user load from EC2 East coast to CloudStone at UMass Amherst.

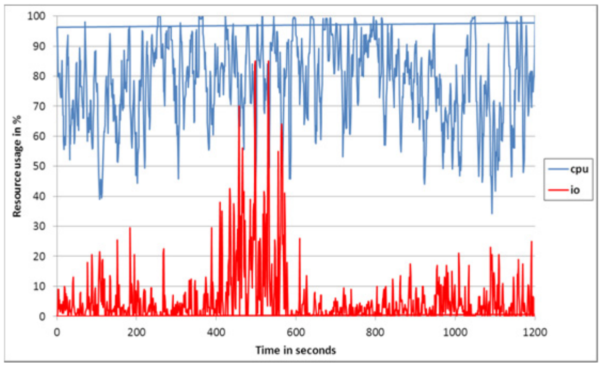


Figure 5. Server side CPU and disk io usage with 25 Firefox browsers from EC2 East coast to CloudStone at UMass Amherst.

Figure 4 and Figure 5 show the average CPU usage measured by vmstat every second on the Web application server for the emulated and realistic browser injection, respectively. While the user CPU time oscillates a lot in the emulated case it averages 63.2%. The user CPU time is more steady and constantly higher with the browser injected load at an average of 77.7%. Moreover, we notice peaks of disk IO during the experiment (for example at time 500 seconds), indicating that the IO subsystem of the server is more stressed when serving the browser generated requests.

5.3.3. Impact of JavaScript on Browser Replay

When a user fills a form, JavaScript code can check the content of the form and validate all the fields before being submitted to the application server. While the server has to spend more resources to send all the JavaScript code to the client, it does not have to treat any malformed requests with improper content that can be caught by the JavaScript validation code running in the Web browser.

In this experiment, we use the addPerson interaction of CloudStone that registers a new user with her profile information. The JavaScript code in that page generates a query to the server when the user name is typed in to check if that name is available or already taken. Other fields such as telephone number are checked for format and other fields are checked for missing information. Entering a zip code generates a query to the Geocoder service that returns the corresponding city and state names that are automatically filled in the corresponding fields of the form.

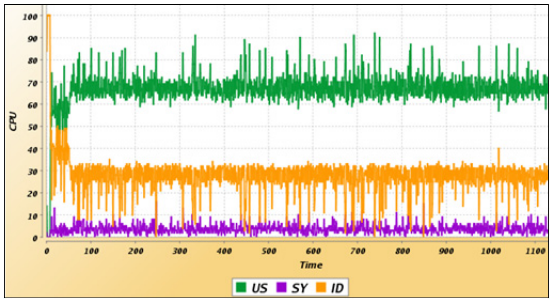


Figure 6. Server side CPU load (user, system, idle) with emulated load injection of 25 virtual users executing CloudStone's addPerson interaction with valid or bad inputs.

The original Olio workload driver generated malformed data that does not pass JavaScript checks but are accepted by the Olio application that does not check data sent from the client. We found this bug in the Olio application that inserts improper data in the database. We use two traces: one with the original malformed data and another one with valid inputs where we fixed the problems found in the original trace.

We emulate 25 clients from a server located in EC2 East coast's data center and run both traces. Figure 6 shows the load observed on the OlioVM when using emulated users. The CPU utilization is steady at around 70% for both traces. As the application does not check data validity and the emulator does not execute JavaScript, there is no change between good and bad inputs.

Figure 7 shows the CPU load observed on the server when the valid input trace is injected through Firefox. As forms are filled with data, additional queries are issued by JavaScript as mentioned earlier. This causes heavy weight write queries to be interlaced with lighter read queries. These additional context switches between the PHP scripts running the application and the Tomcat container running the Geocoder service cause significantly different resource usage in the Web application virtual machine.

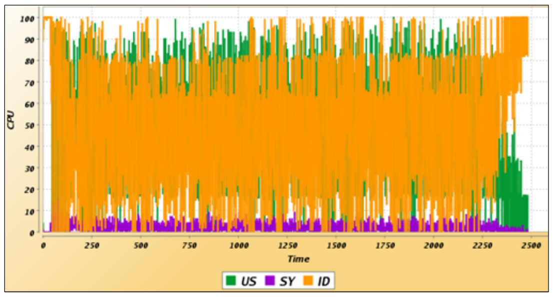


Figure 7. Server side CPU load (user, system, idle) using Firefox (25 browsers running from EC2 East coast) executing CloudStone's addPerson interaction with valid inputs.

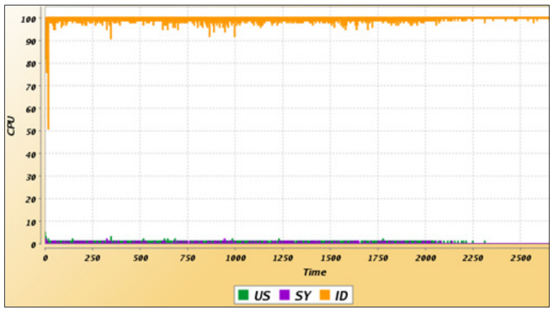


Figure 8. Server side CPU load (user, system, idle) using Firefox (25 browsers running from EC2 East coast) executing CloudStone's addPerson interaction with erroneous inputs.

Figure 8 shows the load seen by the Web application server when the trace with invalid inputs is injected through Firefox. As the JavaScript code checks for erroneous inputs, the requests never reach the application server. The only activity observed by the application is to answer the login uniqueness checks.

5.4. LAN vs WAN

In these experiments, we evaluate the impact of WAN based load injection vs traditional LAN based injection.

5.4.1. Local vs remote users

We observed the response time for a trace of 25 emulated users injected with our replay tool from a machine on the same LAN as the server and from a machine on Amazon EC2 East coast data center. As expected, the latencies are much higher on the WAN with 149ms average vs 44ms on the LAN. However, we observe that the latency standard deviation more than doubles for the WAN compared to the LAN.

The CPU usage for the WAN injection has already been presented in Figure 4 with an average CPU load of 54.8%. The CPU usage for the LAN experiment shows a highly varying CPU load but at a much lower 38.3% average. We attribute most of these differences to the increased connection times that require more processing to perform flow control on the connections, more context switches between longer lived sessions and more memory pressure to maintain more session states and descriptors simultaneously open. The exact root causes of these variations in server resource usage between LAN and WAN needs to be investigated further but we think that BenchLab is an ideal testbed for the research community to conduct such experiments.

5.4.2. Geographically dispersed load injection

We investigate the use of multiple data centers in the cloud to perform a geographically dispersed load injection. We re-use the same 25 user Cloudstone workload and distribute Web browsers in different Amazon data centers as follows: 7 US East coast (N. Virginia), 6 US West coast (California), 6 Europe (Ireland) and 6 Asia (Singapore). Such a setup (25 distributed instances) can be deployed for as little as $0.59/hour using Linux micro instances or $0.84/hour using Windows instances. More powerful small instances cost $2.30/hour for Linux and $3.00/hour for Windows. Figure 9 shows the latency reported by all Web browsers color coded per region (y axis is log scale).

Table 4. Average latency and standard deviation observed in different geographic regions

	US East	US West	Europe	Asia
Avg latency	920ms	1573ms	1720ms	3425ms
Std deviation	526	776	906	1670

As our Web application server is located in the US East coast, the lowest latencies are consistently measured by browsers physically located in the East coast data center. Average latency almost doubles for requests originating from the West coast or Europe. Finally, as expected, the Asian data center experiences the longest latencies. It is interesting to notice that the latency standard deviation also increases with the distance from the Web application server as summarized in Table 4. The server average CPU usage is 74.3% which is slightly less than when all browsers were located in the East coast (77.7%).

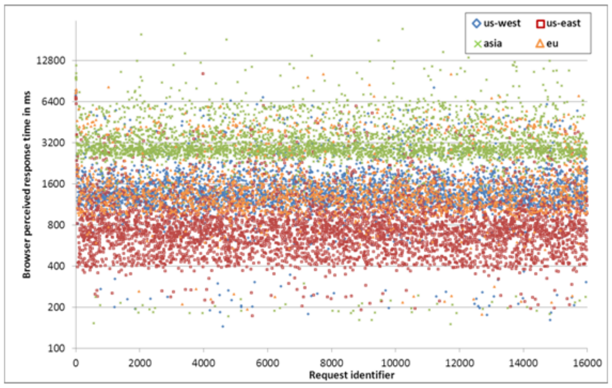


Figure 9. Browser (Firefox) perceived latency in ms (y axis is log scale) on a Cloudstone workload with users distributed as follows: 7 US East coast, 6 US West coast, 6 Europe and 6 Asia. Server location is UMass Amherst (US East coast, Massachusetts).

5.5. Summary

We have shown that real modern Web applications have complex interactions with Web browsers and that the state size of the application can greatly affect the application performance. Traditional trace replay tools cannot reproduce the rich interactions of browsers or match their optimized communication with Web application servers. Therefore the load observed on application servers varies greatly when the same workload is injected through an emulated browser or a real Web browser. This is further accentuated when JavaScript code generates additional queries or performs error checking that prevents erroneous inputs to reach the server.

Finally, we have shown the influence of LAN vs WAN load injection using real Web browsers deployed in a public cloud. Not only the latency and its standard deviation increase with the distance but the load on the server significantly differs between a LAN and a WAN experiment using the exact same workload.

6. Related Work

Benchmarks such as RUBiS [13] and TPC-W [16] have now become obsolete. BenchLab uses CloudStone [15] and Wikibooks [20] as realistic Web 2.0 application backends. The Rain [2] workload generation toolkit separates the workload generation from execution to be able to leverage existing tools such as httperf. BenchLab uses the same concept to be able to replay real workload traces from real applications such as Wikibooks or Wikipedia [17] in Web browsers.

Browser automation frameworks have been developed primarily for functional testing. BenchLab uses real Web browsers for Web application benchmarking. Commercial technologies like HP TruClient [6] or Keynote Web performance testing tools [7] offer load injection from modified versions of Firefox or Internet Explorer. BrowserMob [4] provides a similar service using Firefox and Selenium. However, these proprietary products can only be used in private clouds or dedicated test environments. BenchLab is fully open and can be deployed on any device that has a Web browser. By deploying browsers on home desktops or cell phones, BenchLab can be used to analyze last mile latencies.

Server-side monitors and log analysis tools have been used previously to try to model the dependencies between file accesses and predict the full page load times observed by clients [10][18]. BenchLab's use of real Web browsers allows it to accurately capture the behavior of loading Web pages composed of multiple files, and could be used by Web service providers to monitor the performance of their applications. When deployed on a large number of machines (home desktops, cell phones, cloud data centers…), BenchLab can be used to reproduce large scale flash crowds. When client browsers are geographically dispersed, BenchLab can be used to evaluate Content Delivery Network (CDN) performance or failover mechanisms for highly available Web applications.

BenchLab is designed to be easily deployed across wide geographic areas by utilizing public clouds. The impact of wide area latency on Web application performance has been studied in a number of scenarios [5]; BenchLab provides a standardized architecture to allow application developers and researchers to measure how their systems perform in real WAN environments. We believe that BenchLab can be used to measure the effectiveness of WAN accelerators (CDNs or proxy caches) as well as validate distributions modeling WAN load patterns.

7. Conclusion

We have demonstrated the need to capture the behavior of real Web browsers to benchmark real Web 2.0 applications. We have presented BenchLab, an open testbed for realistic benchmarking of modern Web applications using real Web browsers. BenchLab employs a modular architecture that is designed to support different backend server applications. Our evaluation has illustrated the need for 1) updated Web applications with realistically sized datasets to use as benchmarks, 2) real browser based load injection tools that authentically reproduce user interactions and 3) wide area benchmark client deployments to match the network behavior seen by real applications. We believe that BenchLab meets these needs, and we hope that it will help the research community improve the realism and accuracy of Web application benchmarking. We are making all the BenchLab software (runtime, tools, Web applications…) available to the community under an open source license on our Web site [3].

8. Acknowledgement

The authors would like to thank the anonymous reviewers and our shepherd Geoffrey Voelker for their valuable feeback. We would also like to thank Fabien Mottet and Vivien Quema from INRIA Rhône-Alpes, Guillaume Pierre from Vrije University and Vimal Mathew from UMass, for their contributions to BenchLab. This research was supported in part by grants from Amazon AWS, UMass President's Science & Technology fund, and NSF grants CNS-0916972, CNS-083243, CNS-0720616 and CNS-0855128.

9. References

[1] Apache HttpComponents – http://hc.apache.org/

[2] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox and D. Patterson – *Rain: A Workload Generation Toolkit for Cloud Computing Applications* – Technical Report UCB/EECS-2010-14, February 10, 2010.

[3] BenchLab - http://lass.cs.umass.edu/projects/benchlab/

[4] BrowserMob - http://browsermob.com/performance-testing

[5] S. Chen, K.R. Joshi, M.A. Hiltunen, W.H. Sanders and R.D. Schlichting – *Link Gradients: Predicting the Impact of Network Latency on Multitier Applications* – INFOCOM 2009, pp.2258-2266, 19-25 April 2009.

[6] HP - TruClient technology: Accelerating the path to testing modern applications – Business white paper, 4AA3-0172ENW, November 2010.

[7] R. Hughes and K. Vodicka – Why Real Browsers Matter – Keynote white paper, http://www.keynote.com/docs/whitepapers/why_real_browers_matter.pdf.

[8] D. Krishnamurthy, J. A. Rolia and Shikharesh Majumdar – *A Synthetic Workload Generation Technique for Stress Testing Session-Based Systems* – IEEE Transaction on Software Engineering. 32, 11 - November 2006.

[9] HTTP Archive specification (HAR) v1.2 - http://www.softwareishard.com/blog/har-12-spec/.

[10] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A.G. Greenberg, and Y. Wang - *WebProphet: Automating Performance Prediction for Web Services* – NSDI, 2010, pp.143-158.

[11] E. M. Nahum, M.C. Rosu, S. Seshan and J. Almeida – *The effects of wide-area conditions on WWW server performance* – SIGMETRICS 2001.

[12] Olio – http://incubator.apache.org/olio/

[13] RUBiS Web site – http://rubis.ow2.org.

[14] Selenium - http://seleniumhq.org/

[15] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox and D. Patterson – *Cloudstone: Multi-platform, multi-language benchmark and measurement tools for Web 2.0* – Cloud Computing and its Applications CCA-08, 2008.

[16] TPC-W Benchmark, ObjectWeb implementation, http://jmob.objectWeb.org/tpcw.html

[17] G. Urdaneta, G. Pierre and M. van Steen – *Wikipedia Workload Analysis for Decentralized Hosting* – Elsevier Computer Networks, vol.53, July 2009.

[18] J. Wei, C.Z. Xu - *Measuring Client-Perceived Pageview Response Time of Internet Services* – IEEE Transactions on Parallel and Distributed Systems, 2010.

[19] WikiBench - http://www.wikibench.eu/

[20] Wikibooks – http://www.wikibooks.org

[21] Wikipedia – http://www.wikipedia.org

Resource Provisioning of Web Applications in Heterogeneous Clouds

Jiang Dejun
VU University Amsterdam
Tsinghua University Beijing

Guillaume Pierre
VU University Amsterdam

Chi-Hung Chi
Tsinghua University Beijing

Abstract

Cloud computing platforms provide very little guarantees regarding the performance of seemingly identical virtual machine instances. Such instances have been shown to exhibit significantly different performance from each other. This heterogeneity creates two challenges when hosting multi-tier Web applications in the Cloud. First, different machine instances have different processing capacity so balancing equal amounts of load to different instances leads to poor performance. Second, when an application must be reprovisioned, depending on the performance characteristics of the new machine instance it may be more beneficial to add the instance to one tier or another. This paper shows how we can efficiently benchmark the individual performance profile of each individual virtual machine instance when we obtain it from the Cloud. These performance profiles allow us to balance the request load more efficiently than standard load balancers, leading to better performance at lower costs. The performance profiles also allow us to predict the performance that the overall application would have if the new machine instance would be added to any of the application tiers, and therefore to decide how to make best use of newly acquired machine instances. We demonstrate the effectiveness of our techniques by provisioning the TPC-W e-commerce benchmark in the Amazon EC2 platform.

1 Introduction

Cloud computing is an attractive platform to host Web applications. Besides the advantages of outsourcing machine ownership and system management, Clouds offer the possibility to dynamically provision resources according to an application's workload — and to pay only for the resources that are actually being used. Given a service-level objective (SLO) which states the response time guarantees an application provider wants to maintain, dynamic resource provisioning continuously adjusts the number of computing resources used to host the application. Additional capacity can be added dynamically when the load of user requests increases, and later released when the extra power is no longer necessary.

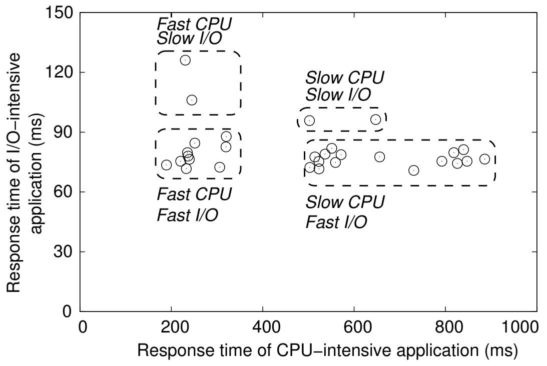
Resource provisioning for Web applications in the Cloud faces two important challenges. First, Web applications are not monolithic. At the very least a Web application is composed of application servers and database servers, which both can benefit from dynamic resource provisioning. However, the effect of reprovisioning an extra machine varies from tier to tier. When adding or removing capacity, one needs to decide which tier must be (de-)provisioned such that the performance remains within the SLO at the lowest cost. Second, computing resources in the Cloud are not homogeneous. This is obvious when considering the many different instance types in a platform such as Amazon's EC2 [1]. However, even an application which would decide to use a single instance type would not experience homogeneous performance. Figure 1(a) illustrates the performance of 30 'identical' EC2 instances when running the same application server or database server workloads [2]. Clearly, some instances are more suitable than others for efficiently running CPU-intensive application servers, but are less suitable for I/O intensive workloads. Other instances have faster I/O but slower CPU, and may be better used as database servers. Finally, we have fast machines which can be used for either of both, and slow machines which should either be given a modest task such as load balancing or de-provisioned altogether. On the other hand, Figure 1(b) shows the response time of individual instance running application server workload on EC2 over a period of 24-hours, measured at a 1-minute granularity [2]. Performance spikes occasionally occur with an average duration of 1 to 3 minutes, but overall the performance of individual instances is consistent over time. The performance spikes of an individual instance are presumably caused by the launch/shutdown operations of the other virtual instances on the same physical machine. The same experiments run in the Rackspace Cloud show similar behavior. Similar observations are also reported for different application fields and performance metrics [8].

Efficient dynamic resource provisioning in these conditions is very difficult. To provision a Web application dynamically and efficiently, we need to predict the performance the application would have if it was given a new machine, such that one can choose the minimum number of machines required to maintain the SLO. However, because of resource heterogeneity it is impossible to predict the performance profile of a new machine instance at the time we request it from the Cloud. We therefore cannot accurately predict the performance the application would have if it was using this new machine in one of its tiers. It is therefore necessary to *profile* the performance of the new machine before deciding how we can make the best use of it.

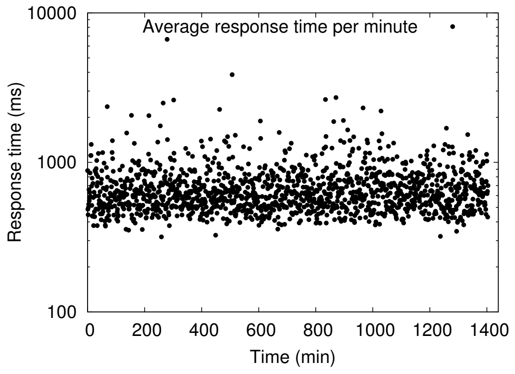
One simple profiling method would consist of sequentially adding the new machine instance to each tier of the application and measure the performance gain it can provide on each tier. However, this approach is extremely inefficient and time-consuming as profiling requires lots of time. For instance, adding a machine instance to a database tier may cost tens of minutes or more, which is not acceptable for dynamic resource provisioning.

In this paper we show how one can efficiently profile new machines using real application workloads to achieve accurate performance prediction in the heterogeneous Cloud. By studying the correlation of demands that different tiers put on the same machine, we can derive the performance that a given tier would have on a new machine instance, without needing to actually run this tier on this machine instance. This per-tier, per-instance performance prediction is crucial to take two important decisions. First, it allows us to balance the request load between multiple heterogeneous instances running the same tier so as to use each machine instance according to its capabilities. Second, when the application needs to expand its capacity it allows us to correctly select which tier of the application should be reprovisioned with a newly obtained instance.

We evaluate our provisioning algorithm in the Amazon EC2 platform. We first demonstrate the importance of adaptive load balancing in Cloud to achieve homogeneous performance from heterogeneous instances. We then use our performance prediction algorithm to drive the dynamic resource provisioning of the TPC-W e-commerce benchmark. We show that our system effectively provisions TPC-W in the heterogeneous Cloud and achieve higher throughput compared with current provision techniques.



(a) EC2 Cloud performance heterogeneity



(b) Consistent performance of individual instance over time

Figure 1: Heterogeneous Cloud performance

The rest of this paper is organized as follows: Section 2 introduces research efforts related to our work. Section 3 shows an example of scaling application on EC2 to highlight the motivation of our work. Section 4 presents the design of our resource provisioning system. Section 5 evaluates our system using both single-tier and multi-tier Web applications. Finally, Section 6 concludes.

2 Related work

A number of research efforts address dynamic resource provisioning of Web applications and model the interactions between tiers of a multi-tier Web application through analytical queueing models [9, 12, 13]. These algorithms can drive the decision to reprovision one tier rather than another for best performance. They also incorporate the effect of provisioning techniques such as caching and master-slave database replication into their provisioning models. We previously extended these works for the dynamic resource provisioning of multi-service Web applications, where Web applications are not only constructed as a sequence of tiers but can also consist of multiple services interacting with each other in a directed acyclic graph [3]. These works however assume that the underlying provisioning machines are homogeneous. This is a reasonable assumption in medium-scale environments such as cluster computers. However, in Cloud computing platforms resources are heterogeneous so these systems do not apply.

A few research works address the problem of provisioning Web applications in heterogeneous resource environments. Stewart et al. predict the performance of Internet Services across various server platforms with different hardware capacities such as processor speeds and processor cache sizes [10]. Similarly, Marin et al. use detailed hardware models to predict the performance of scientific applications across heterogeneous architectures [6]. These approaches rely on detailed hardware metrics to parametrize the performance model. However, in the Cloud such low-level metrics are hidden by the virtualization layer. In addition, Cloud hardware resources are typically shared by virtual instances, which makes it much harder for hardware-based performance models to capture the performance features of consolidated virtual instances. These works therefore cannot be easily extended to predict Web application performance in the Cloud.

JustRunIt is a sandbox environment for profiling new machines in heterogeneous environments using real workloads and real system states [14]. When one needs to decide on the usage of new machines, this work clones an online system to new machines, and duplicate online workload to them. This approach can effectively capture performance characteristics of new virtual instances in the Cloud. However, it requires to clone online environment to new instances at each adaptation, which can be very time-consuming. On the other hand, our work can predict the performance of new instances for running Web applications without actually executing the application on new instances.

Elnikety et al. address the problem of predicting replicated database performance using standalone database profiling [4]. This work inspired us to realize performance prediction through online profiling techniques. However, it addresses a different problem than ours: here the problem is to predict the performance of different database replication techniques rather than accounting for the performance heterogeneity of the database servers themselves. Our work focuses on the latter.

Finally, instead of predicting performance, Kalyvianaki et al. use control theory to allocate CPU resources to multi-tier Web applications hosting across various virtual instances in a virtualized data center [5]. This work mainly focuses on the problem of hardware resource assignment for composing different virtual instances with different capabilities. It does not address performance impact of resource heterogeneity caused by the virtualization. Nathuji et al. also focus on providing Quality-of-Service guarantees to applications running in the Cloud [7]. However, this work aims at dynamically adapting the hardware resource allocation among consolidated virtual instances to mitigate performance interference of different applications, rather than making the best possible use of heterogeneous machine instances.

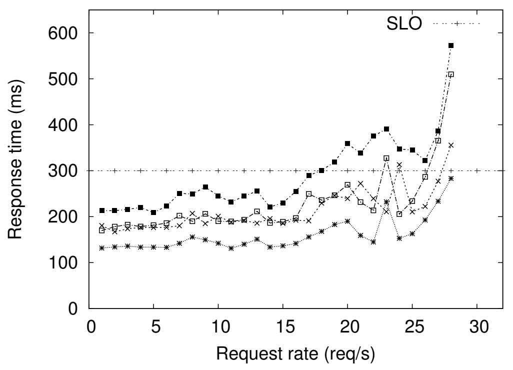


Figure 2: Web application performance heterogeneity under Auto-Scale on EC2

3 Motivating example

The performance heterogeneity of Cloud resources depicted in Figure 1 has strong consequences on Web application hosting. Figure 2 shows the response time of a single-tier CPU-intensive Web application deployed on four 'identical' virtual instances in the Amazon EC2 Cloud, using Amazon's own Elastic Load Balancer (which addresses equal numbers of requests to all instances). As we can see, the four instances exhibit significantly different performance. The response time of the first instance exceeds 300 ms around 17 req/s while the fastest instances can sustain up to 28 req/s before violating the same SLO. As a result, it becomes necessary to re-provision the application at 17 req/s, while the same virtual instances could sustain a much higher workload if they were load balanced according to their individual capabilities.

As we shall see in the next sections, the problem becomes more difficult when considering multi-tier Web applications. When re-provisioning a multi-tier Web application, one must decide which tier a new virtual instance should be added to, such that the overall application performance is maximized. However, this choice largely depends on the individual performance of the new virtual instance: an instance with fast I/O is more likely than another to be useful as a database replica, while an instance with fast CPU may be better used as an extra application server.

4 Dynamic resource provisioning

Dynamic resource provisioning for web applications in the Cloud requires one to predict the performance that heterogeneous machine instances would have when executing a variety of tiers which all have different demands for the machine instances. This performance prediction allows us to choose which tier(s) should ideally benefit from this instance for optimal performance gains of this entire application.

4.1 Solution outline

We address the problem in four steps. First, when using multiple heterogeneous machines to run a single tier, one must carefully balance the load between them to use each machine according to its capacity such that each provisioned instance features with equal response time. We discuss Web application hosting techniques and load balancing in section 4.2.

Second, we need to measure the individual performance profile of new machine instances for running specific application tiers. Benchmarking a machine instance for one tier does not generate a single measurement value, but an estimation of the response time as a function of the request rate. Profiling a tier in a machine requires some care when the tier is already used in production: we need to measure the response time of the tier under a number of specific request rates, but at the same time we must be careful so that the instance does not violate the application's SLO. We discuss profiling techniques in section 4.3.

Third, every time the application needs to provision a new machine instance, it is very inefficient to successively profile each of the application tiers on the new instance. Instead, we calibrate the respective hardware demands of different tiers of the Web application using a single 'calibration' machine instance. We also include two synthetic reference applications in the calibration. After this step, each new instance is benchmarked using the reference applications only. Thanks to the initial calibration, we can predict the performance that this particular machine instance would have if it was executing any tier of the real Web application. We discuss performance prediction in section 4.4.

Finally, knowing the performance that a new machine instance would have if we added it to any tier of the application, we can choose the tier where it would generate the greatest performance improvement for the overall application. We choose the targeted tier by modeling the whole Web application as a queueing network where each tier acts as a separate queue. We discuss the queueing model for provisioning tiers in section 4.5.

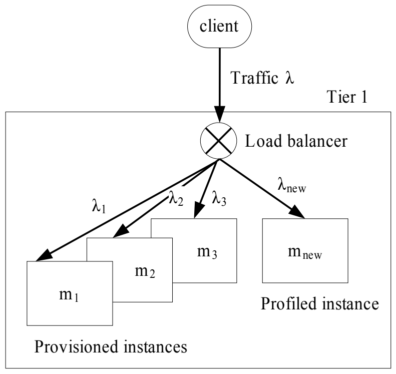


Figure 3: Web application hosting in the Cloud

4.2 Web application hosting

Figure 3 shows the typical hosting architecture of a single tier of the Web application. The provisioned virtual instances m_1, m_2 and m_3 host either the replicated application code if this is an application server tier, or a database replica if this is a database tier. As the performance of provisioned instances largely differs from each other, it would be a bad idea to address the same request rate to all instances. A much better option is to carefully control the respective load of each instance such that they all exhibit the same response time. In this scenario, fast machine instances get to process more requests than slower ones.

We control the workload by deploying a custom load balancer in front of the provisioned instances. To guarantee backend instances to serve with equal response time, the load balancer calculates the weighted workload distribution according to their performance profiles by solving the following set of equations:

$$\begin{cases} \lambda = \lambda_1 + \cdots + \lambda_n \\ r = \mathit{perf}(\mathit{instance}_1, \lambda_1) \\ \ldots \\ r = \mathit{perf}(\mathit{instance}_n, \lambda_n) \end{cases} \tag{1}$$

where λ is the total request rate seen by the load balancer, $\lambda_1, \ldots, \lambda_n$ are the request rates addressed to each provisioned instance respectively and r is the uniform response time of all provisioned instances. The $\mathit{perf}()$ functions are typically defined as a set of measured points, with linear interpolation between each consecutive pair of points.

This set of *n+1* equations can be easily solved to find the values of *r*, λ_1, ..., λ_n. The load balancer uses these values as weights of its weighted Round-Robin strategy.

When adding a new instance m_{new} into this tier, the load balancer thus needs to know the performance profile of this new instance such that it can balance the workload accordingly. This is the goal of instance profiling that we discuss in next.

4.3 Online profiling

Coming up with a machine instance's own performance profile when provisioning a given tier can be done in two different ways: either we measure the actual profile using real request traffic, or we derive the profile from other measured profiles. This section discusses the former.

Profiling a machine instance with a given tier workload consists in deploying the tier service on the machine instance, then addressing traffic with various load intensities to measure the corresponding response times.

We approximate the actual profile of a new instance by measuring performance at carefully selected workload intensities, and using linear interpolation between each consecutive pair of measured points. The output of the online profiling of a new instance is therefore a set of *n* linear functions which cover consecutive workload ranges as follows:

$$r_i = a_i \times \lambda_i + b_i \qquad (1 \leq i \leq n) \tag{2}$$

where *n* is the total number of the consecutive workload ranges, and *r*, λ, *a* and *b* respectively represent average response time, request rate and linear parameters within the workload range *i*.

Generating a performance profile for a synthetic application is relatively easy: one only needs to deploy the synthetic application on the tested machine, and use a separate machine instance to generate a standardized workload and measure the tested instance's performance profile.

Generating a similar performance profile for a real tier of the Web application is harder. We want to address traffic that is as realistic as possible to increase the accuracy of the performance profile. Metrics which make a traffic workload realistic include the respective proportions of simple and complex requests, read/write ratio, popularity distribution of different data items, and so on. All these properties have a significant impact on performance. Instead of trying to synthesize a realistic workload, we prefer to provision the new instance in the tier to profile, and address real live traffic to it (which is realistic by definition).

Profiling a machine instance using live traffic however requires caution. First, we must make sure that profiling this instance will not create an SLO violation for the end users whose requests are processed by the profiled machine instance. For instance, one could simply replace one of the current instances used in the tier with the instance to profile. However, if the new instance is slower than the previous one, the application may violate its SLO. Second, we want to test specific workload intensities regardless of the actual workload received by the tier at the time of the profiling. Profiling a live tier therefore requires careful load balancing where we control the request rate addressed to the profiled instance.

We first need a rough estimation of the variety of performance profiles from one instance to another. Such variety is specific to one Cloud provider, as it largely depends on the consolidation strategies and virtualized performance isolation that the Cloud implements. We calculate the performance variety rate *N* as follows.

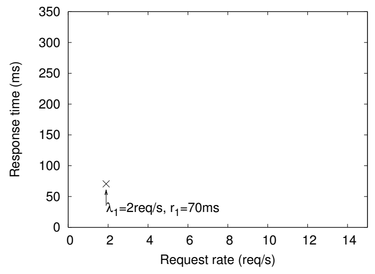
$$N = \frac{T_{max}}{T_{min}} \tag{3}$$

where T_{max} and T_{min} respectively represent the throughput of the fastest and slowest instances in the Cloud when running a given Web application. We set a SLO defining the maximum response time to this Web application. We measure the throughput of this Web application when it violates the SLO. The tested application exhibits either CPU-intensive or I/O-intensive workload for estimating the CPU and IO performance variety separately. For instance, in Amazon EC2 Cloud we observed $N_{CPU} \approx 4$ for CPU-intensive tiers such as application servers and $N_{I/O} \approx 2$ for I/O-intensive tiers such as database servers [2]. Similarly, in Rackspace we observed $N_{CPU} \approx 1.5$ and $N_{I/O} \approx 4$. In a new Cloud platform, one would need to sample a sufficient number of instances to evaluate these numbers.

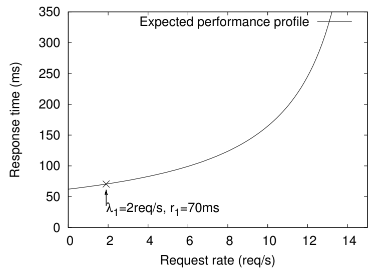
Second, we carefully choose different workload intensities to address to the new machine. One needs to choose the key performance points (λ, r) that represent significant features of the performance profile. For instance, the performance profile of a new instance under low workload can be approximated as a constant value regardless of the load. We ideally want a number of points varying from underload to overload situations, preferably at the inflection points and close to the SLO. The accuracy of the approximated curve increases with the number of measured points. However, this also increases the profiling time.

Figure 4 illustrates our strategy to select the request rate for profiling the new instance. We first address the new instance with request rate $\lambda_1 = \frac{\lambda_{max}}{N}$, where λ_{max} is the current request rate of the fastest instance currently used in this tier. Assuming our estimation of performance variety N is correct, the profiled instance cannot violate the SLO even if it happens to be very slow. This gives us the first performance point (λ_1, r_1) as illustrated in Figure 4(a)

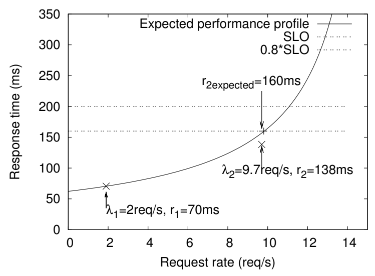
Using this first measurement, we can use queueing theory to generate a first estimate of the entire performance profile of this instance. If we model the tier as an M/M/n queue, then the instance's service time can be computed as:



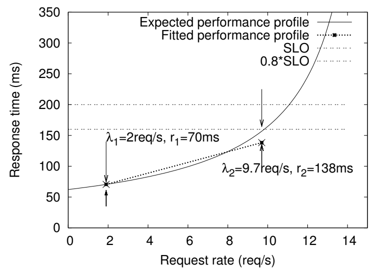
(a) First measurement point selected such that the instance will not violate its SLO



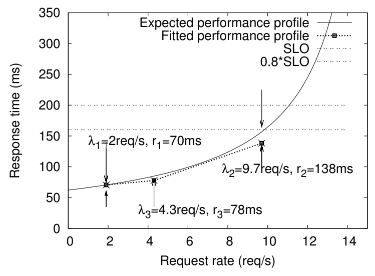
(b) First estimation of the instance's profile thanks to queueing theory



(c) Selection of a second measurement point close to the SLO



(d) Fit performance profile of the new instance



(e) Correction of the performance profile of the new instance

Figure 4: Online profiling process

$$s = \frac{r_1}{1 + \frac{\lambda_1 \times r_1}{n}} \tag{4}$$

where n is the number of CPU cores of this machine (as announced by the Cloud provider). The service time is the response time of the instance under low workload where the effects of request concurrency are negligible. It indicates the capability of the instance to serve incoming requests. We can then use this service time to derive a first performance profile of the new instance as follows:

$$r(\lambda) = \frac{s}{1 - \frac{\lambda \times s}{n}} \tag{5}$$

Figure 4(b) shows the first performance profile of the new instance derived based on the calculated service time. One should however note that this profile built out of a single performance value is very approximate. For a more precise profile, one needs to measure more data points.

Using this profile, we can now calculate a second workload intensity which should bring the instance close to the SLO. We select an expected response time r_2, then derive the workload intensity which should produce this response time.

$$r_2^{expected} = 0.8 \times SLO \tag{6}$$

$$\lambda_2 = \frac{n \times (r_2^{expected} - s)}{r_2^{expected} \times s} \tag{7}$$

Here we set the target response time to 80% of the SLO to avoid violating the SLO of the profiled instance even though the initial performance profile will feature relatively large error margins. We can then address this workload intensity to the new instance and measure its real performance value (λ_2, r_2). As shown in Figure 4(c), the real performance of the second point is somewhat different from the expected 80% of the SLO.

We apply linear regression between the two measured points (λ_1, r_1), (λ_2, r_2) and get the fitted performance profile of the new instance as shown in Figure 4(d). We then let the load balancer calculate the weighted workload distribution between the provisioned instance and the new one.

By addressing the weighted workload intensities to the two instances, we can measure the real response time of the new instance. However, as shown in Figure 4(e), the real performance of the new instance differs slightly from the expected one in Figure 4(d) due to the approximation error of its initial profile. We then correct the performance profile of the new instance by interpolating the third performance point(λ_3, r_3). We show that the above strategy is effective to profile heterogeneous virtual machines and provision single services in Section 5.

Although expressing performance profiles as a function of request rate is useful for load balancing, for performance prediction we need to express performance profiles as a function of CPU utilization (for application server tiers) or I/O utilization (for database server tiers). When profiling a machine instance, we also measure the relevant metrics of resource utilization, and use the exact same technique to build performance profiles that are suitable for performance prediction.

4.4 Performance prediction

To efficiently select the tier in which a new instance will be most valuable to the application as a whole, we first need to know the performance profile of this instance when running each of the application's tiers. A naive approach would be to successively measure this profile with each tier one by one before taking a decision. However, this strategy would be too slow to be of any practical use. Indeed, profiling a new machine with a real application tier requires to first replicate the hosted service to the new machine. For instance, when profiling a new machine for a database tier, one needs to first replicate the entire database to the new machine before starting the profiling process. Replicating a medium-sized database can easily take tens of minutes, and this duration increases linearly with the database size. We therefore need to be able to quickly *predict* the performance profiles, without needing to actually replicate the database.

We found that the most characteristic feature of a virtual instance to predict the performance profile of a given tier in this instance is its resource utilization. Although the absolute response time of two different tiers in the same machine under the same CPU or I/O utilization are not identical, they are highly correlated.

We illustrate this in Figure 5. Each point in this graph represents the response times of two different application server tiers running in the same machine instance, and having the same CPU utilization (respectively 15%, 25%, 65% and 80%). The request rates necessary to reach a given CPU utilization varies from one application to the next. We however observe that the points form an almost perfect straight line. This allows us to derive new performance profiles from already known ones. The same observation is also true for database server tiers, taking the disk I/O bandwidth consumption as the resource utilization metric.

Given the response time and resource utilization of one tier in a given machine instance, we can infer the response time of the second tier in the same machine instance under the same resource utilization. Figure 6 illustrates the input and output of this prediction: we predict the performance of tier 1 and tier 2 on a new machine by

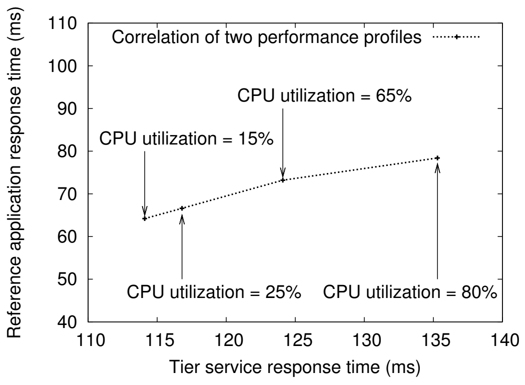


Figure 5: Performance correlation between reference application and tier service

Input:
$perf(machine_{calibration},\ app_{ref}) = f(load)$
$perf(machine_{calibration},\ app_{tier1}) = f(load)$
$perf(machine_{calibration},\ app_{tier2}) = f(load)$
$perf(machine_{new},\ app_{ref}) = f(load)$

Output:
$perf(machine_{new},\ app_{tier1}) = f(load)$
$perf(machine_{new},\ app_{tier2}) = f(load)$

Figure 6: Input and output of the performance profile prediction

correlating their performance and the reference application performance on a calibration machine.

First, we need to measure the application-specific demands of each tier of the application. This has to be done only once per application. This profiling should be done on a single calibration machine, which can be any particular virtual instance in the Cloud. To predict the performance of any particular tier on a new instance quickly, we also benchmark the calibration machine using two synthetic reference applications which respectively exhibit CPU-intensive features characteristic of application servers, and I/O-intensive features characteristic of database servers. The Ref_{CPU} application receives customer names and generates detailed personal information through CPU-intensive XML transformation. The $Ref_{I/O}$ application searches for items related to a customer's previously ordered items from a large set of items. The operations of the reference applications introduce typical CPU-intensive and disk I/O-intensive workloads. The reference applications can be deployed very quickly on any new machine instance, for example by including it to the operating system image loaded by the virtual machine instances. We use Ref_{CPU} as a reference point to predict the performance profiles of application server tiers, and $Ref_{I/O}$ as a reference point to predict the performance profiles of database tiers.

Profiling the same calibration machine instance with one of the Web application's tiers and the corresponding reference application allows us to learn the relationship between the demands that these two applications put on the hardware:

$$perf(app_{tier}, utilization) = \alpha \times perf(app_{ref}, utilization) + \beta$$

The same relationship between the response times of the two applications, captured by the values of α and β, remains true on other machine instances. Knowing the performance profile of the reference application on a newly obtained virtual machine instance from the cloud, we can thus derive the predicted performance profile of *tier1* on the new instance, even though we never even installed this particular tier on this particular instance.

4.5 Resource provisioning

When provisioning a multi-tier Web application, upon a violation of the service-level objective, one needs to decide which tier to re-provision within the whole application. Once a new instance is acquired and profiled, one needs to perform a simple what-if analysis to predict the performance improvement that the whole application would observe if this instance was added in one of the tiers. For simplicity, in this paper we apply our Cloud instance profiling methods to simple two-tier Web applications only. Other performance models of composite Web applications can be used to extend this work to more complex setups [3, 13].

The response time of a two-tier Web application can be computed as follows.

$$R_{app} = R_1 + N_{1,2} \times R_2 \quad (8)$$

where R_{app} is the response time of the whole application, R_1, R_2 are response time of the application server tier and the database tier respectively, $N_{1,2}$ is the request ratio that equals to the request number seen by the second (database) tier caused by one request from the first (application server) tier.

Given the performance profiles of the new instance for each of the application's tiers, we can issue a simple "what-if" analysis: we first use the performance profiles to compute the new application performance if the new instance was added to the first tier, then if it was added to the second tier. The best usage of the new instance is defined as the one which maximizes the application's performance.

5 Experimental evaluation

In this section we evaluate the effectiveness and efficiency of our resource provisioning algorithm for provisioning Web applications on the Amazon EC2 platform.

5.1 Experiment setup

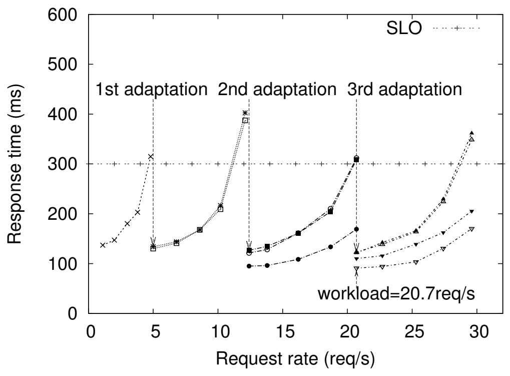
The bulk of our system implementation lies in our custom layer-4 load balancer. In addition to distributing requests to backend servers, the load balancer also profiles new machines when we obtain them from the cloud. By deploying the load balancer in front of the tiers of Web applications, our system can provision Web applications over heterogeneous instances in the Cloud.

We evaluate our resource provisioning algorithm using three Web applications. The first two are the reference applications Ref_{CPU} and $Ref_{I/O}$. The last one is the TPC-W Web application benchmark. This benchmark is structured as a two-tiered application which models an online bookshop like Amazon.com [11]. We run all our experiments in Amazon EC2 platform using small instances.

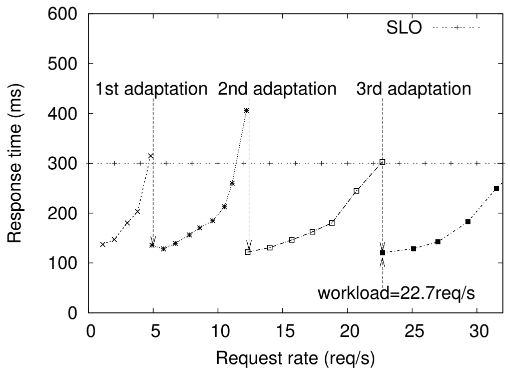
5.2 Importance of adaptive load balancing

We first demonstrate the importance of adaptive load balancing in Cloud using Ref_{CPU} and $Ref_{I/O}$. We deploy each application on a single machine instance, and increase the workload gradually. We set the SLO of the response time of Ref_{CPU} and $Ref_{I/O}$ to 300 ms and 500 ms respectively. We run each experiment using two different setups. First, we use Amazon's Elastic Load Balancer to distribute the traffic between the instances, and Amazon's AutoScale to provision new virtual machine instances when the SLO is violated. Second, we run the same experiment using the exact same instances with our system. Both applications are single-tiered, so here we exercise only the capability of load balancing to adapt to heterogeneous resources.

Figure 7 shows the response time per machine instance running the Ref_{CPU} application. When using the Elastic Load Balancer (ELB), at 5 req/s the system violates the SLO and therefore provisions a new instance. By coincidence, the second instance has a performance profile very close to the first one so they exhibit extremely similar performance. However, after the second and third adaptation we see that different instances exhibit different performance. On the other hand, our system balances the workload such that all instances always exhibit the same performance. This has important consequences in terms of resource usage: when using ELB, the one of the application instances violates its SLO at 20.7 req/s, triggering a request for a fourth instance. When using

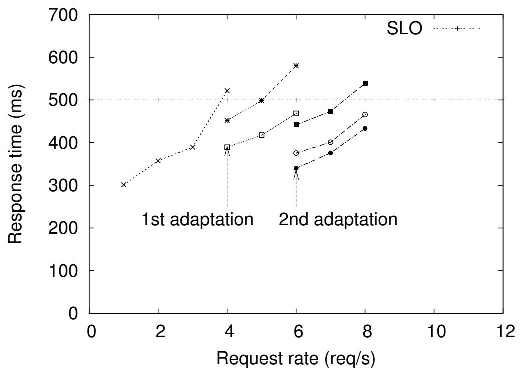


(a) Using Amazon's Elastic Load Balancer

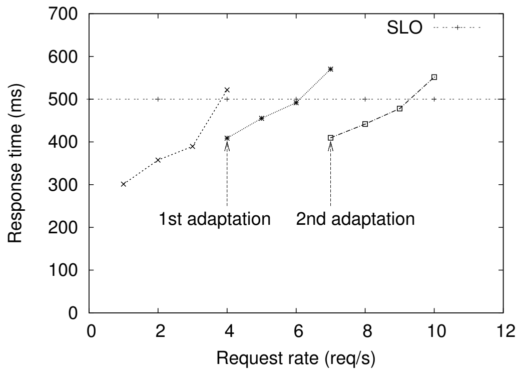


(b) Using our system

Figure 7: Provisioning Ref_{CPU} under increasing workload



(a) Using Amazon's Elastic Load Balancer



(b) Using our system

Figure 8: Provisioning $Ref_{I/O}$ under increasing workload

our system, the third instance (a very fast one) is given a higher workload than the others so the system requires a fourth instance only above 22.7 req/s.

Figure 8 shows similar results for the $Ref_{I/O}$ application. Here as well, our system balances traffic between instances such that they exhibit identical performance, whereas ELB creates significant performance differences between the instances. Our system can sustain up to 9 req/s when using three instances, while ELB can sustain only 7 req/s.

These results show that one should employ adaptive load balancing to correctly assign weights to forwarding instances when distributing traffics in Cloud. By doing so, one can achieve homogeneous performance from heterogeneous instances and make more efficient usage of these instances.

5.3 Effectiveness of Performance Prediction and Resource Provisioning

We now demonstrate the effectiveness of our system to provision multi-tier Web applications. In this scenario, in addition to using our load balancer, we also need to predict the performance that each new machine would have if it was added to the application server or database server tiers to decide which tier a new instance should be assigned to. The Amazon cloud does not have standard automatic mechanisms for driving such choices so we do not compare our approach with Amazon's resource provisioning service.

We use our system to provision the TPC-W e-commerce benchmark using the "shopping mix" workload. This standard workload generates 80% of read-only interactions, and 20% of read-write interactions. We set the SLO of the response time of TPC-W to be 500 ms. We increase the workload by creating corresponding numbers of Emulated Browsers (EBs). Each EB simulates a single user who browses the application. Whenever an EB leaves the application, a new EB is automatically create to maintain a constant load.

When the overall response time of the application violates the SLO, we request a new instance from the Cloud and profile it using the reference application. Thanks to the performance correlations between the tiers of TPC-W and the reference application, we use the performance profile of the new instance to predict the performance of any tier of TPC-W if it was using the new instance.

Table 1: Prediction accuracy during the first experiment run

	Adapt at 90 EBs			Adapt at 160 EBs			Adapt at 210 EBs			Adapt at 270 EBs		
	Real	Predicted	Error	Real	Predicted	Error	Real	Predicted	Error	Real	Predicted	Error
Provision the AS tier	554.6 ms	596.7 ms	+7.6%	578.3 ms	625.1 ms	+8.1%	458.3 ms	490.1 ms	+6.9%	**232.5 ms**	**248.3 ms**	+6.8%
Provision the DB tier	**165.4 ms**	**188.1 ms**	+13.7%	**189.7 ms**	**203.4 ms**	+7.2%	**156.2 ms**	**166.4 ms**	+6.5%	313.4 ms	329.1 ms	+5.0%

Table 2: Prediction accuracy during the second experiment run

	Adapt at 60 EBs			Adapt at 130 EBs			Adapt at 220 EBs			Adapt at 300 EBs		
	Real	Predicted	Error	Real	Predicted	Error	Real	Predicted	Error	Real	Predicted	Error
Provision the AS tier	511.7 ms	567.3 ms	+10.9%	427.9 ms	453.7 ms	+6.0%	**177.5 ms**	**192.3 ms**	+6.9%	541.9 ms	579.2 ms	+6.9%
Provision the DB tier	**152.4 ms**	**168.2 ms**	+10.4%	**218.2 ms**	**230.7 ms**	+5.7%	281.4 ms	302.7 ms	+7.6%	**151.2 ms**	**163.2 ms**	+7.9%

Finally, we compare the performance gains if the new instance was assigned to different tiers of TPC-W and select the tier which gives most performance benefit. We run the entire experiment twice: our provisioning system takes different decisions depending on the characteristics of the machine instances it gets from the Cloud.

Figures 9(a) illustrates the response time of TPC-W in the first run of the experiment. The application violates its SLO around a workload of 90 EBs. We request a new instance from the Cloud, profile it, and predict that it would be most useful if it was assigned to the database tier. When we push the workload further, it adds another database server at 160 EBs, then yet another database server at 210 EBs, then finally an application server at 270 EBs.

Figure 9(b) shows that, if we run the exact same experiment a second time, the machine instances we obtain from the Cloud have different performances. This leads the resource provisioning to take different decisions. It adds two database servers respectively at 60 and 130 EBs, then an application server at 220 EBs, then another database server at 300 EBs.

We can see here that SLO violations occur at different workloads, depending on the performance of the machine instances running the application. We also see that our resource provisioning effectively distinguishes different performance profiles, and takes provisioning decisions accordingly. In particular, at the third adaptation, the first run decides to use the new machine instance as a database server while the second run decides to use its own new machine instance as an application server.

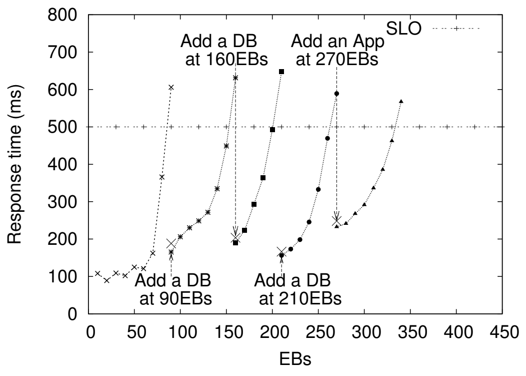
At each adaptation point, the resource provisioning system issues two predictions: it predicts what the new response time of the overall application would be if we assigned the new machine instance to be an application server or a database server. At each adaptation point we also tested the accuracy of these two predictions by deploying each of the two tiers in the new instances and measuring the actual application performance. Tables 1 and 2 show the measured and predicted response times of the whole application at each adaptation point. We can see that all predictions remain within 14% of the measured response times. This level of accuracy is sufficient to take correct provisioning decisions: in this set of experiments, the provisioning always identifies the best use it can make of the new machine instance it received (written in bold text in the table).

5.4 Comparison with other provision techniques

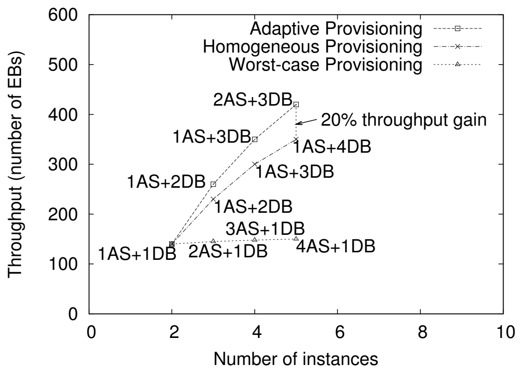
So far we showed the effectiveness of our system to provisioning TPC-W on EC2 by assigning heterogeneous instances to the tier where it gives maximum performance gain. We now demonstrate that our system can improve the throughput of TPC-W running on EC2 compared with two other provisioning techniques: "Homogeneous Provisioning" and "Worst-case Provisioning".

"Homogeneous Provisioning" provisions instances assuming that the performance of these instances is homogeneous. "Homogeneous Provisioning" first profiles the performance of the first two virtual instances hosting TPC-W. At each adaptation, "Homogeneous Provisioning" predicts the performance gains of new instances at each tier using the initial performance profiles, and assigns a new instance to the tier which receives maximum performance gain. "Homogeneous Provisioning" dispatches requests between instances using the round-robin policy. "Worst-case Provisioning" employs our algorithm to first figure out the tier to which a new instance should be assigned. However, "Worst-case Provisioning" systematically adopts the worst possible option. For instance, "Worst-case Provisioning" assigns a new instance to the application server tier if our system decides to assign this instance to the database tier. "Worst-case Provisioning" employs the same load balancing in our system. For comparison, we name our system as "Adaptive Provisioning".

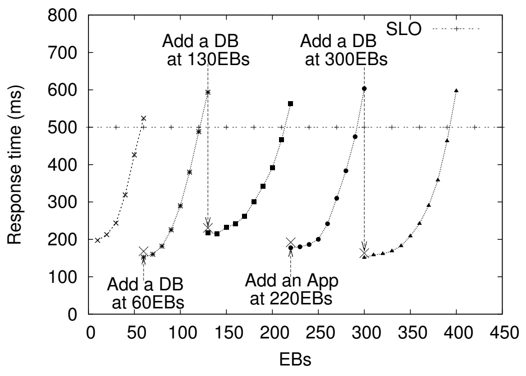
We first use the three techniques to provision TPC-W on EC2 with increasing workload separately. We set the SLO to 500 ms and measure the maximum throughput that a system configuration can sustain before violat-



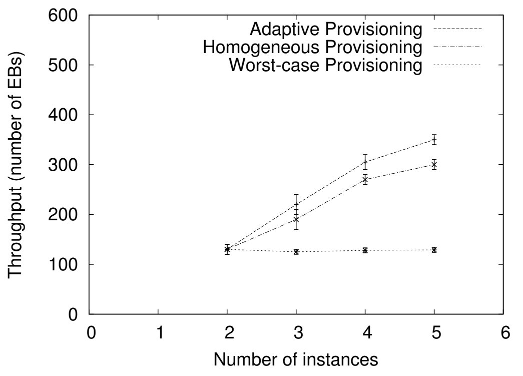
(a) First group



(a) Throughput comparison of single round



(b) Second group



(b) Statistical comparison of throughput over multiple rounds

Figure 9: Provisioning TPC-W under increasing workload

Figure 10: Throughput comparison of three provisioning techniques

ing the SLO. We also record the instance configurations at each adaptation. Figure 10(a) shows the throughputs achieved by each provisioning technique during a single run of the system under increasing workload and the corresponding instance configurations at each adaptation. The three provisioning systems use the exact same instances in the same order so their respective performance can be compared.

"Worst-case Provisioning" decides to provision the application server tier at each adaptation and finally supports around 150 EBs with 5 instances. "Homogeneous Provisioning" and "Adaptive Provisioning" both decide to provision new instances to the database server tier at the first and second adaptation. However, they achieve different throughput at the first two adaptations. The throughput difference is caused by the different load balancing capability of adapting to heterogeneous instances used in each provision technique. At the third adaptation, "Adaptive Provisioning" decides to assign the new instance to the application server tier while "Homogeneous Provisioning" decides to assign the same new instance to the database server tier. After the third adaptation, "Adaptive Provisioning" supports around 420 EBs while "Homogeneous Provisioning" supports around 350 EBs. This represents a 20% gain in throughput.

We then run the same experiment 5 rounds, each with a different set of EC2 small instances. Within each round, we measure the throughput achieved by each provisioning technique using a certain number of instances. The throughputs achieved in different rounds are different due to the performance heterogeneity of small instances. Figure 10(b) shows the average and standard deviation of the throughput achieved by each provisioning technique across multiple rounds. As previously, the "Worst-case Provisioning" behaves as the statistical lower bound of the achievable throughput of TPC-W on EC2. When taking the first adaptation, "Adaptive Provisioning" and "Homogeneous Provisioning" behave similar in terms of achieved throughput. However, when taking more adaptations, "Adaptive Provisioning" supports 17% higher throughput than "Homogeneous Provisioning". This demonstrates that our system makes more efficient use of heterogeneous instances in Cloud and achieves higher throughput using the same resources.

6 Conclusion

Cloud computing provides Web application providers with an attracting paradigm to dynamically vary the number of resources used by their application according to the current workload. However, Cloud computing platforms also have important limitations. In particular, dynamic resource provisioning is made difficult by the fact that each virtual instance has its own individual performance characteristics. Standard resource provisioning techniques provided by Cloud platforms do not take this performance heterogeneity into account, and therefore end up wasting resources.

We demonstrated in this paper that taking performance heterogeneity into account in a resource provisioning system can be practical and bring significant resource savings. One must first capture the performance relationships between different tiers of an application. When the application's workload makes it necessary to provision a new instance, we can efficiently capture its own performance profile, and use this information to drive the resource provisioning decisions: first, it allows us to decide to which tier this new machine instance should be assigned. Second, it allows us to adjust load balancing to make better use of the processing resources of each machine instance.

We hope that these results will allow the creation of new Cloud products such as automated performance monitoring and prediction as a service, and performance-aware load balancers. Providing Cloud users with such tools would allow them to make more efficient use of Cloud resources, and would thereby further increase the attractiveness of Cloud technologies for Web application providers.

7 Acknowledgments

Chi-Hung Chi is supported by the National Natural Science Foundation of China, Project Number 61033006.

References

[1] Amazon EC2: Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[2] DEJUN, J., PIERRE, G., AND CHI-HUNG, C. EC2 performance analysis for resource provisioning of service-oriented applications. In *Proceedings of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing* (Nov. 2009). LNCS 6275.

[3] DEJUN, J., PIERRE, G., AND CHI-HUNG, C. Autonomous resource provisioning for multi-service web applications. In *Proceedings of the 19th Intl. World Wide Web Conference* (Apr. 2010).

[4] ELNIKETY, S., DROPSHO, S., CECCHET, E., AND ZWAENEPOEL, W. Predicting replicated database scalability from standalone database profiling. In *Proceedings of the 4th EuroSys Conference* (Apr. 2009).

[5] KALYVIANAKI, E., CHARALAMBOUS, T., AND HAND, S. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using Kalman filters. In *Proceedings of the ICAC Conference* (June 2009).

[6] MARIN, G., AND MELLOR-CRUMMEY, J. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the SIGMETRICS Conference* (June 2004).

[7] NATHUJI, R., KANSAL, A., AND GHAFFARKHAH, A. Q-Clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th EuroSys conference* (Apr. 2010).

[8] OSTERMANN, S., IOSUP, A., YIGITBASI, N., PRODAN, R., FAHRINGER, T., AND EPEMA, D. A performance analysis of EC2 cloud computing services for scientific computing. In *Proceedings of the CloudComp conference* (Oct. 2010).

[9] SIVASUBRAMANIAN, S. *Scalable hosting of web applications*. PhD thesis, Vrije Universiteit Amsterdam, the Netherlands, Apr. 2007.

[10] STEWART, C., KELLY, T., ZHANG, A., AND SHEN, K. A dollar from 15 cents: Cross-platform management for internet services. In *Proceedings of the USENIX Annual Technical Conference* (June 2008).

[11] TPC-W: A transactional web e-commerce benchmark. http://www.tpc.org/tpcw.

[12] URGAONKAR, B., PACIFICI, G., SHENOY, P., SPREITZER, M., AND TANTAWI, A. An analytical model for multi-tier internet services and its applications. In *Proceedings of the SIGMETRICS Conference* (June 2005).

[13] URGAONKAR, B., PRASHANT, S., ABHISHEK, C., PAWAN, G., AND TIMOTHY, W. Agile dynamic provisioning of multi-tier internet applications. *ACM Transaction on Autonomous Adaptive System* (Mar. 2008).

[14] ZHENG, W., BIANCHINI, R., JANAKIRAMAN, G. J., SANTOS, J. R., AND TURNER, Y. JustRunIt: Experiment-based management of virtualized data centers. In *Proceedings of the USENIX Annual Technical Conference* (June 2009).

C3: An Experimental, Extensible, Reconfigurable Platform for HTML-based Applications

Benjamin S. Lerner Brian Burg
University of Washington

Herman Venter Wolfram Schulte
Microsoft Research

Abstract

The common conception of a (client-side) *web application* is some collection of HTML, CSS and JavaScript (JS) that is hosted within a web browser and that interacts with the user in some non-trivial ways. The common conception of a *web browser* is a monolithic program that can render HTML, execute JS, and gives the user a portal to navigate the web. Both of these are misconceptions: nothing inherently confines webapps to a browser's page-navigation idiom, and browsers can do far more than merely render content. Indeed, browsers and web apps are converging in functionality, but their underlying technologies are so far largely distinct.

We present C3, an implementation of the HTML/CSS/JS platform designed for web-client research and experimentation. C3's typesafe, modular architecture lowers the barrier to webapp and browser research. Additionally, C3 explores the role of *extensibility* throughout the web platform for customization and research efforts, by introducing novel extension points and generalizing existing ones. We discuss and evaluate C3's design choices for flexibility, and provide examples of various extensions that we and others have built.

1 Introduction

We spend vast amounts of time using web browsers: casual users view them as portals to the web, while power users enjoy them as flexible, sophisticated tools with countless uses. Researchers of all stripes view browsers and the web itself as systems worthy of study: browsers are a common thread for web-related research in fields such as HCI, security, information retrieval, sociology, software engineering, and systems research. Yet today's production-quality browsers are all monolithic, complex systems that do not lend themselves to easy experimentation. Instead, researchers often must modify the source code of the browsers—usually tightly-optimized, obscure, and sprawling C/C++ code—and this requirement of deep domain knowledge poses a high barrier to entry, correctness, and adoption of research results.

Of course, this is a simplified depiction: browsers are not entirely monolithic. Modern web browsers, such as Internet Explorer, Firefox or Chrome, support *extensions*, pieces of code—usually a mix of HTML, CSS and JavaScript (JS)—that are written by third-party developers and downloaded by end users, that build on top of the browser and customize it dynamically at runtime.[1] To date, such customizations focus primarily on modifying the user interfaces of browsers. (Browsers also support *plug-ins*, binary components that provide functionality, such as playing new multimedia file types, not otherwise available from the base browser. Unlike extensions, plug-ins cannot interact directly with each other or extend each other further.)

Extensions are widely popular among both users and developers: millions of Firefox users have downloaded thousands of different extensions over two billion times[2]. Some research projects have used extensions to implement their ideas. But because current extension mechanisms have limited power and flexibility, many research projects still must resort to patching browser sources:

1. XML3D [14] defines new HTML tags and renders them with a 3D ray-tracing engine—but neither HTML nor the layout algorithm are extensible.
2. Maverick [12] permits writing device drivers in JS and connecting the devices (e.g., webcams, USB thumb drives, GPUs, etc.) to web pages—but JS cannot send raw USB packets to the USB root hub.
3. RePriv [5] experiments with new ways to securely expose and interact with private browsing informa-

[1] Opera supports widgets, which do not interact with the browser or content, and Safari recently added small but slowly-growing support for extensions in a manner similar to Chrome. We ignore these browsers in the following discussions.

[2] `https://addons.mozilla.org/en-US/statistics/`

tion (e.g. topics inferred from browsing history) via reference-monitored APIs—but neither plug-ins nor JS extensions can guarantee the integrity or security of the mined data as it flows through the browser.

These projects incur development and maintenance costs well above the inherent complexity of their added functionality. Moreover, patching browser sources makes it difficult to update the projects for new versions of the browsers. This overhead obscures the fact that such research projects are essentially extensions to the web-browsing experience, and would be much simpler to realize on a flexible platform with more powerful extension mechanisms. Though existing extension points in mainstream browsers vary widely in both design and power, none can support the research projects described above.

1.1 The extensible future of web browsers

Web browsers have evolved from their beginnings as mere document viewers into web-application runtime platforms. Applications such as Outlook Web Access or Google Documents are sophisticated programs written in HTML, CSS and JS that use the browser *only for rendering and execution* and ignore everything else browsers provide (bookmarks, navigation, tab management, etc.). Projects like Mozilla Prism[3] strip away all the browser "chrome" while reusing the underlying HTML/CSS/JS implementation (in this case, Gecko), letting webapps run like native apps, outside of the typical browser. Taken to an extreme, "traditional" applications such as Firefox or Thunderbird are written using Gecko's HTML/CSS/JS engine, and clearly are not themselves hosted within a browser.

While browsers and web apps are growing closer, they are still mostly separate with no possibility of tight, customizable integration between them. Blogging clients such as WordPress, instant messaging clients such as Gchat, and collaborative document editors such as Mozilla Skywriter are three disjoint web applications, all designed to create and share content. An author might be using all three simultaneously, and searching for relevant web resources to include as she writes. Yet the only way to do so is to "escape the system", copying and pasting web content via the operating system.

1.2 Contributions

The time has come to reconsider browser architectures with a focus on extensibility. We present C3: a reconfigurable, extensible implementation of HTML, CSS and JS designed for web client research and experimentation. C3 is written entirely in C# and takes advantage of .Net's libraries and type-safety. Similar to Firefox building atop Gecko, we have built a prototype browser atop C3, using only HTML, CSS and JS.

[3]http://prism.mozillalabs.com/

By *reconfigurable*, we mean that each of the modules in our browser—Document Object Model (DOM) implementation, HTML parser, JS engine, etc.—is loosely coupled by narrow, typesafe interfaces and can be replaced with alternate implementations compiled separately from C3 itself. By *extensible*, we mean that the default implementations of the modules support run-time extensions that can be systematically introduced to

1. extend the syntax and implementation of HTML
2. transform the DOM when being parsed from HTML
3. extend the UI of the running browser
4. extend the environment for executing JS, and
5. transform and modify running JS code.

Compared to existing browsers, C3 introduces novel extension points (1) and (5), and generalizes existing extension points (2)–(4). These extension points are treated in order in Section 3. We discuss their functionality and their security implications with respect to the same-origin policy [13]. We also provide examples of various extensions that we and others have built.

The rest of the paper is structured as follows. Section 2 gives an overview of C3's architecture and highlights the software engineering choices made to further our modularity and extensibility design goals. Section 3 presents the design rationale for our extension points and discusses their implementation. Section 4 evaluates the performance, expressiveness, and security implications of our extension points. Section 5 describes future work. Section 6 concludes.

2 C3 architecture and design choices

As a research platform, C3's explicit design goals are architectural modularity and flexibility where possible, instead of raw performance. Supporting the various extension mechanisms above requires hooks at many levels of the system. These goals are realized through careful design and implementation choices. Since many requirements of an HTML platform are standardized, aspects of our architecture are necessarily similar to other HTML implementations. C3 lacks some of the features present in mature implementations, but contains all of the essential architectural details of an HTML platform.

C3's clean-slate implementation presented an opportunity to leverage modern software engineering tools and practices. Using a managed language such as C# sidesteps the headaches of memory management, buffer overruns, and many of the common vulnerabilities in production

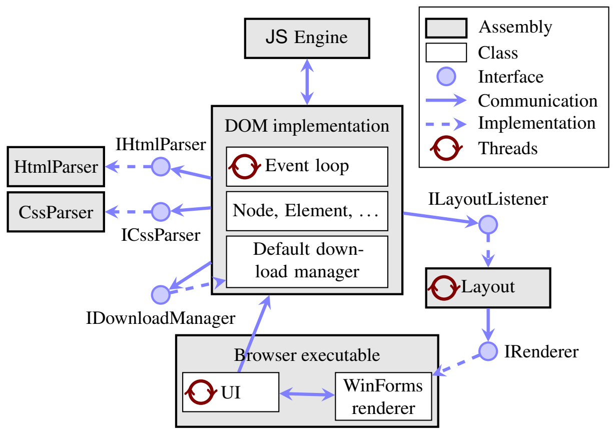


Figure 1: C3's modular architecture

browsers. Using a higher-level language better preserves abstractions and simplifies many implementation details. Code Contracts [4] are used throughout C3 to ensure implementation-level invariants and safety properties—something that is not feasible in existing browsers.

Below, we sketch C3's module-level architecture, and elaborate on several core design choices and resulting customization opportunities. We also highlight features that enable the extension points examined in Section 3.

2.1 Pieces of an HTML platform

The primary task of any web platform is to parse, render, and display an HTML document. For interactivity, web applications additionally require the managing of events such as user input, network connections, and script evaluation. Many of these sub-tasks are independent; Figure 1 shows C3's module-level decomposition of these tasks. The *HTML parser* converts a text stream into an object tree, while the *CSS parser* recognizes stylesheets. The *JS engine* dispatches and executes event handlers. The *DOM implementation* implements the API of DOM nodes, and implements bindings to expose these methods to JS scripts. The *download manager* handles actual network communication and interactions with any on-disk cache. The *layout engine* computes the visual structure and appearance of a DOM tree given current CSS styles. The *renderer* displays a computed layout. The browser's UI displays the output of the renderer on the screen, and routes user input to the DOM.

2.2 Modularity

Unlike many modern browsers, C3's design embraces loose coupling between browser components. For example, it is trivial to replace the HTML parser, renderer frontend, or JS engine without modifying the DOM implementation or layout algorithm. To make such drop-in replacements feasible, C3 shares *no* data structures between modules when possible (i.e., each module is heap-disjoint). This design decision also simplifies threading disciplines, and is further discussed in Section 2.7.

Simple implementation-agnostic interfaces describe the operations of the DOM implementation, HTML parser, CSS parser, JS engine, layout engine, and front-end renderer modules. Each module is implemented as a separate .Net assembly, which prevents modules from breaking abstractions and makes swapping implementations simple. Parsers could be replaced with parallel [8] or speculative[4] versions; layout might be replaced with a parallel [11] or incrementalizing version, and so on. The default module implementations are intended as straightforward, unoptimized reference implementations. This permits easy per-module evaluations of alternate implementation choices.

2.3 DOM implementation

The DOM API is a large set of interfaces, methods and properties for interacting with a document tree. We highlight two key design choices in our implementation: what the object graph for the tree looks like, and the bindings of these interfaces to C# classes. Our choices aim to minimize overhead and "boilerplate" coding burdens for extension authors.

Object trees: The DOM APIs are used throughout the browser: by the HTML parser (Section 2.4) to construct the document tree, by JS scripts to manipulate that tree's structure and query its properties, and by the layout engine to traverse and render the tree efficiently. These clients use distinct but overlapping subsets of the APIs, which means they must be exposed both to JS and to C#, which in turn leads to the first design choice.

One natural choice is to maintain a tree of "implementation" objects in the C# heap separate from a set of "wrapper" objects in the JS heap[5] containing pointers to their C# counterparts: the JS objects are a "view" of the underlying C# "model". The JS objects contain stubs for all the DOM APIs, while the C# objects contain implementations and additional helper routines. This design incurs the overheads of extra pointer dereferences (from the JS APIs to the C# helpers) and of keeping the wrappers synchronized with the implementation tree. However, it permits specializing both representations for their respective usages, and the extra indirection enables

[4]http://hsivonen.iki.fi/speculative-html5-parsing/

[5]Expert readers will recognize that "objects in the JS heap" are implemented by C# "backing" objects; we are distinguishing these from C# objects that do not "back" any JS object.

multiple views of the model: This is essentially the technical basis of Chrome extensions' "isolated worlds" [1], where the indirection is used to ensure security properties about extensions' JS access to the DOM. Firefox also uses the split to improve JS memory locality with "compartments" [15].

By contrast, C3 instead uses a single tree of objects visible to both languages, with each DOM node being a C# subclass of an ordinary JS object, and each DOM API being a standard C# method that is exposed to JS. This design choice avoids both overheads mentioned above. Further, Spur [3], the tracing JS engine currently used by C3, can trace from JS into DOM code for better optimization opportunities. To date, no other DOM implementation/JS engine pair can support this optimization.

DOM language bindings: The second design choice stems from our choice for the first: how to represent DOM objects such that their properties are callable from both C# and JS. This representation must be open: extensions such as XML3D must be able to define new types of DOM nodes that are instantiable from the parser (see Section 3.1) and capable of supplying new DOM APIs to both languages as well. Therefore any new DOM classes must subclass our C# DOM class hierarchy easily, and be able to use the same mechanisms as the built-in DOM classes. Our chosen approach is a thin marshaling layer around a C# implementation, as follows:

- All Spur JS objects are instances of C# classes deriving from `ObjectInstance`. Our DOM class hierarchy derives from this too, and so DOM objects *are* JS objects, as above.
- All JS objects are essentially property bags, or key/value dictionaries, and "native" objects (e.g. `Math`, `Date`) may contain properties that are implemented by the JS runtime and have access to runtime-internal state. All DOM objects are native, and their properties (the DOM APIs) access the internal representation of the document.
- The JS dictionary is represented within Spur as a `TypeObject` field of each `ObjectInstance`. To expose a native method on a JS object, the implementation simply adds a property to the `TypeObject` mapping the (JS) name to the (C#) function that implements it.[6] This means that a single C# function can be called from both languages, and need not be implemented twice.

The `ObjectInstance` and `TypeObject` classes are public Spur APIs, and so our DOM implementation is readily extensible by new node types.

[6]Technically, to a C# function that unwraps the JS values into strongly-typed C# values, then calls a second C# function with them.

2.4 The HTML parser

The HTML parser is concerned with transforming HTML source into a DOM tree, just as a standard compiler's parser turns source into an AST. Extensible compilers' parsers can recognize supersets of their original language via extensions; similarly, C3's default HTML parser supports extensions that add new HTML tags (which are implemented by new C# DOM classes as described above; see also Section 3.1).

An extensible HTML parser has only two dependencies: a means for constructing a new node given a tag name, and a factory method for creating a new node and inserting it into a tree. This interface is far simpler than that of any DOM node, and so exists as the separate `INode` interface. The parser has no hard dependency on a specific DOM implementation, and a minimal implementation of the `INode` interface can be used to test the parser independently of the DOM implementation. The default parser implementation is given a DOM node factory that can construct `INode`s for the built-in HTML tag names. Extending the parser via this factory is discussed in Section 3.1.

2.5 Computing visual structure

The layout engine takes a document and its stylesheets, and produces as output a *layout tree*, an intermediate data structure that contains sufficient information to display a visual representation of the document. The *renderer* then consults the layout tree to draw the document in a platform- or toolkit-specific manner.

Computing a layout tree requires three steps: first, DOM nodes are attributed with style information according to any present stylesheets; second, the layout tree's structure is determined; and third, nodes of the layout tree are annotated with concrete styles (placement and sizing, fonts and colors, etc.) for the renderer to use. Each of these steps admits a naïve reference implementation, but both more efficient and more extensible algorithms are possible. We focus on the former here; layout extensibility is revisited in Section 3.3.

Assigning node styles The algorithm that decorates DOM nodes with CSS styles does not depend on any other parts of layout computation. Despite the top-down implementation suggested by the name "cascading style sheets", several efficient strategies exist, including recent and ongoing research in parallel approaches [11].

Our default style "cascading" algorithm is self-contained, single-threaded and straightforward. It decorates each DOM node with an immutable calculated style object, which is then passed to the related layout tree node during construction. This immutable style suffices thereafter in determining visual appearance.

Determining layout tree structure The layout tree is generated from the DOM tree in a single traversal. The two trees are approximately the same shape; the layout tree may omit nodes for invisible DOM elements (e.g. ⟨**script**/⟩), and may insert "synthetic" nodes to simplify later layout invariants. For consistency, this transformation must be serialized between DOM mutations, and so runs on the DOM thread (see Section 2.7). The layout tree must preserve a mapping between DOM elements and the layout nodes they engender, so that mouse movement (which occurs in the renderer's world of screen pixels and layout tree nodes) can be routed to the correct target node (i.e. a DOM element). A naïve pointer-based solution runs afoul of an important design decision: C3's architectural goals of modularity require that the layout and DOM trees share no pointers. Instead, all DOM nodes are given unique numeric ids, which are *preserved* by the DOM-to-layout tree transformation. Mouse targeting can now be defined in terms of these ids while preserving pointer-isolation of the DOM from layout.

Solving layout constraints The essence of any layout algorithm is to solve constraints governing the placement and appearance of document elements. In HTML, these constraints are irregular and informally specified (if at all). Consequently the constraints are typically solved by a manual, multi-pass algorithm over the layout tree, rather than a generic constraint-solver [11]. The manual algorithms found in production HTML platforms are often tightly optimized to eliminate some passes for efficiency.

C3's architecture admits such optimized approaches, too; our reference implementation keeps the steps separate for clarity and ease of experimentation. Indeed, because the layout tree interface does not assume a particular implementation strategy, several layout algorithm variants have been explored in C3 with minimal modifications to the layout algorithm or components dependent on the computed layout tree.

2.6 Accommodating Privileged UI

Both Firefox and Chrome implement some (or all) of their user interface (e.g. address bar, tabs, etc.) in declarative markup, rather than hard-coded native controls. In both cases this gives the browsers increased flexibility; it also enables Firefox's extension ecosystem. The markup used by these browsers is *trusted*, and can access internal APIs not available to web content. To distinguish the two, trusted UI files are accessed via a different URL scheme: e.g., Firefox's main UI is loaded using `chrome://browser/content/browser.xul`.

We chose to implement our prototype browser's UI in HTML for two reasons. First, we wanted to experiment with writing sophisticated applications entirely within the HTML/CSS/JS platform and experience first-hand what challenges arose. Even in our prototype, such experience led to the two security-related changes described below. Secondly, having our UI in HTML opens the door to the extensions described in Section 3; the entirety of a C3-based application is available for extension. Like Firefox, our browser's UI is available at a privileged URL: launching C3 with a command-line argument of `chrome://browser/tabbrowser.html` will display the browser UI. Launching it with the URL of any website will display that site without any surrounding browser chrome. Currently, we only permit HTML file resources bundled within the C3 assembly itself to be given privileged `chrome://` URLs.

Designing this prototype exposed deliberate limitations in HTML when examining the navigation history of child windows (popups or ⟨**iframe**/⟩s): the APIs restrict access to same-origin sites only, and are write-only. A parent window cannot see what site a child is on unless it is from the same origin as the parent, and can never see what sites a child has visited. A browser must avoid both of these restrictions so that it can implement the address bar.

Rather than change API visibility, C3 extends the DOM API in two ways. First, it gives privileged pages (i.e., from `chrome://` URLs) a new `childnavigated` notification when their children are navigated, just before the `onbeforeunload` events that the children already receive. Second, it treats `chrome://` URLs as trusted origins that *always pass same-origin checks*. The trusted-origin mechanism and the custom navigation event suffice to implement our browser UI.

2.7 Threading architecture

One important point of flexibility is the mapping between threads and the HTML platform components described above. We do not impose any threading discipline beyond necessary serialization required by HTML and DOM standards. This is made possible by our decision to prevent data races by design: in our architecture, data is either immutable, or it is not shared amongst multiple components. Thus, it is possible to choose any threading discipline within a single component; a single thread could be shared among all components for debugging, or several threads could be used within each component to implement worker queues.

Below, we describe the default allocation of threads among components, as well as key concurrency concerns for each component.

2.7.1 The DOM/JS thread(s)

The DOM event dispatch loop and JS execution are single-threaded within a set of related web pages[7]. "Separate" pages that are unrelated[8] can run entirely parallel with each other. Thus, sessions with several tabs or windows open simultaneously use multiple DOM event dispatch loops.

In C3, each distinct event loop consists of two threads: a *mutator* to run script and a *watchdog* to abort run-away scripts. Our system maintains the invariant that all mutator threads are *heap-disjoint*: JS code executing in a task on one event loop can only access the DOM nodes of documents sharing that event loop. This invariant, combined with the single-threaded execution model of JS (from the script's point of view), means all DOM nodes and synchronous DOM operations can be lock-free. (Operations involving local storage are asynchronous and must be protected by the storage mutex.) When a window or ⟨**iframe**/⟩ is navigated, the relevant event loop may change. An event loop manager is responsible for maintaining the mappings between windows and event loops to preserve the disjoint-heap invariant.

Every DOM manipulation (node creation, deletion, insertion or removal; attribute creation or modification; etc.) notifies any registered *DOM listener* via a straightforward interface. One such listener is used to inform the layout engine of all document manipulations; others could be used for testing or various diagnostic purposes.

2.7.2 The layout thread(s)

Each top-level browser window is assigned a *layout* thread, responsible for resolving layout constraints as described in Section 2.5. Several browser windows might be simultaneously visible on screen, so their layout computations must proceed in parallel for each window to quickly reflect mutations to the underlying documents. Once the DOM thread computes a layout tree, it transfers ownership of the tree to the layout thread, and begins building a new tree. Any external resources necessary for layout or display (such as image data), are also passed to the layout thread as uninterpreted .Net streams. This isolates the DOM thread from any computational errors on the layout threads.

2.7.3 The UI thread

It is common for GUI toolkits to impose threading restrictions, such as only accessing UI widgets from their creating thread. These restrictions influence the platform insofar as *replaced elements* (such as buttons or text boxes) are implemented by toolkit widgets.

C3 is agnostic in choosing a particular toolkit, but rather exposes abstract interfaces for the few widget properties actually needed by layout. Our prototype currently uses the .Net WinForms toolkit, which designates one thread as the "UI thread", to which all input events are dispatched and on which all widgets must be accessed. When the DOM encounters a replaced element, an actual WinForms widget must be constructed so that layout can in turn set style properties on that widget. This requires synchronous calls from the DOM and layout threads to the UI thread. Note, however, that *responding* to events (such as mouse clicks or key presses) is *asynchronous*, due to the indirection introduced by numeric node ids: the UI thread simply adds a message to the DOM event loop with the relevant ids; the DOM thread will process that message in due course.

[7]We ignore for now web-workers, which are an orthogonal concern.

[8]Defining when pages are actually separate is non-trivial, and is a refinement of the same-origin policy, which in turn has been the subject of considerable research [7, 2]

3 C3 Extension points

The extension mechanisms we introduce into C3 stem from a principled examination of the various semantics of HTML. Our interactions with webapps tacitly rely on manipulating HTML in two distinct ways: we can interpret it *operationally* via the DOM and JS programs, and we can interpret it *visually* via CSS and its associated layout algorithms. Teasing these interpretations apart leads to the following two transformation pipelines:

- JS global object + HTML source^{1,2}

  $\xrightarrow{\text{HTML } parsing}$ ^{3}DOM subtrees^{4}

  $\xrightarrow{\texttt{onload}}$ DOM document^{5}

  $\xrightarrow{\text{JS } events}$ DOM document . . .

- DOM document + CSS source^{6}

  $\xrightarrow{\text{CSS } parsing}$ CSS content model^{7}

  $\xrightarrow{layout}$ CSS box model

The first pipeline distinguishes four phases of the document lifecycle, from textual sources through to the event-based running of JS: the initial `onload` event marks the transition point after which the document is asserted to be fully loaded; before this event fires, the page may be inconsistent as critical resources in the page may not yet have loaded, or scripts may still be writing into the document stream.

Explicitly highlighting these pipeline stages leads to designing extension points in a *principled* way: we can extend the inputs accepted or the outputs produced by each stage, as long as we produce outputs that are acceptable inputs to the following stages. This is in contrast to

```
public interface IDOMTagFactory {
  IEnumerable<Element> TagTemplates { get; }
}
```

```
public class HelloWorldTag : Element {
  string TagName { get { return "HelloWorld"; } }
  ...
}
public class HelloWorldFactory : IDOMTagFactory {
  IEnumerable<Element> TagTemplates { get {
    yield return new HelloWorldTag();
  } }
}
```

Figure 2: Factory and simple extension defining new tags

the extension models of existing browsers, which support various extension points without relating them to other possibilities or to the browser's behavior as a whole. The extension points engendered by the pipelines above are (as numbered):

1. Before beginning HTML parsing, extensions may provide *new tag names and DOM-node implementations* for the parser to support.

2. Before running any scripts, extensions may *modify the JS global scope* by adding or removing bindings.

3. Before inserting subtrees into the document, extensions may *preprocess* them using arbitrary C# code.

4. Before firing the `onload` event, extensions may declaratively inject new content into the nearly-complete tree using *overlays*.

5. Once the document is complete and events are running, extensions may modify existing event handlers using *aspects*.

6. Before beginning CSS parsing, extensions may provide *new CSS properties and values* for the parser to support.

7. Before computing layout, extensions may provide *new layout box types and implementations* to affect layout and rendering.

Some of these extension points are simpler than others due to regularities in the input language, others are more complicated, and others are as yet unimplemented. Points (1) and (5) are novel to C3. C3 does not yet implement points (6) or (7), though they are planned future work; they are also novel. We explain points (1), (3) and (4) in Section 3.1, points (2) and (5) in Section 3.2, and finally points (6) and (7) in Section 3.3.

3.1 HTML parsing/document construction

Point (1): New tags and DOM nodes The HTML parser recognizes concrete syntax resembling ⟨**tagName** attrName="val"/⟩ and constructs new DOM nodes for each tag. In most browsers, the choices of which tag names to recognize, and what corresponding objects to construct, are tightly coupled into the parser. In C3, however, we abstract both of these decisions behind a factory, whose interface is shown in the top of Figure 2.[9] Besides simplifying our code's internal structure, this approach permits extensions to contribute factories too.

Our default implementation of this interface provides one "template" element for each of the standard HTML tag names; these templates inform the parser which tag names are recognized, and are then cloned as needed by the parser. Any unknown tag names fall back to returning an `HTMLUnknownElement` object, as defined by the HTML specification. However, if an extension contributes another factory that provides additional templates, the parser seamlessly can clone *those* instead of using the fallback: effectively, this extends the language recognized by the parser, as XML3D needed, for example. A trivial example that adds support for a ⟨**HelloWorld**/⟩ tag is shown in Figure 2. A more realistic example is used by C3 to support overlays (see Figure 4 and below).

The factory abstraction also gives us the flexibility to support additional experiments: rather than adding *new* tags, a researcher might wish to *modify existing tags*. Therefore, we permit factories to provide a new template for existing tag names—and we require that at most one extension does so per tag name. This permits extensions to easily subclass the C3 DOM implementation, e.g. to add instrumentation or auditing, or to modify existing functionality. Together, these extensions yield a parser that accepts a superset of the standard HTML tags and still produces a DOM tree as output.

Point (3): Preprocessing subtrees The HTML 5 parsing algorithm produces a document tree in a bottom-up manner: nodes are created and then attached to parent nodes, which eventually are attached to the root DOM node. Compiler-authors have long known that it is useful to support *semantic actions*, callbacks that examine or preprocess subtrees as they are constructed. Indeed, the HTML parsing algorithm itself specifies some behaviors that are essentially semantic actions, e.g., "when an ⟨**img**/⟩ is inserted into the document, download the referenced image file". Extensions might use this ability to collect statistics on the document, or to sanitize it during construction. These actions typically are local—they examine just the newly-inserted tree—and rarely mutate

[9]Firefox seems not to use a factory; Chrome uses one, but the choice of factory is fixed at compile-time. C3 can load factories dynamically.

```
public interface IParserMutatorExtension {
  IEnumerable<string> TagNamesOfInterest { get; }
  void OnFinishedParsing(Element element);
}
```

Figure 3: The interface for HTML parser semantic actions

Base constructions	
⟨**overlay**/⟩	Root node of extension document
⟨**insert** selector="*selector*" where="before|after"/⟩	Insert new content adjacent to all nodes matched by CSS *selector*
⟨**replace** selector="*selector*"/⟩	Replace existing subtrees matching *selector* with new content
⟨**self** attrName="value"… /⟩	Used within ⟨**replace**/⟩, refers to node being replaced and permits modifying its attributes
⟨**contents**/⟩	Used within ⟨**replace**/⟩, refers to children of node being replaced
Syntactic sugar	
⟨**before** … /⟩	⟨**insert** where="before"… /⟩
⟨**after** … /⟩	⟨**insert** where="after"… /⟩
⟨**modify** selector="*sel*" where="before"⟩ ⟨**self** *new attributes*⟩ *new content* ⟨/**self**⟩ ⟨/**modify**⟩	⟨**replace** selector="*sel*"⟩ ⟨**self** *new attributes*⟩ *new content* ⟨**contents**/⟩ ⟨/**self**⟩ ⟨/**replace**⟩ and likewise for where="after"

Figure 4: The overlay language for document construction extensions. The bottom set of tags are syntactic sugar.

the surrounding document. (In HTML in particular, because inline scripts execute *during* the parsing phase, the document may change arbitrarily between two successive semantic-action callbacks, and so semantic actions will be challenging to write if they are not local.)

Extensions in C3 can define custom semantic actions using the interface shown in Figure 3. The interface supplies a list of tag names, and a callback to be used when tags of those names are constructed.

Point (4): Document construction Firefox pioneered the ability to both define application UI and define extensions to that UI using a single declarative markup language (XUL), an approach whose success is witnessed by the variety and popularity of Firefox's extensions. The fundamental construction is the *overlay*, which behaves like a "tree-shaped patch": the children of the ⟨**overlay**/⟩ select nodes in a target document and define content to be inserted into or modified within them, much as hunks within a patch select lines in a target text file. C3 adapts and generalizes this idea for HTML.

Our implementation adds eight new tags to HTML, shown in Figure 4, to define overlays and the various actions they can perform. As they are a language extension to HTML, we inform the parser of these new tags using the `IDOMTagFactory` described above.[10] Overlays can ⟨**insert**/⟩ or ⟨**replace**/⟩ elements, as matched by CSS selectors. To support *modifying* content, we give overlays the ability to refer to the target node (⟨**self**/⟩) or its ⟨**contents**/⟩. Finally, we define syntactic sugar to make overlays easier to write.

```
⟨overlay⟩
  ⟨modify selector="head" where="after"⟩
    ⟨self⟩
      ⟨style⟩
        li > #bullet { color:  blue; }
      ⟨/style⟩
    ⟨/self⟩
  ⟨/modify⟩
  ⟨before selector="li > *:first-child"⟩
    ⟨span class="bullet"⟩&bull;⟨/span⟩
  ⟨/before⟩
⟨/overlay⟩
```

Figure 5: Simulating list bullets (in language of Fig. 4)

Figure 5 shows a simple but real example used during development of our system, to simulate bulleted lists while generated content support was not yet implemented. It appends a ⟨**style**/⟩ element to the end of the ⟨**head**/⟩ subtree (and fails if no ⟨**head**/⟩ element exists), and inserts a ⟨**span**/⟩ element at the beginning of each ⟨**li**/⟩.

The subtlety of defining the semantics of overlays lies in their interactions with scripts: when should overlays be applied to the target document? Clearly overlays must be applied after the document structure is present, so a strawman approach would apply overlays "when parsing finishes". This exposes a potential inconsistency, as scripts that run *during* parsing would see a partial, not-yet-overlaid document, with nodes a and b adjacent, while scripts that run after parsing would see an overlaid document where a and b may no longer be adjacent. However, the HTML specification offers a way out: the DOM raises a particular event, `onload`, that indicates the document has finished loading and is ready to begin execution. Prior to that point, the document structure is in flux—and so we choose to apply overlays *as part of that flux*, immediately before the `onload` event is fired. This may break poorly-coded sites, but in practice has not been an issue with Firefox's extensions.

[10]We apply the overlays using just one general-purpose callback within our code. This callback could be factored as a standalone, ad-hoc extension point, making overlays themselves truly an extension to C3.

3.2 JS execution

Point (2): Runtime environment Extensions such as Maverick may wish to inject new properties into the JS global object. This object is an input to all scripts, and provides the initial set of functionality available to pages. As an input, it must be constructed before HTML parsing begins, as the constructed DOM nodes should be consistent with the properties available from the global object: e.g., `document.body` must be an instance of `window.HTMLBodyElement`. This point in the document's execution is stable—no scripts have executed, no nodes have been constructed—and so we permit extensions to manipulate the global object as they please. (This could lead to inconsistencies, e.g. if they modify `window.HTMLBodyElement` but do not replace the implementation of ⟨**body**/⟩ tags using the prior extension points. We ignore such buggy extensions for now.)

Point (5): Scripts themselves The extensions described so far modify discrete pieces of implementation, such as individual node types or the document structure, because there exist ways to name each of these resources *statically*: e.g., overlays can examine the HTML source of a page and write CSS selectors to name parts of the structure. The analogous extension to script code needs to modify the sources of individual functions. Many JS idioms have been developed to achieve this, but they all suffer from JS's *dynamic* nature: function names do *not* exist statically, and scripts can create new functions or alias existing ones at runtime; no static inspection of the scripts' sources can precisely identify these names. Moreover, the common idioms used by extensions today are brittle and prone to silent failure.

C3 includes our prior work [10], which addresses this disparity by modifying the JS compiler to support *aspect oriented programming* using a dynamic weaving mechanism to advise closures (rather than variables that point to them). Only a dynamic approach can detect runtime-evaluated functions, and this requires compiler support to advise all aliases to a function (rather than individual names). As a side benefit, aspects' integration with the compiler often improves the performance of the advice: in the work cited, we successfully evaluated our approach on the sources of twenty Firefox extensions, and showed that they could express nearly all observed idioms with shorter, clearer and often faster code.

3.3 CSS and layout

Discussion An extensible CSS engine permits incrementally adding new features to layout in a modular, clean way. The CSS 3 specifications themselves are a step in this direction, breaking the tightly-coupled CSS 2.1 specification into smaller pieces. A true test of our proposed extension points' expressiveness would be to implement new CSS 3 features, such as generated content or the flex-box model, as extensions. An even harder test would be to extricate older CSS 2 features, such as floats, and re-implement them as compositional extensions. The benefit to successfully implementing these extensions is clear: a stronger understanding of the semantics of CSS features.

We discovered the possibility of these CSS extension points quite recently, in exploring the consequences of making each stage of the layout pipeline extensible "in the same way" as the DOM/JS pipeline is. To our knowledge, implementing the extension points below has not been done before in any browser, and is planned future work.

Point (6): Parsing CSS values We can extend the CSS language in four ways: 1) by adding new property names and associated values, 2) by recognizing new values for existing properties, 3) by extending the set of selectors, or 4) by adding entirely new syntax outside of style declaration blocks. The latter two are beyond the scope of an extension, as they require more sweeping changes to both the parser and to layout, and are better suited to an alternate implementation of the CSS parser altogether (i.e., a different configuration of C3).

Supporting even just the first two extension points is nontrivial. Unlike HTML's uniform tag syntax, nearly every CSS attribute has its own idiosyncratic syntax:

```
font: italic bold 10pt/1.2em "Gentium", serif;
margin: 0 0 2em 3pt;
display: inline-block;
background-image: url(mypic.jpg);
...
```

However, a style declaration itself is very regular, being a semicolon-separated list of colon-separated name/value pairs. Moreover, the CSS parsing algorithm discards any un-parsable attributes (up to the semicolon), and then parse the rest of the style declaration normally.

Supporting the first extension point—new property names—requires making the parser table-driven and registering value-parsing routines for each known property name. Then, like HTML tag extensions, CSS property extensions can register new property names and callbacks to parse the values. (Those values must never contain semicolons, or else the underlying parsing algorithm would not be able to separate one attribute from another.)

Supporting the second extension point is subtler. Unlike the HTML parser's uniqueness constraint on tag names, here multiple extensions might contribute new values to an existing property; we must ensure that the syntaxes of such new values do not overlap, or else provide some ranking to choose among them.

Point (7): Composing layout The CSS layout algorithm describes how to transform the document tree (the *content model*) into a tree of boxes of varying types, appearances and positions. Some boxes represent lines of text, while others represent checkboxes, for example. This transformation is not obviously compositional: many CSS properties interact with each other in non-trivial ways to determine precisely which types of boxes to construct. Rather than hard-code the interactions, the layout transformation must become table-driven as well. Then both types of extension above become easy: extensions can create new box subtypes, and patch entries in the transformation table to indicate when to create them.

4 Evaluation

The C3 platform is rapidly evolving, and only a few extensions have yet been written. To evaluate our platform, we examine: the performance of our extension points, ensuring that the benefits are not outweighed by huge overheads; the expressiveness, both in the ease of "porting" existing extensions to our model and in comparison to other browsers' models; and the security implications of providing such pervasive customizations.

4.1 Performance

Any time spent running the extension manager or conflict analyses slows down the perceived performance of the browser. Fortunately, this process is very cheap: with one extension of each supported type, it costs roughly 100ms to run the extensions. This time includes: enumerating all extensions (27ms), loading all extensions (4ms), and detecting parser-tag conflicts (3ms), mutator conflicts (2ms), and overlay conflicts (72ms). All but the last of these tasks runs just once, at browser startup; overlay conflict detection must run per-page. Enumerating all extensions currently reads a directory, and so scales linearly with the number of extensions. Parser and mutator conflict detection scale linearly with the number of extensions as well; overlay conflict detection is more expensive as each overlay provides more interacting constraints than other types of extensions do. If necessary, these costs can be amortized further by caching the results of conflict detection between browser executions.

4.2 Expressiveness

Figure 6 lists several examples of extensions available for IE, Chrome, and Firefox, and the corresponding C3 extension points they would use if ported to C3. Many of these extensions simply overlay the browser's user interface and require no additional support from the browser. Some, such as Smooth Gestures or LastTab, add or revise UI functionality. As our UI is entirely script-driven, we support these via script extensions. Others, such as the various Native Client libraries, are sandboxed programs that are then exposed through JS objects; we support the JS objects and .Net provides the sandboxing.

Figure 6 also shows some research projects that are not implementable as extensions in any other browser except C3. As described below, these projects extend the HTML language, CSS layout, and JS environment to achieve their functionality. Implementing these on C3 requires *no hacking of C3* , leading to a much lower learning curve and fewer implementation pitfalls than modifying existing browsers. We examine some examples, and how they might look in C3, in more detail here.

4.2.1 XML3D: Extending HTML, CSS and layout

XML3D [14] is a recent project aiming to provide 3D scenes and real-time ray-traced graphics for web pages, in a declarative form analogous to ⟨**svg**/⟩ for two-dimensional content. This work uses XML namespaces to define new scene-description tags and requires *modifying each browser* to recognize them and construct specialized DOM nodes accordingly. To style the scenes, this work must modify the CSS engine to recognize new style attributes. Scripting the scenes and making them interactive requires constructing JS objects that expose the customized properties of the new DOM nodes. It also entails informing the browser of a new scripting language (AnySL) tailored to animating 3D scenes.

Instead of modifying the browser to recognize new tag names, we can use the new-tag extension point to define them in an extension, and provide a subclassed ⟨**script**/⟩ implementation recognizing AnySL. Similarly, we can provide new CSS values and new box subclasses for layout to use. The full XML3D extension would consist of these four extension hooks and the ray-tracer engine.

4.2.2 Maverick: Extensions to the global scope

Maverick [12] aims to connect devices such as webcams or USB keys to web content, by writing device drivers in JS and connecting them to the devices via Native Client (NaCl) [17]. NaCl exposes a socket-like interface to web JS over which all interactions with native modules are multiplexed. To expose its API to JS, Maverick injects an actual DOM ⟨**embed**/⟩ node into the document, stashing state within it, and using JS properties on that object to communicate with NaCl. This object can then transliterate the image frames from the webcam into Base64-encoded `src` URLs for other scripts' use in ⟨**img**/⟩ tags, and so reuse the browser's image decoding libraries.

There are two main annoyances with Maverick's implementation that could be avoided in C3. First, NaCl

Extensions	Available from	C3-equivalent extension points used
IE:		
Explorer bars		(4) overlay the main browser UI
Context menu items		(4) overlay the context menu in the browser UI
Accelerators		(4) overlay the context menu
WebSlices		(4) overlay browser UI
Chrome:		
Gmail checkers	`https://chrome.google.com/extensions/search?q=gmail`	(4) overlay browser UI, (5) script advice
Skype	`http://go.skype.com/dc/clicktocall`	(4) overlay browser UI, (2) new JS objects, (5) script advice
Smooth Gestures	`http://goo.gl/rN5Y`	(4) overlay browser UI, (5) script advice
Native Client libraries	`http://code.google.com/p/nativeclient/`	(2) new JS objects
Firefox:		
TreeStyleTab	`https://addons.mozilla.org/en-US/firefox/addon/5890/`	(4) overlay tabbar in browser UI, inject CSS
LastTab	`https://addons.mozilla.org/en-US/firefox/addon/112/`	(5) script advice
Perspectives	[16]	(5) script extensions, (4) overlay error UI
Firebug	`http://getfirebug.com/`	(4) overlays, (5) script extensions, (2) new JS objects
Research projects:		
XML3D	[14]	(1) new HTML tags, (6) new CSS values, (7) new layouts
Maverick	[12]	(2) new JS objects
Fine	[6]	(1) HTML ⟨**script**/⟩ tag replacement
RePriv	[5]	(2) new JS objects

Figure 6: Example extensions in IE, Firefox, and Chrome, as well as research projects best implemented in C3, and the C3 extension points that they might use

isolates native modules in a strong sandbox that prevents direct communication with resources like devices; Maverick could not be implemented in NaCl without *modifying the sandbox* to expose a new system call and writing untrusted glue code to connect it to JS; in C3, trusted JS objects can be added without recompiling C3 itself. Second, implementing Maverick's exposed API requires carefully managing low-level NPAPI routines that must mimic JS's name-based property dispatch; in C3, exposing properties can simply *reuse* the JS property dispatch, as in Section 2.3.

Ultimately, using a DOM node to expose a device is not the right abstraction: it is not a node in the document but rather a global JS object like `XMLHttpRequest`. And while using Base64-encoded URLs is a convenient implementation trick, it would be far more natural to call the image-decoding libraries directly, avoiding both overhead and potential transcoding errors.

4.2.3 RePriv: Extensions hosting extensions

RePriv [5] runs in the background of the browser and mines user browsing history to infer personal interests. It carefully guards the release of that information to websites, via APIs whose uses can be verified to avoid unwanted information leakage. At the same time, it offers its own extension points for site-specific "interest miners" to use to improve the quality of inferred information. These miners are all scheduled to run during an `onload` event handler registered by RePriv. Finally, extensions can be written to use the collected information to reorganize web pages at the client to match the user's interests.

While this functionality is largely implementable as a plug-in in other browsers, several factors make it much easier to implement in C3. First and foremost, RePriv's security guarantees rely on C3 being entirely managed code: we can remove the browser from RePriv's trusted computing base by isolating RePriv extensions in an AppDomain and leveraging .Net's freedom from common exploits such as buffer overflows. Obtaining such a strong security guarantee in other browsers is at best very challenging. Second, the document construction hook makes it trivial for RePriv to install the `onload` event handler. Third, AppDomains ensure the memory isolation of every miner from each other and from the DOM of the document, except as mediated by RePriv's own APIs; this makes proving the desired security properties much easier. Finally, RePriv uses Fine [6] for writing its interest miners; since C3, RePriv and Fine target .Net, RePriv can reuse .Net's assembly-loading mechanisms.

4.3 Other extension models

4.3.1 Extensions to application UI

Internet Explorer 4.0 introduced two extension points permitting customized toolbars (Explorer Bars) and context-menu entries. These extensions were written in native C++ code, had full access to the browser's internal DOM representations, and could implement essentially any functionality they chose. Unsurprisingly, early extensions often compromised the browser's security and stability. IE 8 later introduced two new extension points that permitted self-updating bookmarks of web-page snippets (Web Slices) and context-menu items to speed access to repetitive tasks (Accelerators), providing safer implementations of common uses for Explorer Bars and context menus.

The majority of IE's interface is not modifiable by extensions. By contrast, Firefox explored the possibility that entire application interfaces could be implemented in a markup language, and that a declarative extension mechanism could *overlay* those UIs with new constructions. Research projects such as Perspectives change the way Firefox's SSL connection errors are presented, while others such as Xmarks or Weave synchronize bookmarks and user settings between multiple browsers. The UI for these extensions is written in precisely the same declarative way as Firefox's own UI, making it as simple to extend Firefox's browser UI as it is to design any website.

But the single most compelling feature of these extensions is also their greatest weakness: they permit implementing features that were never anticipated by the browser designers. End users can then install multiple such extensions, thereby losing any assurance that the composite browser is stable, or even that the extensions are compatible with each other. Indeed, Chrome's carefully curtailed extension model is largely a reaction to the instabilities often seen with Firefox extensions. Chrome permits extensions only minimal change to the browser's UI, and prevents interactions between extensions. For comparison, Chrome directly implements bookmarks and settings synchronization, and now permits extension context-menu actions, but the Perspectives behavior remains unimplementable by design.

Our design for overlays is based strongly on Firefox's declarative approach, but provides stronger semantics for overlays so that we can detect and either prevent or correct conflicts between multiple extensions. We also generalized several details of Firefox's overlay mechanism for greater convenience, without sacrificing its analyzability.

4.3.2 Extensions to scripts

In tandem with the UI extensions, almost the entirety of Firefox's UI behaviors are driven by JS, and again extensions can manipulate those scripts to customize those behaviors. A similar ability lets extensions modify or inject scripts within web pages. Extensions such as LastTab change the tab-switching order from cyclic to most-recently-used, while others such as Ghostery block so-called "web tracking bugs" from executing. Firefox exposes a huge API, opening basically the entire platform to extension scripts. This flexibility also poses a problem: multiple extensions may attempt to modify the same scripts, often leading to broken or partially-modified scripts with unpredictable consequences.

Modern browser extension design, like Firefox's Jetpack or Chrome's extensions, are typically developed using HTML, JS, and CSS. While Firefox "jetpacks" are currently still fully-privileged, Chrome extensions run in a sandboxed process. Chrome extensions cannot access privileged information and cannot crash or hang the browser. While these new guarantees are necessary for the stability of a commercial system protecting valuable user information, they also restrict the power of extensions.

One attempt to curtail these scripts' interactions with each other within web pages is the Fine project [6]. Instead of directly using JS, the authors use a dependently-typed programming language to express the precise read- and write-sets of extension scripts, and a security policy constrains the information flow between them. Extensions that satisfy the security policy are provably non-conflicting. The Fine project can target C3 easily, either by compiling its scripts to .Net assemblies and loading them dynamically (by subclassing the ⟨**script**/⟩ tag), or by statically compiling its scripts to JS and dynamically injecting them into web content (via the JS global-object hook). Guha et al. successfully ported twenty Chrome extensions to Fine and compiled them to run on C3 with minimal developer effort.

As mentioned earlier, C3 includes our prior work on aspect-oriented programming for JS [10], permitting extensions clearer language mechanisms to express how their modifications apply to existing code. Beyond the performance gains and clarity improvements, by eliminating the need for brittle mechanisms and exposing the intent of the extension, compatibility analyses between extensions become feasible.

4.4 Security considerations

Of the five implemented extension points, two are written in .Net and have full access to our DOM internals. In particular, new DOM nodes or new JS runtime objects that subclass our implementation may use protected DOM fields inappropriately and violate the same-origin policy. We view this flexibility as both an asset and a liability: it permits researchers to experiment with alternatives to the SOP, or to prototype enhancements to HTML and the DOM. At the same time, we do not advocate these extensions for web-scale use. The remaining extension points are either limited to safe, narrow .Net interfaces or are written in HTML and JS and inherently subject to the SOP. Sanitizing potentially unsafe .Net extensions to preserve the SOP is itself an interesting research problem. Possible approaches include using .Net AppDomains to segregate extensions from the main DOM, or static analyses to exclude unsafe accesses to DOM internals.

5 Future work

We have focused so far on the abilities extensions have within our system. However, the more powerful extensions become, the more likely they are to conflict with one another. Certain extension points are easily amenable to conflict detection; for example, two parser tag extensions cannot both contribute the same new tag name. However, in previous work we have shown that defining conflicts precisely between overlay extensions, or between JS runtime extensions, is a more challenging task [9] .

Assuming a suitable notion of extension conflict exists for each extension type, it falls to the extension loading mechanism to ensure that, whenever possible, conflicting extensions are not loaded. In some ways this is very similar to the job of a compile-time linker, ensuring that all modules are compatible before producing the executable image. Such load-time prevention gives users a much better experience than in current browsers, where problems never surface until runtime. However not all conflicts are detectable statically, and so some runtime mechanism is still needed to detect conflict, blame the offending extension, and prevent the conflict from recurring.

6 Conclusion

We presented C3, a platform implementing of HTML, CSS and JS, and explored how its design was tuned for easy reconfiguration and runtime extension. We presented several motivating examples for each extension point, and confirmed that our design is at least as expressive as existing extension systems, supporting current extensions as well as new ones not previously possible.

References

[1] Barth, A., Felt, A. P., Saxena, P., and Boodman, A. Protecting browsers from extension vulnerabilities. In *NDSS* (2010).

[2] Barth, A., Weinberger, J., and Song, D. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *SSYM'09: Proceedings of the 18th conference on USENIX security symposium* (Berkeley, CA, USA, 2009), USENIX Association, pp. 187–198.

[3] Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., and Venter, H. SPUR: A trace-based JIT compiler for CIL. In *OOPSLA/SPLASH '10: Proceedings of the 25th ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications* (New York, NY, USA, 2010), ACM.

[4] Fähndrich, M., Barnett, M., and Logozzo, F. Embedded contract languages. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing* (New York, NY, USA, 2010), ACM, pp. 2103–2110.

[5] Fredrikson, M., and Livshits, B. RePriv: Re-envisioning in-browser privacy. Tech. rep., Microsoft Research, Aug. 2010.

[6] Guha, A., Fredrikson, M., Livshits, B., and Swamy, N. Verified security for browser extensions. MSR-TR to be available 11/01, September 2010.

[7] Jackson, C., and Barth, A. Beware of finer-grained origins. In *In Web 2.0 Security and Privacy (W2SP 2008)* (2008).

[8] Jones, C. G., Liu, R., Meyerovich, L., Asanovic, K., and Bodík, R. Parallelizing the Web Browser. In *HotPar '09: Proceedings of the Workshop on Hot Topics in Parallelism* (March 2009), USENIX.

[9] Lerner, B. S., and Grossman, D. Language support for extensible web browsers. In *APLWACA '10: Proceedings of the 2010 Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications* (New York, NY, USA, 2010), ACM, pp. 39–43.

[10] Lerner, B. S., Venter, H., and Grossman, D. Supporting dynamic, third-party code customizations in JavaScript using aspects. In *OOPSLA '10: Companion of the 25th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2010), ACM.

[11] Meyerovich, L. A., and Bodik, R. Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on the World Wide Web* (2010), WWW '10, pp. 711–720.

[12] Richardson, D. W., and Gribble, S. D. Maverick: Providing web applications with safe and flexible access to local devices. In *Proceedings of the 2011 USENIX Conference on Web Application Development* (June 2011), WebApps'11.

[13] Ruderman, J. Same origin policy for javascript, Oct. 2010.

[14] Sons, K., Klein, F., Rubinstein, D., Byelozyorov, S., and Slusallek, P. XML3D: interactive 3d graphics for the web. In *Web3D '10: Proceedings of the 15th International Conference on Web 3D Technology* (New York, NY, USA, 2010), ACM, pp. 175–184.

[15] Wagner, G., Gal, A., Wimmer, C., Eich, B., and Franz, M. Compartmental memory management in a modern web browser. In *Proceedings of the International Symposium on Memory Management* (June 2011), ACM. To appear.

[16] Wendlandt, D., Andersen, D. G., and Perrig, A. Perspectives: Improving ssh-style host authentication with multi-path probing. In *Proceedings of the USENIX Annual Technical Conference (Usenix ATC)* (June 2008).

[17] Yee, B., Sehr, D., Dardyk, G., Chen, J., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on* (May 2009), pp. 79 –93.

The Effectiveness of Application Permissions

Adrienne Porter Felt*, Kate Greenwood, David Wagner
University of California, Berkeley
apf, kate_eli, daw@cs.berkeley.edu

Abstract

Traditional user-based permission systems assign the user's full privileges to all applications. Modern platforms are transitioning to a new model, in which each application has a different set of permissions based on its requirements. Application permissions offer several advantages over traditional user-based permissions, but these benefits rely on the assumption that applications generally require less than full privileges. We explore whether that assumption is realistic, which provides insight into the value of application permissions.

We perform case studies on two platforms with application permissions, the Google Chrome extension system and the Android OS. We collect the permission requirements of a large set of Google Chrome extensions and Android applications. From this data, we evaluate whether application permissions are effective at protecting users. Our results indicate that application permissions can have a positive impact on system security when applications' permission requirements are declared up-front by the developer, but can be improved.

1 Introduction

Browsers and smartphone operating systems provide development platforms that support thriving markets for third-party applications. However, third-party code creates risks for the user. Some third-party authors are malicious [3, 14], and third-party code can introduce vulnerabilities because the authors of third-party applications usually are not security experts [10, 19].

In order to protect users from the threats associated with third-party code, modern platforms use application permissions to control access to security- and privacy-relevant parts of their APIs. Users decide whether to allow individual applications to access these sensitive resources. *Time-of-use* systems prompt users to approve permissions as needed by applications at runtime, and *install-time* systems ask developers to declare their applications' permission requirements up-front so that users can grant them during installation.

Traditional user-based permission systems assign the user's full privileges to all of the user's applications. In the application permission model, however, each application can have a customized set of permissions based on its individual privilege requirements. If most applications can be satisfied with less than the user's full privileges, then three advantages of application permissions over the traditional user-based model are possible:

- **User Consent:** Security-conscious users may be hesitant to grant access to dangerous permissions without justification. For install-time systems, this might alert some users to malware at installation; for time-of-use systems, this can prevent an installed application from accessing sensitive content.
- **Defense in Depth:** For install-time systems, the impact of a vulnerability in an application will be limited to the vulnerable application's declared privileges. This could also be true for a time-of-use system in which developers declare their applications' maximum possible permissions up-front.
- **Review Triaging:** Up-front application permission declarations facilitate central review because security reviewers can ignore low-privilege applications and focus on applications with dangerous permissions. This may decrease the average review time.

The real-world impact of these potential advantages depends on low application permission requirements. We evaluate the practical benefits of application permissions. by performing a large-scale study of Google Chrome extensions and Android applications.

We perform a measurement study that quantifies the permission use of 1000 Google Chrome extensions and 956 Android applications. Both platforms use install-time permissions. Our study provides detailed data on the permission requirements of applications in the wild. From this data, we assess whether the platforms are realizing the potential benefits of application permissions.

We find that almost all applications ask for fewer than maximum permissions. Only 14 of 1000 extensions request the most dangerous privileges, and the average Android application requests fewer than 4 of 56 available

*This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

dangerous permissions. In comparison, all typical desktop applications receive the equivalent of all 56 Android permissions. These results indicate that application permission systems with up-front permission declarations can decrease the impact of application vulnerabilities and simplify review. This supports the adoption of install-time permission systems. Current time-of-use platforms do not require up-front permission declarations, which means that they do not provide the same defense in depth or review triaging benefits. However, time-of-use platforms could gain these advantages by requiring developers to state the maximum set of permissions that the application will require at runtime.

Although developers use fewer than full permissions, Google Chrome and Android users are presented with at least one dangerous permission request during the installation of almost every extension and application. Warning science literature indicates that frequent warnings desensitize users, especially if most warnings do not lead to negative consequences [15, 11]. Users are therefore not likely to pay attention to or gain information from install-time permission prompts in these systems. Changes to these permission systems are necessary to reduce the number of permission warnings shown to users.

We examine the effects of developer incentives, developer error, wildcard permissions, and permission granularity on permission usage. We find that more than 10% of applications request unneeded permissions. Our results show that developers are willing to make use of fine-grained permissions, motivating a fine-grained permission design. We suggest error detection tools and other platform changes to reduce permission requests.

We view the Google Chrome and Android permission systems as case studies for the future of application permissions. Our primary contribution is a large-scale study that demonstrates the defense in depth and review triaging benefits of application permissions with up-front declarations. This motivates the addition of up-front developer permission declarations to time-of-use permission systems. We also discuss tools and changes that would improve the effectiveness of these systems. Our results should guide the design of in-progress and future permission systems, and we also provide concrete suggestions for Google Chrome and Android.

2 Background

Popular platforms with application permissions include Apple iOS, the Safari extension system, and Facebook. In-progress platforms with application permissions include Mozilla Jetpack, the W3C device API for web applications, and the Google installable web application platform. In this paper, we focus on the Google Chrome extension system and Android application platform, which feature install-time permissions.

2.1 Google Chrome Extensions

Browser extension platforms allow third-party code to run as part of the browser environment. *Extensions* change the user's browsing experience by editing web sites and changing browser behavior. All extensions are free from the official Google Chrome extension gallery.

Google Chrome extensions can have three types of components: core extensions, content scripts, and plug-ins. A core extension comprises the main, persistent portion of an extension. Content scripts are injected into web sites; when the page loads, the content script's JavaScript executes in the context of the site. A content script has full access to its host site's page. Core extensions and content scripts are written in JavaScript, whereas plug-ins are native executables.

The extension gallery prompts the user with a warning (e.g., Figure 1) that indicates what privileges the extension has requested:

Plug-ins. Plug-ins are native executables, so a plug-in grants the extension full permissions to the user's machine. The installation warning for an extension with a plug-in says the extension "can access all data on your computer." Extensions with plug-ins are reviewed.

Browser managers. Core extensions can access the extension API, which is a set of browser managers. Each manager is controlled with one permission. The managers include history, bookmarks, and geolocation. The browser warns that extensions with these permissions can access "your browsing history," "bookmarks," and "your physical location," respectively. Non-security relevant browser managers also exist, but their use does not prompt a warning so we do not consider them.

Web access. The developer must specify web permissions for content scripts and core extensions. Content script web permissions determine which domains they are installed on by default. A core extension can send `XMLHttpRequests` (XHRs) and inject code into the domains it has permissions for. Content script and core extension domain permissions are listed separately.

All-domain access is the broadest web permission. If either the content scripts or the core extension have all-domain access, the browser warning states that the exten-

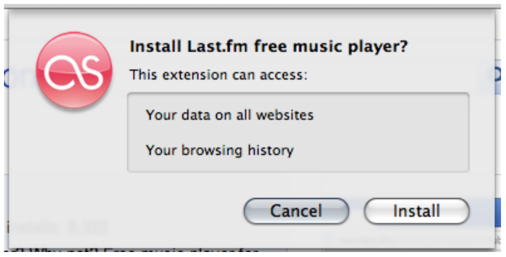
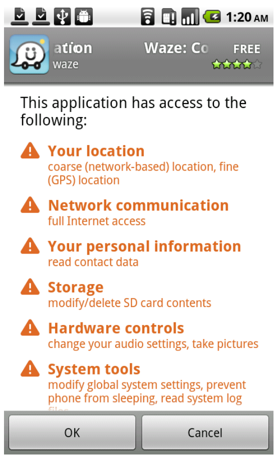

Install Last.fm free music player?
This extension can access:
Your data on all websites
Your browsing history

Figure 1: Google Chrome extension installation.



This application has access to the following:

Your location
coarse (network-based) location, fine (GPS) location

Network communication
full Internet access

Your personal information
read contact data

Storage
modify/delete SD card contents

Hardware controls
change your audio settings, take pictures

System tools
modify global system settings, prevent phone from sleeping, read system log

Figure 2: Android application installation.

sion "can access your data on all web sites." Alternately, a developer can list specific domains, using wildcards for protocols or subdomains (e.g., `*://*.bar.com`). The installation warning will list the requested domains.

2.2 Android Applications

The Android smartphone operating system supports third-party Java applications, which can be installed by users through the Android Market. Some third-party applications are free, and some are paid. Android applications do not need to be reviewed prior to inclusion in the Android Market, although other phone vendors like RIM and Apple maintain official review processes.

Android's API provides access to phones' cameras, microphones, GPS, text messages, WiFi, Bluetooth, etc. Most device access is controlled by permissions, although large parts of the overall API are not protected by permissions. Applications can define their own extra permissions, but we only consider permissions defined by the Android OS. There are 134 permissions in Android 2.2. Permissions are categorized into threat levels:

Normal. API calls with annoying but not harmful consequences are protected with Normal permissions. Examples include accessing information about available WiFi networks, vibrating the phone, and setting the wallpaper.

Dangerous. API calls with potentially harmful consequences are protected with Dangerous permissions. These include actions that could cost the user money or leak private information. Example permissions are the ones used to protect opening a network socket, recording audio, and using the camera.

Signature. The most sensitive operations are protected with Signature permissions. These permissions are only granted to applications that have been signed with the device manufacturer's certificate. Market applications are only eligible for Signature permissions if they are updates to applications that were pre-installed and signed by the device manufacturer. Requests for Signature permissions by other applications will be ignored. An example is the ability to inject user events.

SignatureOrSystem. This category includes signed applications and applications that are installed into the `/system/app` folder. Typically, this only includes pre-installed applications. Advanced users who have rooted their phones [9] can manually install applications into this folder, but the official Market installation process will not do so. Requests for SignatureOrSystem permissions by other applications will be ignored. For example, these permissions protect the ability to turn off the phone.

The Android Market displays a permission prompt to the user during installation for Dangerous permissions (e.g., Figure 2). Warnings for Dangerous permissions are grouped into functionality categories. For example, all Dangerous permissions related to location are displayed as part of the same location warning. Normal permissions can be viewed once the application is installed but are hidden behind a collapsed drop-down menu. Signature/System permissions are not displayed to users at all.

3 Permission Prevalence

We examine the frequency of permission requests in Google Chrome extensions and Android applications. These results should be compared to traditional systems that grant all applications full privileges.

3.1 Chrome Extensions

We study the 1000 "most popular" extensions, as ranked in the official Google Chrome extension gallery[1]. Of these, the 500 most popular extensions are relevant to user consent and application vulnerabilities because they comprise the majority of user downloads. The 500 less popular extensions are installed in very few browsers, but they are relevant to reviewers because reviewers would need to examine all extensions in the directory. Table 1 provides an overview of our results.

3.1.1 Popular Extensions

Of the 500 most popular extensions, 91.4% ask for at least one security-relevant permission. This indicates that nearly every installation of an extension generates at least one security warning[2].

[1]We crawled the directory on August 27, 2010.

[2]We discovered that Google Chrome sometimes fails to generate a warning for history access. The bug has been fixed for new versions [7]. Our analysis assumes that all requests for history access correctly generate a warning. The bug affects 5 of extensions in our set.

Permission	Popular	Unpopular
Plug-ins	2.80 %	0.00 %
Web access	82.0 %	60.8 %
All domains	51.6 %	21.8 %
Specific domains	30.4 %	39.0 %
Browser manager(s)	74.8 %	43.4 %

Table 1: We measure the prevalence of permissions in 1000 Google Chrome extensions, split into the 500 most popular and 500 less popular. For web access, we report the highest permission of either the content script or core extension.

Plug-ins. Only 14 of the 500 extensions include plug-ins.

Browser managers. The majority of security warnings are caused by the window manager, which is requested by almost 75% of the 500 extensions. Requesting access to the window manager generates a warning about history access because history is indirectly available through the window manager. The bookmark and geolocation managers are requested infrequently: 44 times and once, respectively.

All domains. Half of the 500 extensions request all-domain access for either content scripts or the core extension. 52% request access to all `http` sites, and 42% ask for all `https` sites.

Specific domains. One-third of extensions only request a set of specific domains. This reduces the attack surface and removes the possibility that an extension is snooping on sensitive web data.

No warning. Only 43 of the 500 extensions do not request access to a security-relevant permission. 38 do not ask for any permissions at all; they load normal web sites into their extension windows or apply “themes” to the user interface. The remainder use browser managers that are not relevant to privacy or security.

3.1.2 Unpopular Extensions

Not all of the extensions listed in the “most popular” directory ranking are popular. After approximately the first 500 of 1000 popularity-ranked extensions, the number of users per extension abruptly decreases, and applications are no longer ranked solely according to the number of users. (Although the ranking algorithm is private, we believe it incorporates time.) Figure 3 shows the transition. 16.2% of the bottom 500 extensions have fewer than ten users. These 500 low-ranked extensions are of uneven quality. E.g., two of them are unaltered versions of the example extension on the developer web site.

Table 1 presents the results of our survey of the 500 less popular extensions. 71.6% of the less popular extensions have at least one security-relevant permission. When compared to the top 500 extensions, the unpopular extensions request far fewer permissions than popular extensions. We hypothesize that this is because less popular extensions offer less functionality. All of the differences are significant at a 1% significance level.

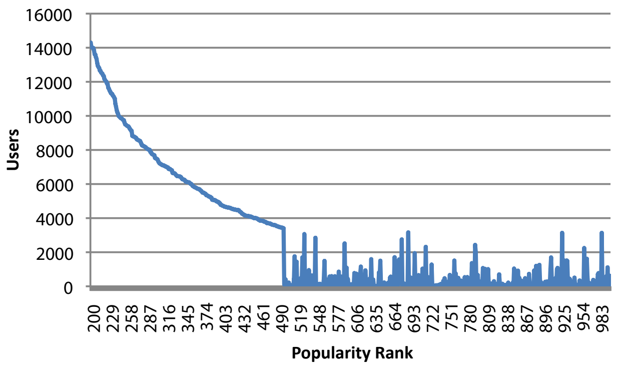


Figure 3: Users per extension. We omit the first 200 for graph clarity; the most popular extension has 1.3M users.

Unranked extensions are strictly less popular than the unpopular extensions in our data set. If one were to review the remaining 5,696 unranked Google Chrome extensions, we expect their permission requirements would be equivalent to or less than the permission requirements of these 500 unpopular applications. We note with caution that future studies on permissions need to consider the effect of popularity. E.g., a study that looks at the full set of 6,696 extensions to evaluate warning frequency would would likely underestimate the number of warnings that users see in practice by approximately 20%.

3.1.3 Evaluation

User Consent. Nearly all popular extensions (91% of the top 500) generate at least one security warning, which decreases the value of the warnings. History and all-domain permissions are requested by more than half of extensions; users have no reason to be suspicious of extensions with these permissions because they are not anomalous. However, warnings about plug-ins are rare and therefore potentially notable.

Defense in Depth. This study shows that the permission system dramatically reduces the scope of potential extension vulnerabilities. A negligible number of extensions include plug-ins, which means that the typical extension vulnerability cannot yield access to the local machine. This is a significant improvement over the Firefox and Internet Explorer extension systems, which provide *all* extensions with access to the local file system. We also find that all-domain access is frequent but not universal: 18% of popular extensions need no web access, and 30.4% only need limited web access. This means that the permission system prevents *half* of popular extensions from having unnecessary web privileges.

Review Triaging. Of the 1000 extensions in our study, only 1.4% require review under current Google Chrome review triaging procedures. (It should be noted, however, that we do not know how many extensions requiring review were submitted to the directory.) These results suggest that the Firefox extension review process could be significantly streamlined if Mozilla were to adopt a similar permission system. Reviewers could indisputably ignore 28% of submitted extensions regardless of the exact triaging criteria, based on the number of less-popular extensions with no security-relevant permissions.

3.2 Android Applications

We survey 100 paid and 856 free applications from the Android Market[3]. For the paid applications, we selected the 100 most popular. The free set is comprised of the 756 most popular and 100 most recently added applications. Unlike Google Chrome extensions, we observe no differences between popular and recently added free applications, so we present them together. It is possible that we do not see a popularity bias in Android applications because of differences in the developer communities and entrance barriers. We do not compare applications based on their categories in the Android Market; the categories are loosely defined and include a wide variety of different functionality [1]. Although Android applications written by the same developer could collude, we consider each application's permissions independently. Legitimate developers have no incentive to hide communication and circumvent permission warnings.

3.2.1 Dangerous Permissions

We are primarily concerned with the prevalence of Dangerous permissions, which are displayed as a warning to users during installation and can have serious security ramifications if abused. We find that 93% of free and 82% of paid applications have at least one Dangerous permission, i.e., generate at least one permission prompt.

Android permissions are grouped into functionality categories, and Table 1(a) shows how many applications use at least one Dangerous permission from each given category. This provides a relative measure of which parts of the protected API are used by applications. All of the permissions in a category display the same permission prompt, so Table 1(a) also indicates how often users see each type of permission request.

A small number of permissions are requested very frequently. Table 1(b) shows the most popular Dangerous permissions. In particular, the `INTERNET` permission is heavily used. We find that 14% of free and 4% of paid applications request `INTERNET` as their only Dangerous

[3]The applications were collected in October 2010.

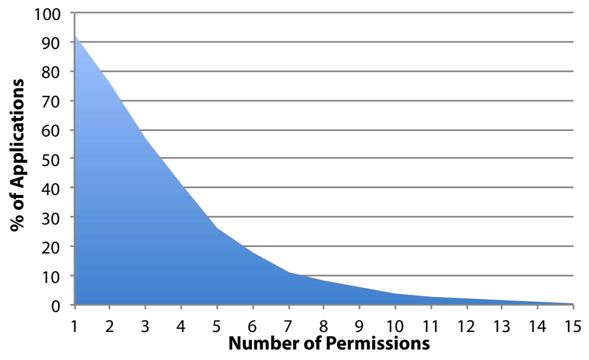


Figure 4: Percentage of paid and free applications with *at least* the given number of Dangerous permissions.

permission. Barrera et al. hypothesize that free applications often need the `INTERNET` permission only to load advertisements [1]. The disparity in `INTERNET` use between free and paid applications supports this hypothesis, although it is still the most popular permission for paid applications.

The prevalence of the `INTERNET` permission means that most applications with access to personal information also have the ability to leak it. For example, 97% of the 225 applications that ask for `ACCESS_FINE_LOCATION` also request the `INTERNET` permission. Similarly, 94% and 78% of the respective applications that request `READ_CONTACTS` and `READ_CALENDAR` also have the `INTERNET` permission. We find that significantly more free than paid applications request both Internet access and location data, which possibly indicates widespread leakage of location information to advertisers in free applications. This corroborates a previous study that found that 20 of 30 randomly selected free applications send user information to content or advertisement servers [5].

Although many applications ask for at least one Dangerous permission, the number of permissions required by an application is typically low. Even the most highly privileged application in our set asks for 26 permissions, which is less than half of the available 56 Dangerous permissions. Figure 4 shows the distribution of Dangerous permission requests. Paid applications use an average of 3.99 Dangerous permissions, and free applications use an average of 3.46 Dangerous permissions.

3.2.2 Signature and System Permissions

Applications can request Signature and SignatureOrSystem permissions, but the operating system will not grant the request unless the application has been signed by the device manufacturer (Signature) or installed in the `/system/app` folder (System). It is pointless for a typical application to request these permissions because the permission requests will be ignored.

(a) Prevalence of Dangerous permissions, by category.

Category	Free	Paid
NETWORK**	87.3 %	66 %
SYSTEM_TOOLS	39.7 %	50 %
STORAGE**	34.1 %	50 %
LOCATION**	38.9 %	25 %
PHONE_CALLS	32.5 %	35 %
PERSONAL_INFO	18.4 %	13 %
HARDWARE_CONTROLS	12.5 %	17 %
COST_MONEY	10.6 %	9 %
MESSAGES	3.7 %	5 %
ACCOUNTS	2.6 %	2 %
DEVELOPMENT_TOOLS	0.35 %	0 %

(b) The most frequent Dangerous permissions and their categories.

Permission (Category)	Free	Paid
INTERNET** (NETWORK)	86.6 %	65 %
WRITE_EXTERNAL_STORAGE** (STORAGE)	34.1 %	50 %
ACCESS_COARSE_LOCATION** (LOCATION)	33.4 %	20 %
READ_PHONE_STATE (PHONE_CALLS)	32.1 %	35 %
WAKE_LOCK** (SYSTEM_TOOLS)	24.2 %	40 %
ACCESS_FINE_LOCATION (LOCATION)	23.4 %	24 %
READ_CONTACTS (PERSONAL_INFO)	16.1 %	11 %
WRITE_SETTINGS (SYSTEM_TOOLS)	13.4 %	18 %
GET_TASKS* (SYSTEM_TOOLS)	4.4 %	11 %

Table 2: Survey of 856 free and 100 paid Android applications. We indicate significant difference between the free and paid applications at 1% (**) and 5% (*) significance levels.

As far as we are aware, none of the paid applications in our data set are signed or distributed by device manufacturers. Three of the paid applications request Signature permissions, and five request SignatureOrSystem permissions. Of the free applications, 25 request Signature permissions, 30 request SignatureOrSystem permissions, and four request both. We have found four of the aforementioned free applications pre-installed on phones; the remainder will not receive the permissions on a typical device. Requests for unobtainable permissions may be developer error or leftover from testing.

3.2.3 Evaluation

User Consent. Nearly all applications (93% of free and 82% of paid) ask for at least one Dangerous permission, which indicates that users are accustomed to installing applications with Dangerous permissions. The INTERNET permission is so widely requested that users cannot consider its warning anomalous. Security guidelines or anti-virus programs that warn against installing applications with access to both the Internet and personal information are likely to fail because almost all applications with personal information also have INTERNET.

Several important categories are requested relatively infrequently, which is a positive finding. Permissions in the PERSONAL_INFO and COST_MONEY categories are only requested by a fifth and a tenth of applications, respectively. The PERSONAL_INFO category includes permissions associated with the user's contacts, calendar, etc.; COST_MONEY permissions let applications send text messages or make phone calls without user confirmation[4]. Users have reason to be suspicious of applications that ask for permissions in these categories. However, users may not notice these rare warnings because the overall rate is so high.

[4]The separate PHONE_CALLS category contains permissions that modify telephony state but do not cost the user money.

Defense in Depth. Given the prevalence of Dangerous permissions, an application vulnerability is likely to occur in an application with at least one Dangerous permission. However, the average Android application is much less privileged than a traditional operating system program. Every desktop Windows application has full privileges, whereas no Android application in our set requests more than half of the available Dangerous permissions. A majority of the Android applications ask for less than seven, and only 10% have access to functionality that costs the user money. This is a significant improvement over the traditional full-privilege, user-based approach.

Review Triaging. A hypothetical review process could exempt applications that do not have Dangerous permissions. Unfortunately, this alone would not reduce reviewer workload much. Only 18% of paid and 7% of free applications would be exempt from review. To improve this, a review process could also exclude applications whose only Dangerous permission is INTERNET. An application with only the INTERNET permission cannot leak sensitive personal information because reading user data requires a second permission. This would increase the number of exempt applications to 22% of paid and 21% of free applications.

4 Reducing Application Privileges

Our application survey indicates that up-front permission declarations can promote defense in depth security and provide moderate review triaging advantages. However, a large number of applications still ask for dangerous permissions. Decreasing the number of privileges that applications require to function will improve the utility of permissions. We investigate factors that influence permission requirements and present corresponding suggestions for reducing the frequency of permission usage.

4.1 Developer Incentives

Developer incentives can encourage or discourage permission requests. Current incentives include the length of the review process, how the automatic update system treats additional permissions, and pressure from users.

Review Process. Formal review can delay an application's entrance into the directory. Developers are often concerned about the length of the review process [13]. If dangerous permissions increase the review time (and a lack of dangerous permissions decreases it), then developers have an incentive to use as few permissions as necessary. Google Chrome extensions have to undergo a review process if they include plug-ins, which incentivizes developers to not use plug-ins. Other platforms could adopt similar review systems or institute a timed delay for applications with more permissions.

Pressure From Users. The ultimate goal of a developer is to reach as many users as possible. If users are hesitant to install applications with certain permissions, then developers are motivated to avoid those permissions. Users can express their dislike of permission requests in application comments and e-mails to the developer.

We read the user comments for 50 randomly selected Google Chrome extensions with at least one permission. Of the 50 extensions, 8 (15%) have at least one comment questioning the extension's use of permissions. The percentage of comments pertaining to permissions ranges widely, from 1 of 2 to 5 of 984. A majority of the permission comments refer to the extension's ability to access browsing history. Several commenters state that the permission requests are preventing them from installing an application, e.g., "Really would like to give it a try. ... But why does it need access to my history? I hope you got a plausible answer because I really would like to try it out." These comments indicate that a small number of users are pressuring developers to use fewer permissions.

Additionally, developers of 3 of the 50 extensions descriptions include an explanation of their permission usage. This indicates that these developers are concerned about user reactions to permission requests.

Automatic Updates. Android and Google Chrome automatically update applications as they become available, according to user preferences. However, automatic updates do not proceed for applications whose updates request more permissions. Instead, the user needs to manually install the update and approve the new permissions; in Android, this amounts to several additional screens. This incentivizes developers to request unnecessary permissions in in case later versions require the permissions. If update UIs were improved to minimize the user effort required to update applications with new permissions, this disincentive might be eliminated.

4.2 Developer Error

Developers may ask for unnecessary permissions due to confusion or forgetfulness. We explore the prevalence of developer error. Tools that help developers select correct permissions could reduce application privileges without requiring any changes to the permission system itself.

4.2.1 Errors in Google Chrome Extensions

Browser Managers. We count the extensions that request browser managers but do not use them. About half of the extensions in our set of 1000 "popular" extensions request access to security-relevant browser managers. We search their source code (including remotely sourced scripts) for references to their requested browser managers. 14.7% of the 1000 extensions are overprivileged by this measure because they request access to managers that they never use. It is possible for an extension to name a browser manager without explicitly including the name as a string (e.g., `"book"+"marks"`); we examined a random sample of 15 overprivileged extensions and found no evidence of developers doing this.

Domains. We also review fifty randomly selected extensions for excessive domain access (see Appendix A). For each extension, we compare the permissions it requests with the domains needed to implement its functionality, which we determine by manually exercising the user interface and consulting its source code when necessary. We find that 41 of the 50 extensions request access to web data, and 7 of those are overprivileged: 5 request too many domain permissions for their core extensions, and 2 install content scripts on unnecessary domains.

The reasons for overprivilege are diverse. One example is "PBTweet+", which requests web access for a nonexistent core extension; other examples are "iBood" and "Castle Age Autoplayer", which request access to all domains even though they only interact with iBOOD and Facebook, respectively.

"Send using Gmail (no button)" demonstrates a common error, which is that developers sometimes request access to all and specific domains in the same list. We find that an additional 27 of the 1000 popularity-ranked extensions also make this mistake. This is a conservative measure of wildcard-induced error; subdomain wildcards can feature the same mistake, like asking for both `http://www.example.com` and `http://*.example.com`.

4.2.2 Errors in Android Applications

We manually review the top free and top paid application from eighteen Android Market categories (see Appendix A for a list). For each of the applications, we compare its functionality to the permissions it requests. To determine an application's functionality requirements, we exercise the user interface. Android's permission documentation is incomplete; when we were unable to determine whether functionality requires permissions, we conservatively assumed it does.

Of the 36 applications, 4 are overprivileged. Unnecessary `INTERNET` permissions account for three of the overprivileged applications. One of the developers may have done this with the mistaken belief that launching the browser requires the `INTERNET` permission, since that is how the application interacts with the Internet. The fourth overprivileged application requests `ACCESS_FINE_LOCATION` unnecessarily.

In addition to the four overprivileged applications, another four could re-implement the same functionality without the `INTERNET` permission. For example, "DocsToGo" provides the ability to update the application over the Internet even though that functionality is already provided by the Android Market, and "Jesus Hates Zombies" could store its small set of static resources locally.

4.2.3 Tools for Error Reduction

As far as we are aware, none of the prominent platforms with install-time permissions provide developer tools to detect unnecessary permissions. We recommend that future platforms provide developers with tools to guide the writing of permission declarations. Such a tool could help reduce privileges by aiding developers in correct permission selection. The tool could run whenever an application is submitted to the directory, or it could be provided to developers as part of the development or packaging process. If unnecessary permissions are found, the developer could be prompted to remove them.

Our Google Chrome extension overprivilege detection tool is simple but sufficient to find some types of errors. As shown in Section 4.2.1, a JavaScript text search is sufficient to remove unnecessary browser manager permissions from 147 of the 1000 popularity-ranked extensions. Our text search has a small number of false positives; e.g., we found three extensions that only contain references to browser managers in remotely sourced scripts. However, a developer can disregard a warning if she feels it is incorrect. Our tool also detects simple redundant wildcard errors and asks the developer to remove the broad wildcard in favor of the more specific domain. Detecting the larger problem of overly broad domain requests is a challenging open problem for future research in JavaScript program analysis.

A similar Android tool could analyze applications to find all Android API calls, and from that deduce what permissions the applications need. The tool could ask the developer to discard permissions that are not required by any of the API calls. The tool cannot completely replace developers; developers must still edit their permission requirements if they want to include additional permissions for inter-application interactions. (Applications can choose to only accept messages from applications with certain permissions.) Unfortunately, incomplete documentation currently prevents this tool from being built; the documentation does not completely state which API calls require which permissions. Experimentally determining the permission-API relationship is an active area of future research.

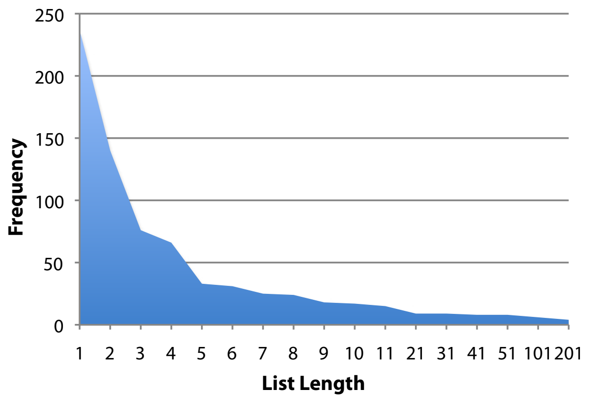


Figure 5: The number of content script specific domain lists with *at least* a given length. Note the non-linear x-axis.

4.3 Wildcards

Domain access in the Google Chrome extension system relies on wildcards. A developer can write `<all_urls>` or `*://*/*` and gain access to all domains, or she can define a list of specific domains. When it is not feasible for a developer to list all possible subdomains, she can use wildcards to capture multiple subdomains. However, a developer might choose to use a wildcard even though it includes more privileges than the application requires.

Compliance. To determine whether developers are willing to write specific domain lists when they can more easily request access to all domains, we evaluate the prevalence of specific domain lists in the 1000 popularity-ranked extensions. Of the 714 extensions that need access to web data, 428 use a specific domain list for either a content script or core extension. This is a surprising and positive finding: 60% of developers whose extensions need web access choose to opt in to domain restrictions for at least one component. However, 367 extensions also have at least one component that requests full domain access. (An extension with multiple content scripts might request full domain access for some scripts but place restrictions on others.)

Developer Effort. We suspect that developers will default to requesting all-domain access if the number of specific domains in the list grows too high. To examine this further, we consider the 237 content scripts that use specific domain lists. The lists are short: only 31 are longer than five. Figure 5 presents the distribution. This indicates that most developers either request a very small number of domains or opt to request full domain access, with few in-between. However, six developers wrote eight lists that are longer than fifty domains. These outliers result from developers internationalizing their extensions by repeating the same domains with different suffixes; wildcards cannot be used to represent suffixes because the domains may have different owners.

Noncompliance. Section 4.2 describes a manual analysis of fifty extensions. Five of those extensions are overprivileged due to improper wildcard use. Two of the developers choose to request all-domain access rather than write specific domain lists, two write specific domain lists but unnecessarily use wildcards for subdomains, and one incorrectly requests all-domain access alongside specific domains. In other words, 10% of the extensions with web access request excessive permissions because their developers are unable or unwilling to write sufficiently specific domain lists.

In summary, our findings are twofold. We show that 60% of extension developers write at least one specific domain list. This demonstrates that the option to write specific domain lists is a worthwhile part of a declarative permission system. On the other hand, 40% of developers whose extensions need web access do not write any specific domain lists. Furthermore, our manual analysis indicates that 10% of extensions with web access use wildcards improperly.

4.4 Permission Granularity

If a single permission protects a diverse set of API calls, then an application seeking to use only a subset of that functionality will be overprivileged. Separating a coarse permission into multiple permissions can improve the correlation between permissions and application requirements. On the other hand, excessively fine-grained permissions would burden developers with a large list of permissions required to perform simple actions.

4.4.1 Google Chrome Browser Managers

At the time of our study, Google Chrome extension permissions were at the granularity of a browser manager: one permission per entire browser manager. This posed a problem for the window manager, which includes some methods that provide indirect access to history via the location property of loaded windows. Using the window manager generated history warnings, regardless of whether the extension used any of the methods that provide access to the location property.

The fact that the window manager caused a history warning was confusing to users and developers. Consider this quote from the developer of *Neat Bookmarks*:

> Installing this extension will ask for permission to access your browsing history, which is totally useless, not used and not stored by the extension at all. Not really sure why 'History' is part of 'Bookmarks' in the Chrome browser.

The developer is so confused by the history warning that he or she believes it is caused by the extension's use of the bookmark manager, rather than the window manager.

Since the time of our study, the window manager has been changed so that certain methods do not require any permission. Consequently, developers can access some non-history-related functionality without acquiring a permission that shows users the history warning.

4.4.2 Fine-Grained Android Permissions

We evaluate whether Android's fine-grained permissions are an improvement over a coarser-grained alternative.

Categories. Android permission categories are high-level functionality groups. Categories are comprised of multiple permissions, which developers must request individually. A coarse-grained permission system might simply have one permission per category, but Android subdivides each category into multiple finer-grained permissions. We find that no application (out of 956) requires all of the permissions in any category except `STORAGE`, a category with only one permission. This demonstrates that coarse-grained permissions at the category level would overprivilege all extensions.

Read/Write. Android controls access to data with separate read and write permissions. For example, access to contacts is governed by `READ_CONTACTS` and `WRITE_CONTACTS`. We find that 149 applications request one of the contacts permissions, but none requests both. 10 of 19 applications with calendar access request both read and write permissions. Text messages are controlled by three primary permissions; only 6 of the 53 applications with text message permissions request all three. These results demonstrate that separate read and write permissions reflect application requirements better than coalesced permissions would.

Location. Location is separated into "fine" and "coarse" permissions, referring to the precision of the location measurement. `ACCESS_FINE_LOCATION` governs GPS location, and cell location is controlled by ACCESS_COARSE_LOCATION. 358 applications request at least one of the location permissions; 133 request only ACCESS_COARSE_LOCATION. This indicates that 37% of applications that need to know the user's location are satisfied with a "coarse" location metric, which benefits user privacy.

Future permission systems should consider adopting similar fine-grained permissions.

4.4.3 Coarse-Grained Android Permissions

Not all of Android's permissions are fine-grained. The INTERNET permission lets an application send HTTP(S) requests to all domains, load any web site into an embedded browser window ("WebView"), and connect to arbitrary destinations and ports. The granularity of the INTERNET permission is important because 86.6% of free and 65% of paid applications in our large-scale study use it.

We find that 27 of the 36 Android applications in our manual review (Section 4.2.2) have the INTERNET permission. Of those, 13 only use the Internet to make HTTP(S) requests to specific domains. These Android applications rely on backend servers for content, much like web applications. A fourteenth application additionally uses the INTERNET permission to support Google AdSense, which displays advertisements from a single domain in a WebView.

These results indicate that many applications would tolerate a limited Internet permission that only permits HTTP(S) or WebView access to a specific list of domains, similar to what Google Chrome offers extensions. This hypothetical limited permission would be sufficient for 52% of the 27 applications that use INTERNET.

5 Reducing User Prompts

Our study in Section 3 demonstrates that almost all extensions and applications trigger prompts for dangerous permissions during installation. The high rate of permission warnings makes it unlikely that even an alert, security-conscious user would pay special attention to an application with several dangerous privileges.

Possible solutions to this problem depend on the intended role of permission prompts. If permission prompts are only intended to inform the user and decrease platform liability, then perhaps their presentation and frequency do not matter. If a prompt is supposed to warn or alert the user, however, then increasing user attention will improve its efficacy. In order to preserve the significance of truly important warnings, one possibility is to de-emphasize or remove lesser warnings.

5.1 Google Chrome

Google Chrome currently presents all permissions equally. Critical extension privileges (e.g., including a plug-in) should always be prominently displayed as part of the installation process, but less significant permissions (e.g., bookmarks) could be omitted from the installation warning and simply listed on the download page.

Not all Internet access needs to be displayed to users. Web sites with private information (e.g., financial, commercial, and e-mail sites) use TLS to protect users from man-in-the-middle attacks. We assume that HTTP-only sites are not concerned about eavesdropping. If Google Chrome were to only show warnings for extensions with access to HTTPS sites, 148 of the 500 most popular extensions would no longer trigger web access warnings. 102 extensions would no longer prompt a warning at all, reducing the number of extensions with at least one warning from 91.4% to 71% of the 500 most popular extensions. Users would be at risk of man-in-the-middle attacks on HTTP-only sites, but they already are at risk of this on their networks.

5.2 Android

Android ranks permissions by threat level, and only Dangerous permissions are displayed to users. However, there is still great variance within Dangerous permissions. Dangerous permissions let an application perform actions that cost the user money (e.g., send text messages), pertain to private information (e.g., location, contacts, and the calendar), and eavesdrop on phone calls. On the other hand, Dangerous permissions also guard the ability to connect to paired Bluetooth devices, modify audio settings, and get the list of currently running applications. Users may not care about Dangerous permissions that cannot cause direct harm to the user or phone. De-emphasizing the less-threatening Dangerous permissions could reduce the number of user warnings.

WAKE_LOCK and WRITE_EXTERNAL_STORAGE are two of the most popular Dangerous permissions, and neither has a clear implication for users. The WAKE_LOCK permission lets an application perform actions that keep the phone awake without user interaction. Playing music, for example, requires this permission. Although the permission could be used to slowly drain the battery, it does not pose a serious privacy or security threat. 26% of the 956 applications have the WAKE_LOCK permission. The WRITE_EXTERNAL_STORAGE permission controls access to the SD card, which could be used to access other applications' files that are on the SD card. However, the user has no way of differentiating between legitimate and illegitimate access to the SD card. It seems reasonable for all applications to store data, and only the developer knows whether to use internal or external storage. 35.7% of the 956 applications have this Dangerous permission.

INTERNET is the most popular permission. The higher prevalence of the INTERNET permission in free applications and past work [5] indicate that free applications commonly use the Internet to contact advertisers. Section 4.4.3 suggests enabling applications to request access to a specific list of web domains. Accordingly, the Android Market could display a less severe warning for applications with limited Internet access than for applications with the full INTERNET. The warning could further notify the user if a known advertising domain is included in the specific domain list.

6 Related Work

Google Chrome Extensions. When Barth et al. introduced the Google Chrome extension permission system, they conducted a motivating analysis of 25 Google Chrome extensions [2]. However, their sample set is too limited to be definitive. Google employees authored 9 of the 25 extensions, and the extension platform had only been public for a few weeks prior to their study. The results of our large-scale evaluation of Google Chrome extensions show that their small-scale study overestimated the prevalence of extension privileges. Guha et al. [8] performed a concurrent, short study of the permissions used by Google Chrome extensions, although they do not study the effect of popularity. We provide a significantly more detailed discussion of extension privileges.

Android Applications. Barrera et al. [1] analyze the permissions requested by 1,100 free Android applications. They primarily focus on the structure of the permission system; they group applications together using a neural network and look for patterns in permission group requests. They note that 62% of the applications collected in December 2009 use the INTERNET permission. Significantly more applications in our data set use the INTERNET permission, which is possibly due to changes in applications over time. We also provide data that can be used to evaluate two of their proposals for changes to Android permissions. First, they suggest that applications should be able to simultaneously request multiple permissions with wildcards (e.g., android.permission.SMS.*). Our Google Chrome survey shows that developers often use wildcards to request excessive privileges, and our Android study shows that the majority of applications do not need access to all permissions in a group. Next, they propose that the INTERNET permission should support specific domain lists. A manual review finds that 14 of 27 applications with the INTERNET permission would indeed be satisfied with access to a list of specific domains.

Researchers at SMobile present a survey of the permissions requested by 48,694 Android applications [18]. They do not state whether their sample set is composed of free applications, paid applications, or a combination. They report that 68% of the applications in their sample set request enough permissions to be considered "suspicious." We similarly find that applications have high privilege requests. They also report with alarm that 9 applications request access to the BRICK permission, which can be used to make a phone non-operational. However, this is a Signature permission; it is only available to a very small number of applications signed by the device manufacturer. We find that a surprising number of applications request Signature and SignatureOrSystem permissions, given that most applications are unable to actually use these permissions.

Kirin [6] is a tool that evaluates the security of an Android application. It compares the application's requested permissions to a set of permission rules. They propose several rules and test them against 311 applications. Their rules are specific enough to only flag a small number of the applications in our set, but we did not check to see whether the applications are malicious.

User Warnings. We consider whether installation warnings are of value to security-conscious users. Other researchers have examined the best way to visually display installation permissions to users [17] but not examined the frequency of prompts in install-time permission systems. Warning science literature indicates that frequent exposure to specific warnings, especially if the warnings do not lead to negative consequences, drastically reduce the warnings' effectiveness [11, 15]. Other researchers have shown that browser warnings for phishing sites and invalid SSL certificates are ignored by most users [4, 16]; it is possible that even infrequent permission installation warnings will be ignored.

LUA. Windows users can reduce application privileges by running Windows as a low-privileged user account (LUA). While in LUA mode, all applications have reduced privileges. When an application wants to perform a task that requires administrative privileges, Windows presents the user with a prompt for approval. Unlike the application permission model discussed in this paper, only two security modes are available (user or administrative). Furthermore, in practice, users run in administrative mode all the time, thereby granting the system's full privileges to applications [12].

7 Conclusion

This study contributes evidence in support of application permission systems. Our large-scale analysis of Google Chrome extensions and Android applications finds that real applications ask for significantly fewer than the maximum set of permissions. Only 14 of 1000 Google Chrome extensions use native code, which is the most dangerous privileges. Approximately 30% of extension developers restrict their extensions' web access to a small set of domains. All Android applications ask for less than half of the available set of 56 Dangerous permissions, and a majority request less than 4.

These findings indicate that permission systems with up-front permission declarations have two advantages over the traditional user permission model: the impact of a potential third-party vulnerability is greatly reduced when compared to a full-privilege system, and a number of applications could be eligible for expedited review. These results can be extended to time-of-use permission systems if the system requires developers to declare a set of maximum permissions.

However, our study shows that users are frequently presented with requests for dangerous permissions during application installation in install-time systems. As a consequence, installation security warnings may not be an effective malware prevention tool, even for alert users. Future work should identify which permission warnings are useful to users and consider alternate methods of presenting permissions to users.

References

[1] Barrera, D., Kayacik, H. G., van Oorschot, P. C., and Somayaji, A. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *ACM CCS* (2010).

[2] Barth, A., Felt, A. P., Saxena, P., and Boodman, A. Protecting Browsers from Extension Vulnerabilities. In *NDSS* (2010).

[3] Cluley, G. Windows Mobile Terdial Trojan makes expensive phone calls. http://www.sophos.com/ blogs/gc/g/2010/04/10/windows-mobile-terdial-trojan-expensive-phone-calls/.

[4] Egelman, S., Cranor, L. F., and Hong, J. You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings. In *CHI* (2008).

[5] Enck, W., Gilbert, P., Chun, B., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI* (2010).

[6] Enck, W., Ongtang, M., and McDaniel, P. D. On Lightweight Mobile Phone Application Certification. In *ACM CCS* (2009).

[7] Felt, A. P. Issue 54006: Security: Extension history permission does not generate a warning. http://code.google.com/p/chromium/issues/ detail?id=54006, August 2010.

[8] Guha, A., Fredrikson, M., Livshits, B., and Swamy, N. Verified Security for Browser Extensions. In *IEEE Security and Privacy* (2011).

[9] Ibrahim, S. Universal 1-Click Root App for Android Devices. http://androidspin.com/2010/08/10/ universal-1-click-root-app-for-android-devices/, August 2010.

[10] Liverani, R. S., and Freeman, N. Abusing Firefox Extensions. Defcon17, July 2009.

[11] Magat, W., Viscusi, W. K., and Huber, J. Consumer processing of hazard warning information. *Journal of Risk and Uncertainty* (1988).

[12] Motiee, S., Hawkey, K., and Beznosov, K. Do Windows Users Follow the Principle of Least Privilege? Investigating User Account Control Practices. In *SOUPS* (2010).

[13] Mozilla Add-ons Blog. The Add-on Review Process and You. http://blog.mozilla.com/addons/2010/02/ 15/the-add-on-review-process-and-you.

[14] Seriot, N. iPhone Privacy. *Black Hat DC* (2010).

[15] Stewart, D. W., and Martin, I. M. Intended and Unintended Consequences of Warning Messages: A Review and Synthesis of Empirical Research. *Journal of Public Policy Marketing 13*, 1 (1994).

[16] Sunshine, J., Egelman, S., Almuhimedi, H., Atri, N., and Cranor, L. F. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *USENIX Security Symposium* (2009).

[17] Tam, J., Reeder, R. W., and Schechter, S. I'm Allowing What? Disclosing the authority applications demand of users as a condition of installation. Tech. Rep. MSR-TR-2010-54, Microsoft Research, 2010.

[18] Vennon, T., and Stroop, D. Threat Analysis of the Android Market. Tech. rep., SMobile Systems, 2010.

[19] Willison, S. Understanding the Greasemonkey vulnerability. http://simonwillison.net/2005/Jul/ 20/vulnerability/.

A Manual Review

Android Applications. Jesus Hates Zombies, Compass, Aquarium Live Wallpaper, Movies, Mobile Banking, Calorie Counter by FatSecret, Daily Horoscope, Pandora Radio, The Weather Channel, Advanced Task Killer, Google Sky Map, Barcode Scanner, Facebook for Android, NFL Mobile Aquarium, Live Wallpaper, weird facts, Google Maps Screen Crack, screen krack, twidroyd for twitter, touch to talk, open home, pageonce pro, personal finance, baby esp, gentle alarm, picsay pro, beautiful widgets, iQuran Pro, Grocery King, Touitor Premium, MLB.com at Bat 2010, myBackupPro, London Journey, BeyondPod Unlock Key, Text to Speech Extended, DocumentsToGo Full

Google Chrome Extensions. Orkut Chrome Extension, Google Similar Pages beta (by Google), Proxy Switchy!, AutoPager Chrome, Send using Gmail (no button), Blog this! (by Google), Fbsof, Diigo Web Highlighter and Bookmark, Woot!, Pendule, Inline Search & Look Up, YouTube Middle-Click Extension, Send to Google Docs, [Non-English Title], PBTweet+, Search Center, Yahoo Mail Widget for Google Chrome, Google Reader Compact, Chromed Movilnet, Ubuntu light-themes scrollbars, Persian Jalali Calender, Intersect, deviantART Message Notifier, Expand, Castle Age Autoplayer Alpha Patched, Patr Pats Flickr App, Better HN, Mark the visited links, Chrome Realtime Search, Gtalk, SpeedyLinks, Slick RSS, Yahoo Avatar, Demotivation.ru ads remover, [Non-English Title], PPTSearch Edu Sites, Page2RSS, Good Habits, VeryDou, Wikidot Extender, Close Left, iBood, Facebook Colored, eBay Espana (eBay.es) Busqueda avanzada, Keep Last Two Tabs, Google Transliteration Service, Ohio State University Library Proxy Extension, Add to Google Calendar, Rocky, Short Youtube

Experiences on a Design Approach for Interactive Web Applications

Janne Kuuskeri
Department of Software Systems
Tampere University of Technology
Korkeakoulunkatu 1, FI-33720 Tampere, Finland
janne.kuuskeri@tut.fi

Abstract

Highly interactive web applications that offer a lot of functionality are increasingly replacing their desktop counterparts. However, the browser and the web itself were originally designed for viewing and exchanging documents and data and not for running applications. Over the years, web pages have slowly been transformed into web applications and as a result, they have been forcefully fit into an unnatural mold for them. In this paper we present a pattern for implementing web applications in a way that completes this transition and creates a more natural environment for web applications to live in. In the pattern, the full MVC stack is implemented in the client while the server is completely decoupled via a RESTful interface. We also present experiences in building an industrial-scale application utilizing this pattern.

1 Introduction

Over the recent years, web has become the most important platform for delivering and running applications. With the ubiquitous web, these applications are easily accessible regardless of place and time and without any installation requirements, end users have begun to favor web applications over traditional desktop applications. On the other hand, the rapidly increasing mobile application market has proven that web applications are not always loaded on demand anymore but they are also being installed on devices.

The fast growth of the web as an application platform has raised the standard for its inhabitants. Rich and dynamic user interfaces with realtime collaborative features have now become the norm. Moreover, with the popularity of social networks, applications are expected to link or embed information from other web applications. Finally, if the end user is not immediately satisfied with the web site, she will simply enter a new URL and start using another similar service. This has lead application developers to push the limits of the browser and the web standards.

Unfortunately, browsers and the standards of the web have not quite been able to keep up with the pace. Therefore, application developers have been forced to work around the standards and to do whatever it takes to meet end users' demands. The application logic and the presentation have gotten mixed in the mayhem of HTML, CSS, JavaScript and, some server-side scripting system, say PHP. Within this technology rush, good software development practices and patterns have been somewhat forgotten. Without extra attention, code base gradually loses clear separation of concerns and the responsibilities of different components become unclear [11]. For these reasons many web applications have become difficult to scale or even maintain.

In this paper, we describe a pattern for building web applications for scalability both in terms of throughput and in terms of provided functionality. The suggested pattern advises breaking the application in two autonomous parts: a RESTful web service and a single page web application that utilizes it. Following this pattern will contribute to having a clean and consistent design throughout the application, thereby making the end result easier to test and maintain in general. As a side effect, the versatility of the application is greatly increased by offering easier adoption for clients other than browsers.

Neither RESTful interfaces nor single page web applications are new ideas by themselves. However, in this paper we suggest using the RESTful API directly from the JavaScript application using Ajax requests. The main contributions of this paper are firstly, to describe the pattern for implementing complex web applications and secondly, to present experiences from utilizing this pattern in a real world, large scale application. This application is currently in production, and experiences listed in the paper are based on actual feedback. By providing experiences and comparison we show the benefits of the suggested pattern.

The rest of the paper is structured as follows. In Section 2 we examine the main components of traditional web applications and identify their weaknesses. Next, in Section 3, we introduce our pattern for building complex web applications. In Section 4 we present how to apply the pattern in a real world application and give insight as to what kind of design the application has. Section

5 discusses the pros and cons of the suggested approach and Section 7 provides a review of future work. Section 8 concludes the paper with some final remarks.

2 Traditional Way of Building Web Applications

In this section, we illustrate the procedure that is endorsed by most of the popular frameworks for building and running web applications. We briefly describe each relevant component of a web application from the viewpoint of this article. This is done so that we are able to review our approach in later sections and to show how it differs from the approach presented here.

2.1 Building Blocks of Web Applications

Today, numerous web frameworks are based on the MVC [9] pattern. Some frameworks call it something else and many of them interpret it a bit differently but at the high level, there are usually three main components that comprise a web application: the *model*, the *view* and the *controller*. Not all applications follow this structure rigorously but usually these three components are identifiable in one form or another. Sometimes application logic is separated into its own module, sometimes it is part of one of the other modules, and sometimes it is scattered over multiple modules. In any case, it provides a *good enough* abstraction for us to describe the components of web applications, and their supposed responsibilities.

2.1.1 The View

In this section, by *view*, we refer to what the browser shows and executes. The main components of the view are defined below.

HTML – Traditionally, the hierarchy and overall layout of a web page is composed using HTML. In the browser the HTML becomes part of the DOM tree of the page. The DOM is what ultimately defines the hierarchy of the page. The DOM is dynamic and may be altered at runtime, thereby making web pages dynamic.

CSS – Cascading Style Sheets are used to define what the page will look like after it has been rendered on the screen. If no CSS rules are provided, the browser will use a set of default rules. CSS has little to do with the dynamicity of the page; pages with no CSS rules can still be dynamic and provide client side functionality.

JavaScript – In order to create a web page that has client side functionality, there must be a programming language to implement that logic. If we leave out the option of using any custom browser plugins, JavaScript is the only choice for implementing any application logic within the page.

2.1.2 The Controller

In web applications, the role of the controller varies the most. What is common for all the different interpretations however, is that controller is the component that takes in all the HTTP requests coming in from the browser. Many times the controller takes the role of the dispatcher and forwards the request to appropriate handler in the model and afterwards it finds the correct view that is associated with the request. When the request has been processed, controller functions return HTTP response back to the browser. In some applications there is a single centralized controller that handles all requests while in others there is a controller function bound to each URL that the web application exposes.

2.1.3 The Model

The model component refers to the database layer of the application. This does not need to be an actual database, but commonly web applications have some kind of persistent storage, where objects of the application logic are stored and retrieved. Many times an *Object Relational Mapper* (ORM) such as Hibernate, SqlAlchemy or ActiveRecord is used to implement the mapping between the domain objects and the persistent storage. Sometimes the model component is further divided into the business logic layer and the data access layer.

2.2 Burden of Building Web Applications

As suggested by section 2.1 a typical web application consists of many different types of resources. These include (but are not limited to) HTML templates, CSS files, JavaScript files, image files and the source code for the server side implementation. The application logic is usually implemented mainly by the server side but depending on the application a fair bit of application logic may also be in the JavaScript files and even inside HTML files, which usually incorporate some kind of templating language. CSS and image files on the other hand are purely static resources and only affect the appearance of the application.

When the application grows, a careful design needs to be in place, not only for the application logic, but also for the directory hierarchy and the responsibilities; which part of the application should be responsible of which functionality. Therefore, when building web applications with a lot of functionality and dynamic features, it has become a common practice to use a set of existing tools and frameworks to make the development easier. Frameworks such as *Spring* [5], *Django* [1] and *Ruby on Rails* [3] do a good job at making it easier for the developers to implement and maintain their projects. They even provide guidance on how to assign responsibilities for different components but these guidelines are not enforced in any way and in the end it is up to the developer to figure out what works best for a particular application.

This kind of approach to building web applications also easily results in tightly coupled systems. Not only do the server and the client become dependent on each other but there is also coupling between different components or technologies within the application. For example, the server usually sends a complete HTML page to the browser but on some occasions it may only send parts of the page and rely on the client side JavaScript to fetch the rest of the data (e.g. XML) using Ajax requests. Also the application flow has to be agreed in advance between the client and the server. As a result it would be very difficult to implement a completely different kind of user interface for the application without making any changes to server side components.

2.3 Burden of Running Web Applications

By its very nature, HTTP is a stateless protocol. Therefore, each request occurs in complete isolation from any other request. When user clicks on a link on a web page and another page within the same application is presented, the browser has to load that new page completely from the server and forget everything it knew about the previous page. Thus, from the client's perspective the application is restarted each time a page is loaded. However, many applications require their internal state to be maintained while the user is running the application and new pages are loaded. That is why the client side of the application depends on the server to maintain the application state between page loads.

This kind of behavior is usually implemented as a server side session. Each time the browser sends an HTTP request, it sends a cookie with the request. Within the cookie there is a unique identifier that identifies the client's session and the server then has to interpret all the data in the session to see where – in the flow of the application – the client was. Now the server can resume the application state and continue into processing the request. In the following, we list common steps that the server has to take when a page is requested:

1. Resume the application state by processing the received cookie and recovering the associated session.
2. Invoke appropriate controller function to execute the application logic.
3. Use the model to retrive and update associated persistent data.
4. Locate correct view and populate it with the data from the model.
5. Update the session to reflect the new state of the client's application.
6. Return the populated HTML page to the browser.

It should be noted, that we have purposefully left out things that are not relevant in the scope of this paper.

When the browser receives the newly composed HTML page it has to parse and render it on the screen. This includes fetching all the related resources that the page uses, in essence, CSS, JavaScript and image files. While some of these resources may come from the cache, some of them do not. This laborious sequence of events takes place every time the user performs a function that requires a new page to be loaded. There has been a lot of research on how to make web sites faster ([16, 17]) and there are dozens of tricks that developers need to know and implement in order to minimize the overhead and latency during page loading.

3 The Partitioned Way of Building Web Applications

Given the complexity of building, running and maintaining web applications with a lot of functionality we argue that these applications should be implemented in a simpler way. This simplicity can be achieved by breaking the application in parts and strictly defining their responsibilities. In the following sections we describe this process in more detail.

3.1 Breaking the Application in Two

Based on the fact that the web builds on top of HTTP, applications using it as their platform are distributed over the network. Furthermore, as HTTP is a resource oriented and stateless protocol, we argue that web applications should be broken into services provided by the server and the client that uses these services to compose a meaningful web application. From this separation we draw the following set of rules that web applications should adhere to:

1. Application state is stored in the application, not in the server.
2. Client application is a self contained entity built on top of services provided by single or multiple sites.
3. Services that are used for implementing the application do not make any assumptions about the clients using them.

The motivation and consequences of the rules are the following:

1) As already mentioned, from the client's perspective the application is restarted each time a new page is loaded. Effectively this means that either the application should be implemented as a so called *single page web application* or it explicitly stores its state using services provided by the server.

2) Making the client application a self contained entity makes the responsibilities explicit and unambiguous: the server decides which services it exposes and

the client is free to use them how it chooses. Moreover, clients become applications in their own right; they are run (and possibly even started) completely inside the browser, while using the services of the server only when they need to. This approach is also utilized by many of today's HTML5 mobile applications.

3) When the server side of the application becomes agnostic about its clients, it completely decouples the server and the client. They can be developed and tested independently. Furthermore, it allows any type of client (not just browsers) to use the service.

Following this pattern, makes the browser a platform for running JavaScript powered applications and the server a datastore responsible for maintaining application's persistent data. A rough analogy to the world of traditional desktop applications is that the browser now becomes the operating system where applications are run and the services provided by the server become the remote database for the application. The main difference being that remote services may actually contain complex operations as opposed to being a simple data store.

3.2 Services

In the scope of this paper we are only interested in the interface of a web service, not its implementation. It is indeed the interface and its properties that enable clients to utilize it into building fully working applications. To support wide variety of clients and functionality, the interface should be as accessible and as general as possible. Accessibility comes from adhering to widely accepted standards and picking a technology that is supported by most of the clients. Choosing the right level of generality however, can be difficult. The interface must cover all the requirements laid out for the applications using it and at the same time, to support scalability, it should not be needlessly restricted and specific to use cases it implements.

By choosing the RESTful architectural style [14], the service interface is able to support any client that implements HTTP protocol. Given that web applications already use the HTTP protocol, this is not really a restriction. In the following we describe how RESTful interface is able to fulfill all the requirements presented earlier.

Accessibility – The RESTful architectural style adheres to the HTTP/1.1 protocol [8], which is supported by virtually all clients using the web as their application platform. Furthermore, REST does not mandate any format for the transferred data. The representations of the resources exposed by the RESTful interface may use any format or even support multiple formats. This makes RESTful interfaces even more accessible: JavaScript clients usually prefer JSON format, while some other client may prefer XML.

Application State – REST makes very clear distinction between the application state and the state of the resources. In REST, the server is only responsible for storing the states of its resources. The state of the application must be stored by the client application itself. There should be no server side sessions: each request made by the client happens in complete isolation from any other request.

Generic Interface – The uniform interface of REST brings with it a certain level of generality automatically. When the interface is designed in terms of resources instead of functions, it remains much more generic. Clients can browse through the resources, apply standard HTTP operations on them and receive standard HTTP response codes in return. In addition, REST promotes the use of *Hypermedia as the Engine of Application State*, which means that via the representations the interface may provide options or guidance to the client about the direction if should take during the application flow. Granted, the resources and their representations still need to be carefully designed to make the interface more generic, but REST provides a good framework for doing so.

Scalability – Because of its stateless nature and the lack of server side sessions, RESTful interface is horizontally scalable by definition. New servers can be added behind the load balancer to handle the increased demand. RESTful interfaces are also easy to scale in terms of functionality because of their resource oriented architecture (ROA) [14]. When all the relevant artifacts of the system are modeled and exposed as resources with the uniform interface, client developers have a lot of leeway to implement web applications with different kind of functionality.

3.3 Clients

Clients of the web services defined above can range from simple scripts to other web services or JavaScript UIs running in the browser. Other examples would be normal desktop UIs, mobile applications or mashup applications utilizing multiple services. In this paper we are mainly interested in the UIs that run in the browser, although some of the following discussion will also hold for other types of clients.

For applications with a lot of functionality and interactivity, we endorse creating single page web applications, where the whole application runs within a single web page and no further page loads are necessary. This approach has several benefits when compared to traditional web applications where the browser is used for navigating from page to page. We will discuss these benefits further in later on.

Single page web applications are loaded into browser from a set of static bootstrapping files. These files include HTML, CSS and JavaScript files. Usually these files are loaded from the server but they may also be "pre-installed" onto the device. This is a common situation for mobile applications. After the application is loaded into the browser, it behaves like a traditional desktop application: UI components are shown and hidden from the screen dynamically and all user events are handled by the JavaScript code running in the browser. When new data is needed from the database, it is fetched using Ajax requests to server side resources. Similarly Ajax requests are used for creating, modifying, and deleting resources of the RESTful interface.

With this approach the state of the client side application always remains in the browser while the server is responsible for the server side resources and their states. This creates a strict and clear separation of concerns between the client and the server: the server exposes uniform interface for a set of resources without making any assumptions about clients' actions and the client is free to use these resources as it sees fit while maintaining the state of application flow.

The client becoming a "stand alone" application which uses external resources only when it really needs to, also has several benefits. The user interface becomes much more responsive since the client itself fulfills user's actions as far as possible. This, for one, reduces the network traffic down to minimum; only the data that is actually needed is queried from the server. All the user interface components and resources only need to be downloaded once. These advantages also increase the robustness of the application because the use of possibly unreliable network is decreased.

4 Case: Valvomo

To better illustrate the concept of single page applications and the partitioned way of implementing web applications we take a real world example. The application is called *Valvomo* (*Fin. control room*) and its design and implementation is joint work between us (the authors) and StrataGen Systems. The application domain sits in the field of paratransit. In short, paratransit is a flexible form of passenger transportation. It offers services that are not fixed (at least not strictly) to certain stops or schedules. Services are usually run by taxis or mini-buses. A typical use case is when a customer calls into a call center telling where and when she wants to be picked up and where she wants to go. The dispatcher at the other end of the line then enters the order into the system and tells the customer where and when exactly is she going to be picked up. Orders made by different customers may overlap and still be carried out by the same vehicle. This way the service that the vehicle is driving becomes dynamic. The level of flexibility offered to customers and vehicle operators varies considerably between different system providers.

4.1 Overview of the User Interface

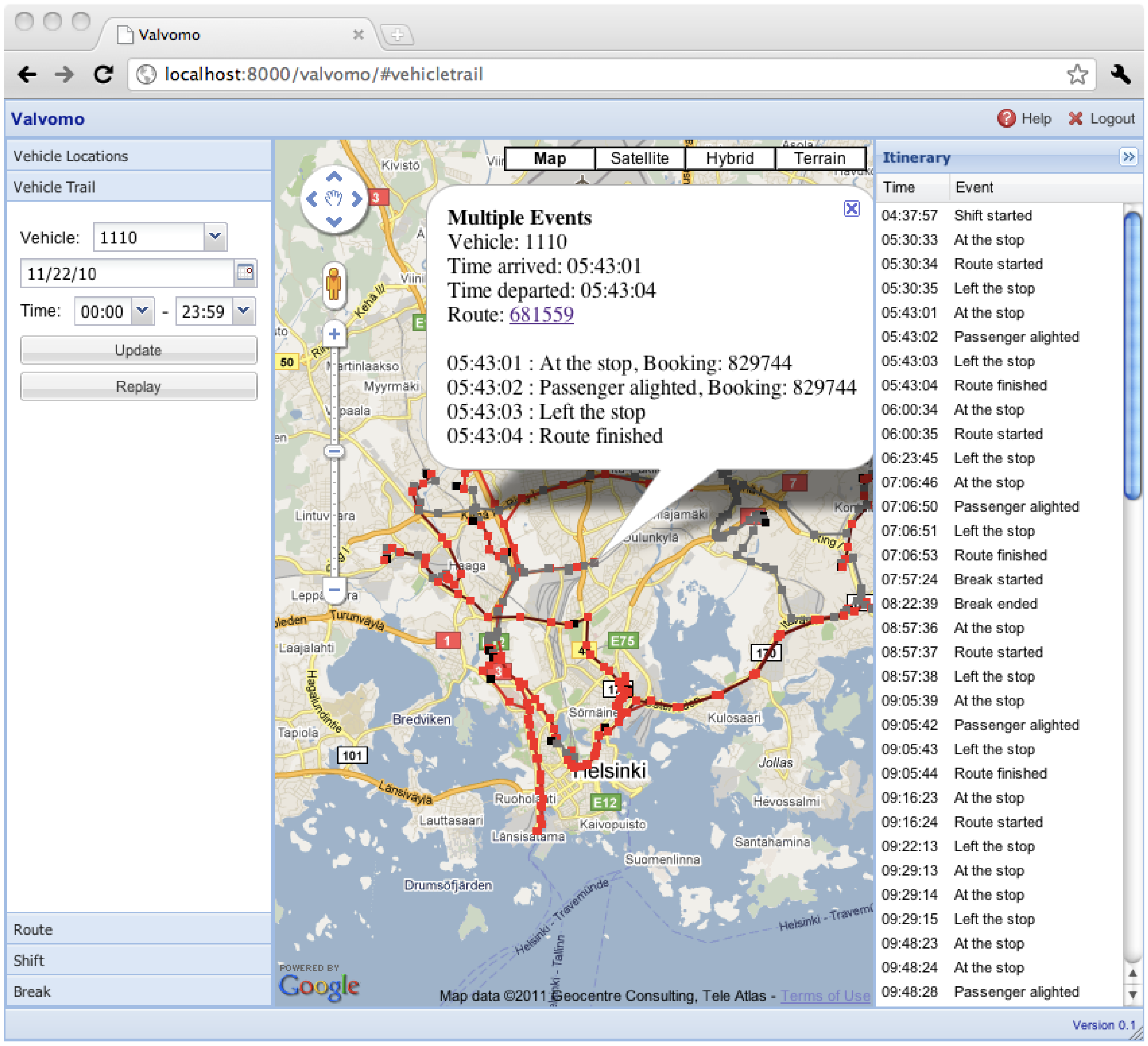
The purpose of the Valvomo application is to enable the vehicle operators to control their fleet better. Operators may track their vehicles in real time and see whether the services are running on time and immediately be notified if a service is running late. Moreover, operators are able to browse all the historical data that the vehicle has sent. This includes for example routes it has driven, customers picked up and dropped off, stops visited, breaks taken and the vehicle's operating hours. Figure 1 gives an overview of the user interface of the Valvomo application. It is a single page application with five accordion panels on the left, a map view in the center and a collapsible itinerary view on the right.

User may enter various search criteria using the input fields in different accordion panels on the left. Vehicle's actions are visible as a color encoded polyline on the map. By clicking either on the itinerary or on the nodes of the polyline, user is presented with detailed information about the corresponding vehicle event. The map will contain different data about the vehicle based on which accordion is active. The dockable itinerary panel on the right hand side contains chronological summary of the same data that is visible on the map.

Switching between accordions will cause the map and the itinerary to be refreshed with data related to the active accordion. The user interface was implemented so that each accordion will remember its state and data so the user is free to switch between accordions without having to worry about losing data. To respect the asynchronous and event driven programming model of JavaScript in the browser, all the networking is carried out with asynchronous Ajax request and the user interface is updated incrementally as data becomes available. For example when the user has requested a lot of data (a vehicle can easily have over 1000 events per day) a throbber icon is shown in the status bar and the user may navigate to other accordions while waiting for that data. When the user notices that the download has finished she can go back to the corresponding accordion and the data is visible on the map. We should point out that the user can actually use browser's back and forward buttons when going back and forth the accordion panels. Usually this is a problem with single page applications but we have taken special care that the application handles the navigation buttons in order to provide more fluent user experience.

4.2 Implementation of the User Interface

The user interface is a single page web application implemented in JavaScript, HTML, and CSS. For all the



Multiple Events
Vehicle: 1110
Time arrived: 05:43:01
Time departed: 05:43:04
Route: 681559

05:43:01 : At the stop, Booking: 829744
05:43:02 : Passenger alighted, Booking: 829744
05:43:03 : Left the stop
05:43:04 : Route finished

Time	Event
04:37:57	Shift started
05:30:33	At the stop
05:30:34	Route started
05:30:35	Left the stop
05:43:01	At the stop
05:43:02	Passenger alighted
05:43:03	Left the stop
05:43:04	Route finished
06:00:34	At the stop
06:00:35	Route started
06:23:45	Left the stop
07:06:46	At the stop
07:06:50	Passenger alighted
07:06:51	Left the stop
07:06:53	Route finished
07:57:24	Break started
08:22:39	Break ended
08:57:36	At the stop
08:57:37	Route started
08:57:38	Left the stop
09:05:39	At the stop
09:05:42	Passenger alighted
09:05:43	Left the stop
09:05:44	Route finished
09:16:23	At the stop
09:16:24	Route started
09:22:13	Left the stop
09:29:13	At the stop
09:29:14	At the stop
09:29:15	Left the stop
09:48:23	At the stop
09:48:24	At the stop
09:48:28	Passenger alighted

Figure 1: Valvomo

data it uses RESTful API which in turn is connected to the paratransit system. To obey the *same origin policy* [15] of browsers, the bootstrapping HTML and the REST API are in the same domain but under different paths. Overview of the different components in the Valvomo web application is given in Figure 2. It follows strictly the principles laid out in Section 3.

These kinds of highly dynamic user interfaces in browsers have always been somewhat cumbersome to implement. Main reasons for this include DOM, CSS and JavaScript incompatibilities between browsers. Also, the performance of the JavaScript interpreter and the DOM tree incorporated in browsers has been quite low. However, due to the intense competition in the browser market during the past few years, the performance of JavaScript interpreters has gotten significantly better. This has allowed for bigger and more complex JavaScript applications to be run in the browser.

The incompatibilities between browsers have made it almost impossible to implement any kind of dynamicity in web pages without the use of a JavaScript library that hides these incompatibilities. There are dozens of JavaScript libraries and toolkits to help developers make better JavaScript applications for the browser. Some of them focus on the language itself, some provide merely UI widgets and effects and some are full blown UI toolkits that completely hide the DOM and CSS.

For the Valvomo application we chose the Ext JS (now part of the Sencha [4] platform) toolkit mainly because of its suitability for creating single page JavaScript only

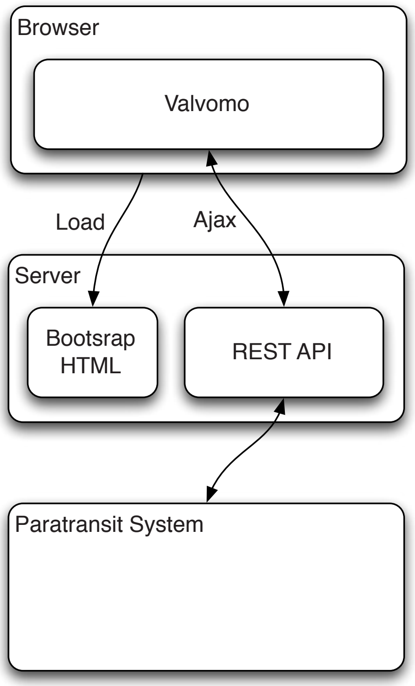


Figure 2: Valvomo Overview

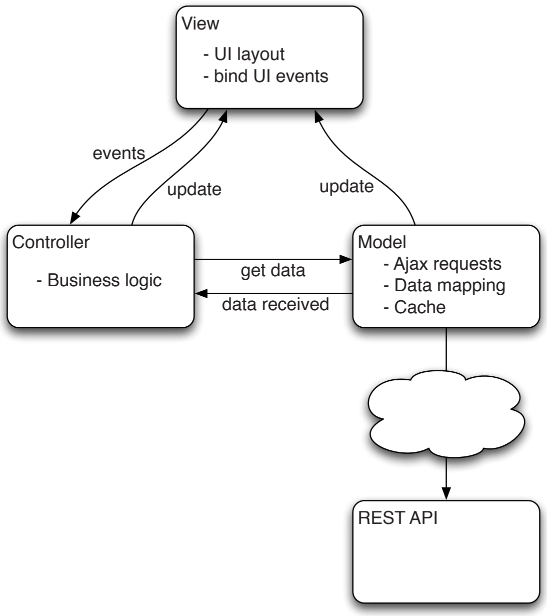


Figure 3: The MVC implementation in Valvomo

applications that fill the whole browser viewport. Furthermore, it is very mature and well documented. Ext JS completely hides the CSS and HTML from the developer and allows application to be created solely in JavaScript. The API for creating user interfaces reminds one of many popular desktop UI toolkits such as Qt or gtk.

In the implementation we followed the MVC pattern, used a lot of JavaScript closures for information hiding and put all Valvomo functions and objects under our own namespace. Because the whole user interface is implemented in JavaScript we were able to implement all components of the MVC pattern into the client. Our interpretation of the MVC pattern is depicted in Figure 3. Due to the nature of the resource oriented architecture of the server, this approach puts the server in the role of a mere data storage. Therefore, it becomes natural to wrap the networking code calling the RESTful interface, inside the *model* in the JavaScript application.

View – The view consists solely of the declaration of the user interface and its events. After the user interface and its layout is defined, the events are bound to the functions of the controller module. Ext JS offers a clean approach for defining the user interface in terms of – what Ext JS calls – *pre-configured classes*. What this means is that the definition of the user interface is broken into smaller components whose properties are preconfigured with suitable values. These components may then be reused and their properties possibly overridden before rendering on the screen.

Controller – The controller handles all events coming from the view and most of the events coming from the model. For example, each time the user clicks on a button, the controller handles the event and possibly uses the model to perform the desired action.

Model – The model issues all the Ajax calls required by the Valvomo application. It also implements automatic mappings between JSON representations of the service interface and JavaScript objects used by the Valvomo user interface. Usually, when the data is received from the server, a function of the controller module is registered to handle it. However some components of the view module, like data grids and combo boxes, are updated directly by the model.

4.3 The REST implementation

The RESTful interface on the server side is implemented in Python using the Django framework. Python was chosen over node.js, which is a very promising server side JavaScript environment. In Python's favor was its maturity and the maturity of its frameworks and libraries. For instance, the paratransit system uses Oracle database and provides a custom TCP/IP protocol for external connections. Python has a good support for connecting to oracle databases and excellent event-driven networking engine called Twisted. Moreover, the Django framework has a lot to offer for building RESTful web services. It provides a comprehensive set of plugins, flexible object relational mapper (ORM) and good test suite.

On top of Django we are using a small library called Piston, which streamlines the creation of RESTful web services in Django. It provides utilities such as mapping database tables to resources with uniform interface, mapping errors to proper HTTP response codes and serialization of domain objects into JSON or XML. We have also done some minor modifications to the library in order to make it more extensive and suitable for our use.

4.4 Characteristic of the REST interface

The definition of the Valvomo service interface bears no surprises; it is a very typical RESTful interface. That exactly is one of the benefits of REST and ROA. When the whole system is expressed as a set of resources and exposed via the uniform interface, the client developers can find the interface familiar and intuitive. In the following we briefly describe the relevant parts of the REST interface.

The paratransit system we are connecting to is a very mature application suite (15 years in production). Therefore, the domain model is very stable and well known by developers, managers, and, customers. Therefore, when we chose to apply the *Domain-driven design* (DDD) [7], we already had the *ubiquitous language* that all stakeholders could understand. This also made the identification of resources simpler because we were able to do fairly straightforward mapping from domain entities into RESTful resources.

URLs used in the interface adhere to the following scheme:

```
/api/{version}/{namespace}/{resource}/
```

Because we expect the interface to grow to cover all the resources of a fully functional paratransit system, we used `namespaces` to collect resources into logical groups. The `resource` maps to an entity collection in the domain model. If the `resource` is followed by `/{id}/`, the URL is mapped to a specific entity. For example, to refer to a vehicle number 123 one would use the URL.

```
/api/v1/fleet/vehicle/123/
```

Some of the resources also have subresources such as `fleet/vehicle/events/123/` which would contain all the events that the vehicle has sent. Sometimes a property of a resource is important enough that it makes sense to make it explicit and expose it as its own subresource. An example would be a list of cancelled orders:

```
/api/v1/scheduling/order/cancelled/
```

When a valid order is `POST`ed into the `order/cancelled/` resource, the underlying paratransit system performs the *order cancellation* function.

As its default data representation format, the interface uses JSON. XML is also supported and clients may use it as an alternative. By default, all representations have links to their related resources. For example, the representation of *order* does not embed any *passenger* representation but instead it contains a link to it. From the following example we have left out most of the fields of the order representation. Also the link is shortened to make it fit better.

```
{
  id: 345,
  passenger: {
    link: {
      href: "scheduling/passenger/567/",
      rel: "related"
    }
  }
}
```

Similarly the representation of *vehicle* contains links to its events and operator. This way the server may assist the client in finding new resources that may be of interest to it.

5 Experiences

Despite the fact that the Valvomo is an application that is still being actively developed, it has also been used successfully in production over six months now. New features are constantly being added to the user interface and new resources are added to the REST API. Currently, the Valvomo system is mainly being used in Finland but an initial installation is already in place in UK as well. When the application gets more mature and feature rich it will become part of StrataGen's US sites too.

In this section we discuss the experiences of building a web application in two partitions: a single page web application with a RESTful API on the server. We also go into more detail about pros and cons of this approach compared to a more traditional way of building web applications defined in Section 2. To gather the experiences presented in this section we used our own experience and interviewed several employees of the StrataGen Systems who have years of experience in creating web applications.

5.1 Advantages

The most important benefit of the partitioned way of building web applications is having the user interface completely decoupled from the server side implementation. This division along with making the server agnostic about its clients spawn several advantages.

1) *Accessibility of the service interface*. When the service makes no assumptions about its clients, any client

that conforms to the interface is allowed to use it. For example, this includes JavaScript code running in the browser or in a mobile device as well as programmatic clients such as other web services.

2) *Reusable service interface*. RESTful interface allows resources with uniform interface to be used freely to create new kinds of applications without any modifications to the server side implementation. When there is no predefined application flow between the server and the client, clients have much more control over how and what kind of applications to implement.

For example, in the case of Valvomo, the RESTful interface has been separated into its own product and renamed to Transit API. At the time of writing this the Transit API already has five other applications running on top of it. Three of them are programmatic clients written in C# and two of them are browser applications written in JavaScript. Moreover, the three JavaScript applications share code through a common library that interfaces with the Transit API. This kind of code reuse would be extremely difficult, if not impossible, with the traditional – tightly coupled – way of building web applications. In this scenario, there would have to be three full stack web applications and at least one web service API.

3) *Reusable user interface*. Not only does this approach allow different types of clients to exist but also the user interface may be transferred on top of different service as long as the service implements the same RESTful interface. For example, in the near future, the Valvomo application will be run on top of another paratransit system and there the Transit API will be implemented in C#. In addition, there are plans to implement the Transit API in front of another logistical system that is not paratransit at all and yet we are able to use the same Valvomo user interface without any code modifications.

4) *Responsibilities are easier to assign and enforce*. Traditionally, there has been a lot of confusion and "gray areas" in assigning responsibilities in web applications: which tasks should be handled in the client and which ones in the server? Furthermore, which technology should be used: HTML template, JavaScript or the server side implementation? However, when the client is completely decoupled from the server the two can be considered as two distinct products. This makes many aspects in assigning responsibilities implicit and many questions obvious. Following paragraphs give few examples.

Data validation – The server cannot trust any data that it receives from the client; everything needs to be validated. Granted, this is what should happen in traditional web applications too but inexperienced developers often get confused about this and believe that it is "their own" HTML page that is sending validated data. When implementing the server as stand alone RESTful web service it is much clearer that the data may come from any type of client and no assumptions can be made about its validity.

Error handling – The client and the server are both responsible for their own error handling. In between there is only HTTP and its standard return codes. When an error occurs in the server, it will return a corresponding HTTP response code to the client. The client reads the return code and acts accordingly. When there is an error in the client, the client handles the error as it sees best fit and the server never needs to know about that. In the traditional model where pages are loaded frequently the error handling is more complex. For example, errors may occur with cookies, sessions or page rendering. Careful design needs to be in place in order to determine which errors should be handled by which part of the application and what should be presented to the user. When this is not the case, users end up seing error pages like `404 Not Found`, `500 Internal Server Error` or a server side stack trace.

Localization – When the server exposes a generic RESTful interface, only the client needs to be localized. Error messages, calendar widgets and button labels are all localized in the client. Furthermore, when the whole client is implemented in JavaScript, the localization does not get fragmented over HTML, JavaScript and (for example) Java. Of course, textual content such as product descriptions need to have localized versions in the server's database but even then, the client asks for a specific version of the resource.

5) *Application flow and state*. According to REST guidelines, the application state is stored in the client and the states of all the resources are stored on the server. This unambiguously specifies the responsibilities of states between the client and the server. The client itself is responsible for maintaining the application flow and the server is free from having to store and maintain clients' sessions. No more does the client need to send the cookie to the server and the server does not have to worry about cleaning up expired and unused sessions.

6) *Lucid development model*. Traditionally, the development of complex web applications has been troubled with fragmentation of application logic over many technologies. Parts of the application logic are implemented using a server side programming language while other parts are implemented in say, HTML template language or JavaScript. To add to this disarray, the DOM is many times exploited to work around any limitations or attempts to do information hiding. However, with the partitioned way of implementation and a clear distinction of responsibilities both the client and the server can be implemented separately both having their own inter-

nal design. Moreover, when the whole client application is implemented in JavaScript only and using a programming model that we are familiar with from the desktop, it allows easy adoption of proven software patterns and best practices.

7) *Easier testing*. Web applications are traditionally difficult to test automatically. There are all kinds of tools that do help this process but still they are far from writing simple unit tests for a code that does not have user interface, let alone HTTP connection. The partitioned way of implementing web applications does not itself help in testing the user interface but testing the RESTful API can be very easy. Some frameworks – such as Django – offer a way to write unit tests against the REST API without even having to start a web server when the tests are run. Even if the framework does not have such support, automatic regression tests would still be fairly easy to write using isolated HTTP request and checking the response codes and possible contents.

8) *Network traffic*. When the user interface is implemented within a single page, there is no need to download any additional HTML or CSS files after the application has been bootsrapped and running. Only the data that is requested by the user is downloaded from the server. Thus, there is no overhead in the network traffic and from this follows that the user interface stays responsive at all times and therefore becomes more robust. On the other hand, when the services on the server conform to the RESTful architectural style, the full caching capabilities of the HTTP become available. In many of the traditional web applications the payload data is embedded within the downloaded HTML page. That makes the data much more difficult – if not even impossible – to cache efficiently.

5.2 Disadvantages

1) *Framework support*. Web frameworks offer a lot of conventions, tools and code generators for building traditional web applications. While the RESTful service interface can benefit from these tools, the single page user interface gets no benefit. The level of guidance the developer gets for building the user interface is completely dependent upon the chosen JavaScript toolkit.

2) *Search engines*. It is very difficult for the search engines to crawl a single page web application. They could crawl the RESTful service interface but the documents returned by the interface are without any context and therefore difficult to rank.

3) *Accessibility*. For visually impaired, single page web applications can be difficult to use. In traditional web applications with more static pages the accessibility is easier to take into account. While possible, it is much more laborious to implement a web application that has good support for accessibility.

4) *Lack of HTML*. One of the best features of HTML and CSS is that web pages can be designed and even written by graphical designers. In single page web applications, with heavy use of JavaScript, this becomes impossible. The user interface must be implemented by a software developer.

6 Related Work

As mentioned earlier, our approach embraces many of the existing methods and patterns. We now briefly describe some of these existing approaches and underline how our solution stands out among them.

6.1 MVC in Web Applications

While there are other approaches into building web applications – such as continuations [6, 13] – the MVC pattern is the one adopted by most of the popular web frameworks today. There has been a lot of research on how to implement the MVC pattern in the realm of web applications. Specifically, [12] defines and discusses all the different scenarios of how the MVC pattern can be implemented in web applications. The paper elaborates on how the MVC pattern should be incorporated by Rich Internet Applications (RIAs). It is also their conclusion that while the components of MVC may be distributed differently in different types of applications, for RIAs, it is best to implement the full MVC stack in the browser.

Another paper [10] suggests a dynamic approach of distributing the components of MVC between the client and the server. This is accomplished through a method called *flexible web-application partitioning* (fwap) and it allows for different partitioning schemes without any modifications to the code. For example, depending on the use case it may be appropriate to sometimes deploy controller on the server while at other times it is best to have it in the browser.

However, for all the popular MVC web frameworks – such as Struts, Rails or ASP .Net MVC – the term MVC always refers to the traditional way of partitioning web applications (as described in Section 2). In this model the whole MVC stack is implemented in the server and the view is transferred into the browser after being generated in the server. Of course, the view may have dynamic features through the use of JavaScript and CSS but it does not affect how the components of the MVC are laid out.

6.2 RESTful Web Services

For a *web site* that supports both browser based user interface and programmable API, it is common to have, indeed, two separate interfaces for these purposes. Good examples are Netflix (http://www.netflix.com/) and del.icio.us (http://del.icio.us/) which both have separate interfaces for browser and other clients. Usually the interface accessed by the browser is so tightly coupled to the application flow that it would be very difficult for programmatic clients to consume it. Therefore a separate API interface is required. This API can then be a pure RESTful API that is looking at the same data as the browser interface.

There also seems to be confusion in the terminology when REST is being discussed in the context of web applications. It is often assumed that a web application is RESTful if the URLs in the browser's address bar look readable and intuitive. While this is a good thing, it does not mean that the web application itself is being RESTful. Also it does not mean that interface would be easy to consume by programmatic clients.

6.3 Mashup Applications

Mashups are fairly good good example of building web applications according to the partitioned pattern described in this paper. Mashup applications indeed decouple the user interface from the third party server side implementations. They use services that are accessible without predefined application flow to create new kinds of applications: services are agnostic about their clients and clients use services as they see best fit.

However, mashups fall into slightly different category than what is the focus of this paper. Even though mashup applications consume external service APIs, the applications themselves may be implemented in the traditional, tightly coupled, way while only using external services for some parts of the application. The focus of this paper is to represent a pattern for building complex web applications and provide experiences in doing so.

6.4 Comparing Our Approach

What differentiates our approach from these existing solutions is that we clearly assign the whole MVC stack into the browser. Moreover if the application does not need any services provided by the server that becomes the whole application. At the minimum however, there is usually some kind of persistent data that the application needs and for that it uses the RESTful service. Of course, it depends on the application, how many and what kind of services it consumes. In any case the state of the application and even the application logic – as much as possible – stays in the browser.

Another distinctive feature in our approach is providing a single interface for both applications running in the browser and programmatic clients accessing the interface from various environments. This explicitly decouples the application running in the browser from the server side implementation because the same interface must be consumable by other clients too. Therefore, in the interface, there cannot be any customizations or assumptions made about the browser side application.

7 Future Work

The most important part of future work is the design and implementation of a unified and more coherent authentication and authorization scheme. Currently, the Valvomo service API supports traditional cookie based authentication for browser clients and two legged OAuth [2] for programmatic clients. The authorization, in turn, is implemented somewhat specifically for each use case. The next research topic for will be finding out what is best way to implement authentication so that the same method works for both the browser clients and the native clients. More importantly, this should be done without cookies. As part of this investigation we also seek to find a generic solution for implementing authorization of resources. Right now, it seems that some kind of *Role Based Access Control* that can be defined per resource might be suitable.

Another subject for future work is implementing the Valvomo service API using node.js or another server side JavaScript framework that conforms to the CommonJS specification. Running JavaScript on both sides of the application would provide a more uniform development environment and enable better code reuse because we would be able to share code like domain objects, validators, utility libraries and test cases.

8 Conclusions

The high demand for feature rich web applications have made them very complex to develop and eventually difficult to maintain. The gradual shift from static web pages into dynamic web applications has created an unnatural development environment for them. We need to rethink how these complex web application should be developed and run.

In this paper we have presented our experiences in partitioned way of building complex and feature rich web applications. This pattern advises into breaking the application clearly in two parts: the client and the server. To make the division unambiguous and explicit the server interface should be implemented as a RESTful web service. Browser based clients on the other hand should be implemented as single page web applications to maximize interactivity and responsiveness of the user interface.

To prove the usefulness of the suggested pattern we have utilized it in a large scale industrial application. The experiences of this undertaking are presented in section 5.

References

[1] Django. `http://www.djangoproject.com/`, 2011.

[2] Oauth. `http://oauth.net/`, 2011.

[3] Ruby on rails. `http://rubyonrails.org/`, 2011.

[4] Sencha. `http://www.sencha.com/`, 2011.

[5] The spring framework. `http://www.springsource.org/`, 2011.

[6] DUCASSE, S., LIENHARD, A., AND RENGGLI, L. Seaside: A flexible environment for building dynamic web applications. *IEEE Softw. 24* (September 2007), 56–63.

[7] EVANS. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[8] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[9] KRASNER, G., AND POPE, S. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program. 1*, 3 (1988), 26–49.

[10] LEFF, A., AND RAYFIELD, J. T. Web-application development using the model/view/controller design pattern. *Enterprise Distributed Object Computing Conference, IEEE International 0* (2001), 0118.

[11] MIKKONEN, T., AND TAIVALSAARI, A. Web applications ? spaghetti code for the 21st century. *Software Engineering Research, Management and Applications, ACIS International Conference on 0* (2008), 319–328.

[12] MORALES-CHAPARRO, R., L. M. P. J. C., AND SÁNCHEZ-FIGUEROA, F. Mvc web design patterns and rich internet applications. In *Proceedings of the Conference on Engineering Software and Databases* (2007).

[13] QUEINNEC, C. Continuations and web servers. *Higher Order Symbol. Comput. 17* (December 2004), 277–295.

[14] RICHARDSON, L., AND RUBY, S. *RESTful Web Services*. O'Reilly, 2007.

[15] RUDERMAN, J. The same origin policy, 2001.

[16] SOUDERS, S. *High performance web sites*, first ed. O'Reilly, 2007.

[17] SOUDERS, S. *Even Faster Web Sites: Performance Best Practices for Web Developers*, 1st ed. O'Reilly Media, Inc., 2009.

Exploring the Relationship Between Web Application Development Tools and Security

Matthew Finifter
University of California, Berkeley
finifter@cs.berkeley.edu

David Wagner
University of California, Berkeley
daw@cs.berkeley.edu

Abstract

How should software engineers choose which tools to use to develop secure web applications? Different developers have different opinions regarding which language, framework, or vulnerability-finding tool tends to yield more secure software than another; some believe that there is no difference at all between such tools. This paper adds quantitative data to the discussion and debate.

We use manual source code review and an automated black-box penetration testing tool to find security vulnerabilities in 9 implementations of the same web application in 3 different programming languages. We explore the relationship between programming languages and number of vulnerabilities, and between framework support for security concerns and the number of vulnerabilities. We also compare the vulnerabilities found by manual source code review and automated black-box penetration testing.

Our findings are: (1) we do not find a relationship between choice of programming language and application security, (2) automatic framework protection mechanisms, such as for CSRF and session management, appear to be effective at precluding vulnerabilities, while manual protection mechanisms provide little value, and (3) manual source code review is more effective than automated black-box testing, but testing is complementary.

1 Introduction

The web has become the dominant platform for new software applications. As a result, new web applications are being developed all the time, causing the security of such applications to become increasingly important. Web applications manage users' personal, confidential, and financial data. Vulnerabilities in web applications can prove costly for organizations; costs may include direct financial losses, increases in required technical support, and tarnished image and brand.

Security strategies of an organization often include developing processes and choosing tools that reduce the number of vulnerabilities present in live web applications. These software security measures are generally focused on some combination of (1) building secure software, and (2) finding and fixing security vulnerabilities in software after it has been built.

How should managers and developers in charge of these tasks decide which tools – languages, frameworks, debuggers, etc. – to use to accomplish these goals? What basis of comparison do they have for choosing one tool over another? Common considerations for choosing (e.g.,) one programming language over another include:

- How familiar staff developers are with the language.
- If new developers are going to be hired, the current state of the market for developers with knowledge of the language.
- Interoperability with and re-usability of existing in-house and externally-developed components.
- Perceptions of security, scalability, reliability, and maintainability of applications developed using that language.

Similar considerations exist for deciding which web application development framework to use and which process to use for finding vulnerabilities.

This work begins an inquiry into how to improve one part of the last of these criteria: the basis for evaluating a tool's inclination (or disinclination) to contribute to application security.

Past research and experience reveal that different tools *can* have different effects on application security. The software engineering and software development communities have seen that an effective way to preclude buffer overflow vulnerabilities when developing a new application is to simply use a language that offers automatic memory management. We have seen also that even if other requirements dictate that the C language must be used for development, using the safer `strlcpy` instead of `strcpy` can preclude the introduction of many buffer overflow vulnerabilities.

This research is an exploratory study into the security properties of some of the tools and processes that organizations might choose to use during and after they build their web applications. We seek to understand whether the choice of language, web application development framework, or vulnerability-finding process affects the security of the applications built using these tools.

We study the questions by analyzing 9 independent implementations of the same web application. We collect data on (1) the number of vulnerabilities found in these implementations using both a manual security review and an automatic black-box penetration testing tool, and (2) the level of security support offered by the frameworks. We look in these data sets for patterns that might indicate differences in application security between programming languages, frameworks, or processes for finding vulnerabilities. These patterns allow us to generate and test hypotheses regarding the security implications of the various tools we consider.

This paper's main contributions are as follows:

- We develop a methodology for studying differences in the effect on application security that different web application development tools may have. The tools we consider are programming languages, web application development frameworks, and processes for finding vulnerabilities.
- We generate and test hypotheses regarding the differences in security implications of these tools.
- We develop a taxonomy for framework-level defenses that ranges from *always on* framework support to no framework support.
- We find evidence that automatic framework-level defenses work well to protect web applications, but that even the best manual defenses will likely continue to fall short of their goals.
- We find evidence that manual source code analysis and automated black-box penetration testing are complementary.

2 Goals

Programming language. We want to measure the influence that programming language choice has on the security of the software developed using that language. If such an influence exists, software engineers (or their managers) could take it into account when planning which language to use for a given job. This information could help reduce risk and allocate resources more appropriately.

We have many reasons to believe that the features of a programming language could cause differences in the security of applications developed using that language. For example, research has shown that type systems can statically find (and therefore preclude, by halting compilation) certain types of vulnerabilities [21, 20]. In general, static typing can find bugs (any of which could be a vulnerability) that may not have been found until the time of exploitation in a dynamically-typed language.

Also, one language's standard libraries might be more usable, and therefore less prone to error, than another's. A modern exception handling mechanism might help developers identify and recover from dangerous scenarios.

But programming languages differ in many ways beyond the languages themselves. Each language has its own community, and these often differ in their philosophies and values. For example, the Perl community values TMTOWTDI ("There's more than one way to do it") [4], but the Zen of Python [16] states, "[t]here should be one – and preferably, only one – obvious way to do it." Clear documentation could play a role as well.

Therefore, we want to test whether the choice of language measurably influences overall application security. If so, it would be useful to know whether one language fares better than another for any specific class of vulnerability. If this is the case, developers could focus their efforts on classes for which their language is lacking good support, and not worry so much about those classes in which data show their language is strong.

Web application development framework. Web application development frameworks provide a set of libraries and tools for performing tasks common in web application development. We want to evaluate the role that they play in the development of secure software. This can help developers make more informed decisions when choosing which technologies to use.

Recently, we have seen a trend of frameworks adding security features over time. Many modern frameworks take care of creating secure session identifiers (e.g., Zend, Ruby on Rails), and some have added support for automatically avoiding cross-site scripting (XSS) or cross-site request forgery (CSRF) vulnerabilities (e.g., Django, CodeIgniter). It is natural to wonder if frameworks that are pro-active in developing security features yield software with measurably better security, but up to this point we have no data showing whether this is so.

Vulnerability-finding tool. Many organizations manage security risk by assessing the security of software before they deploy or ship it. For web applications, two prominent ways to do so are (1) black-box penetration testing, using automated tools designed for this purpose, and (2) manual source code analysis by an analyst knowledgeable about security risks and common vulnerabilities. The former has the advantage of being mostly automated and being cheaper; the latter has a reputation as

Team Number	Language	Frameworks used
1	Perl	DBIx::DataModel, Catalyst, Template Toolkit
2	Perl	Mason, DBI
5	Perl	Gantry, Bigtop, DBIx::Class
3	Java	abaXX, JBoss, Hibernate
4	Java	Spring, Spring Web Flow, Hibernate, Acegi Security
9	Java	Equinox, Jetty, RAP
6	PHP	Zend Framework, OXID framework
7	PHP	proprietary framework
8	PHP	Zend Framework

Table 1: Set of frameworks used by each team.

more comprehensive but more expensive. However, we are not aware of quantitative data to measure their relative effectiveness. We work toward addressing this problem by comparing the effectiveness of manual review to that of automated black-box penetration testing. Solid data on this question may help organizations make an informed choice between these assessment methods.

3 Methodology

In order to address these questions, we analyze several independent implementations of the same web application specification, written using different programming languages and different frameworks. We find vulnerabilities in these applications using both manual source code review and automated black-box penetration testing, and we determine the level of framework support each implementation has at its disposal to help it contend with various classes of vulnerabilities. We look for associations between: (1) programming language and number of vulnerabilities, (2) framework support and number of vulnerabilities, and (3) number of vulnerabilities found by manual source code analysis and by automated black-box penetration testing.

We analyze data collected in a previous study called Plat_Forms [19]. In that work, the researchers devised and executed a controlled experiment that gave 9 professional programming teams the same programming task for a programming contest. Three of the teams used Perl, three used PHP, and the remaining three used Java.

The contest rules stated that each team had 30 hours to implement the specification of a web application called People By Temperament [18]. Each team chose which frameworks they were going to use. There was little overlap in the set of frameworks used by teams using the same programming language. Table 1 lists the set of frameworks used by each team.

The researchers collected the 9 programs and analyzed their properties. While they were primarily concerned with metrics like performance, completeness, size, and usability, we re-analyze their data to evaluate the security properties of these 9 programs.

Each team submitted a complete source code package and a virtual machine image. The VM image runs a web server, which hosts their implementation of People by Temperament. The source code packages were trimmed to remove any code that was not developed specifically for the contest, and these trimmed source code packages were released under open source licenses.[1]

For our study, we used the set of virtual machine images and the trimmed source code packages. The Plat_Forms study gathered a lot of other data (e.g., samples at regular intervals of the current action of each developer) that we did not need for the present study. The data from our study are publicly available online.[2]

3.1 People by Temperament

We familiarized ourselves with the People by Temperament application before beginning our security analysis. The application is described as follows:

> PbT (People by Temperament) is a simple community portal where members can find others with whom they might like to get in contact: people register to become members, take a personality test, and then search for others based on criteria such as personality types, likes/dislikes, etc. Members can then get in contact with one another if both choose to do so. [18]

People by Temperament is a small but realistic web application with a non-trivial attack surface. It has security goals that are common amongst many web applications. We list them here:

- **Isolation between users.** No user should be able to gain access to another user's account; that is, all information input by a user should be integrity-protected with respect to other users. No user should be able to view another user's confidential information without approval. Confidential information includes a user's password, full name, email address, answers to personality test questions, and list of contacts. Two users are allowed to view each other's full name and email address once they have agreed to be in contact with one another.
- **Database confidentiality and integrity.** No user should be able to directly access the database, since it contains other users' information and it may contain confidential web site usage information.
- **Web site integrity.** No user should be able to vandalize or otherwise modify the web site contents.

[1] `http://www.plat-forms.org/sites/default/files/platforms2007solutions.zip`

[2] `http://www.cs.berkeley.edu/~finifter/datasets/`

Integer-valued	Stored XSS Reflected XSS SQL injection Authentication or authorization bypass
Binary	CSRF Broken session management Insecure password storage

Table 2: The types of vulnerabilities we looked for. We distinguish binary and integer-valued vulnerability classes. Integer-valued classes may occur more than once in an implementation. For example, an application may have several reflected XSS vulnerabilities. The binary classes represent presence or absence of an application-wide vulnerability. For example, in all implementations in this study, CSRF protection was either present throughout the application or not present at all.

Review Number	**Dev. Team Number**	**Lang.**	**SLOC**	**Review Time (min.)**	**Review Rate (SLOC/hr)**
1	6	PHP	2,764	256	648
2	3	Java	3,686	229	966
3	1	Perl	1,057	210	302
4	4	Java	2,021	154	787
5	2	Perl	1,259	180	420
6	8	PHP	2,029	174	700
7	9	Java	2,301	100	1,381
8	7	PHP	2,649	99	1,605
9	5	Perl	3,403	161	1,268

Table 3: Time taken for manual source code reviews, and number of source lines of code for each implementation.

- **System confidentiality and integrity.** No user should be able to gain access to anything on the web application server outside of the scope of the web application. No user should be able to execute additional code on the server.

The classes of vulnerabilities that we consider are presented in Table 2. A vulnerability in any of these classes violates at least one of the application's security goals.

3.2 Vulnerability data

We gathered vulnerability data for each implementation in two distinct phases. In the first phase, a reviewer performed a manual source code review of the implementation. In the second phase, we subjected the implementation to the attacks from an automated black-box web penetration testing tool called Burp Suite Pro [17].

We used both methods because we want to find as many vulnerabilities as we can. We hope that any failings of one method will be at least partially compensated by the other. Although we have many static analysis tools at our disposal, we chose not to include them in this study because we are not aware of any that work equally well for all language platforms. Using a static analysis tool that performs better for one language than another would have introduced systematic bias into our experiment.

3.2.1 Manual source code review

One reviewer (Finifter) reviewed all implementations. This reviewer is knowledgeable about security and had previously been involved in security reviews.

Using one reviewer for all implementations avoids the problem of subjectivity between different reviewers that would arise if the reviewers were to examine disjoint sets of implementations. We note that having multiple reviewers would be beneficial if each reviewer was able to review all implementations independently of all other reviewers.

The reviewer followed the Flaw Hypothesis Methodology [6] for conducting the source code reviews. He used the People by Temperament specification and knowledge of flaws common to web applications to develop a list of types of vulnerabilities to look for. He performed two phases of review, first looking for specific types of flaws from the list, then comprehensively reviewing the implementation. He confirmed each suspected vulnerability by developing an exploit.

We randomly generated the order in which to perform the manual source code reviews in order to mitigate any biases that may have resulted from choosing any particular review order. Table 3 presents the order in which the reviewer reviewed the implementations as well as the amount of time spent on each review.

The reviewer spent as much time as he felt necessary to perform a complete review. As shown in Table 3, the number of source lines of code reviewed per hour varies widely across implementations; the minimum is 302 and the maximum is 1,605. Cohen [7] states that "[a]n expected value for a meticulous inspection would be 100-200 LOC/hour; a normal inspection might be 200-500." It is unclear upon what data or experience these numbers are based, but we expect the notion of "normal" to vary across different types of software. For example, we expect a review of a kernel module to proceed much more slowly than that of a web application. Additionally, we note that the number of source lines of code includes both static HTML content and auto-generated code, neither of which tends to require rigorous security review.

To help gauge the validity of our data for manual source code review, we test the following hypotheses:

- Later reviews take less time than earlier reviews.
- More vulnerabilities were found in later reviews.
- Slower reviews find more vulnerabilities.

If we find evidence in support of either of the first two hypotheses, this may indicate that the reviewer gained experience over the course of the reviews, which may have biased the manual review data. A more experienced reviewer can be expected to find a larger fraction of the vulnerabilities in the code, and if this fraction increases with each review, we expect our data to be biased in showing those implementations reviewed earlier to be more secure. Spearman's rho for these two hypotheses is $\rho = 0.633$ ($p = 0.0671$) and $\rho = -0.0502$ ($p = 0.8979$), respectively, which means that we do not find evidence in support of either of these hypotheses.

If we find evidence in support of the third of these hypotheses, this may indicate that the reviewer did not allow adequate time to complete one or more of the reviews. This would bias the data to make it appear that those implementations reviewed more quickly are more secure than those for which the review proceeded more slowly. The correlation coefficient between review rate and number of vulnerabilities found using manual analysis is $r = 0.0676$ ($p = 0.8627$), which means we do not find evidence in support of this hypothesis. The lack of support for these hypotheses modestly increases our confidence in the validity of our manual analysis data.

3.2.2 Black-box testing

We used Portswigger's Burp Suite Professional version 1.3.08 [17] for black box testing of the implementations. We chose this tool because a previous study has shown it to be among the best of the black box testing tools [11] and because it has a relatively low cost.

We manually spidered each implementation before running Burp Suite's automated attack mode (called "scanner"). All vulnerabilities found by Burp Suite were manually verified and de-duplicated (when necessary).

3.3 Framework support data

We devised a taxonomy to categorize the level of support a framework provides for protecting against various vulnerability classes. We distinguish levels of framework support as follows.

The strongest level of framework support is *always on*. Once a developer decides to use a framework that offers always-on protection for some vulnerability class, a vulnerability in that class cannot be introduced unless the developer stops using the framework. An example of this is the CSRF protection provided automatically by Spring Web Flow [10], which Team 4 used in its implementation. Spring Web Flow introduces the notion of tokens, which define flows through the UI of an application, and these tokens double as CSRF tokens, a well-known protection mechanism for defending against CSRF vulnerabilities. Since they are integral to the functionality the framework provides, they cannot be removed or disabled without ceasing to use the framework entirely.

The next strongest level of framework support is *opt-out* support. This level of support provides protection against a vulnerability class by default, but it can be disabled by the developer if he so desires. Team 2's custom ORM framework provides opt-out support for SQL injection. If the framework is used, SQL injection cannot occur, but a developer can opt out by going around the framework to directly issue SQL queries.

Opt-in support refers to a defense that is disabled by default, but can be enabled by the developer to provide protection throughout the application. Enabling the protection may involve changing a configuration variable or calling into the framework code at initialization time. Once enabled, opt-in support defends against all subsequent instances of that vulnerability class. Acegi Security, used by Team 4, provides a `PasswordEncoder` interface with several different implementations. We consider this opt-in support because a developer can select an implementation that provides secure password storage for his application.

Manual support is the weakest level of framework support. This term applies if the framework provides a useful routine to help protect against a vulnerability class, but that routine must be utilized by the developer *each time protection is desired*. For example, many frameworks provide XSS filters that can be applied to untrusted data before it is included in the HTML page. These filters spare a developer the burden of writing a correct filter, but the developer must still remember to invoke the filter every time untrusted data is output to a user. Manual support is weak because a developer has many opportunities to make an error of omission. Forgetting to call a routine (such as an XSS filter) even once is enough to introduce a vulnerability. We use the term *automatic* support to contrast with manual support; it refers to any level of support stronger than manual support.

For each implementation, we looked at the source code to discover which frameworks were used. We read through the documentation for each of the frameworks to find out which protection mechanisms were offered for each vulnerability class we consider. We defined the implementation's level of support for a particular vulnerability class to be the highest level of support offered by any framework used by the implementation.

3.4 Individual vulnerability data

We gather data about each individual vulnerability to deepen our understanding of the current framework ecosystem, the reasons that developers introduce vulnerabilities, and the limitations of manual review. For each vulnerability, we determine how far the developers would have had to stray from their chosen frameworks in order to find manual framework support that could have prevented the vulnerability. Specifically, we label each vulnerability with one of the following classifications:

	Java	Perl	PHP
Number of programmers	9	9	9
Mean age (years)	32	32	32
Mean experience (years)	7.1	8.7	9.8

Table 4: Statistics of the programmers.

1. **Framework used.** Framework support that could have prevented this vulnerability exists in at least one of the frameworks used by the implementation.
2. **Newer version of framework used.** Framework support exists in a newer version of one of the frameworks used by the implementation.
3. **Another framework for language used.** Framework support exists in a different framework for the same language used by the implementation.
4. **Some framework for some language.** Framework support exists in some framework for some language other than the one used by the implementation.
5. **No known support.** We cannot find framework support in any framework for any language that would have stopped the vulnerability.

We label each vulnerability with the *lowest* level at which we are able to find framework support that could have prevented the vulnerability. We do so using our awareness and knowledge of state-of-the-art frameworks as well as the documentation frameworks provide.

Similarly, for each vulnerability, we determine the level at which the developers could have found automatic (i.e., opt-in or better) framework support. We evaluate this in the same manner as we did for manual support, but with a focus only on automatic protection mechanisms.

3.5 Threats to validity

Experimental design. The Plat_Forms data were gathered in a non-randomized experiment. This means that the programmers chose which language to use; the language was not randomly assigned to them by the researchers. This leaves the experiment open to selection bias; it could be the case that more skilled programmers tend to choose one language instead of another. As a result, any results we find represent what one might expect when hiring new programmers who choose which language to use, rather than having developers on staff and telling them which language to use.

Programmer skill level. If the skill level of the programmers varies from team to team, then the results represent the skills of the programmers, not inherent properties of the technologies they use for development. Fortunately, the teams had similar skills, as shown in Table 4.

Security awareness. Security was not explicitly mentioned to the developers, but all were familiar with security practices because their jobs required them to be [23]. It may be that explicitly mentioning security or specifying security requirements would have changed the developers' focus and therefore the security of the implementations, but we believe that the lack of special mention is realistic and representative of many programming projects. In the worst case, this limits the external validity of our results to software projects in which security is not explicitly highlighted as a requirement.

Small sample size. Due to the cost of gathering data of this nature, the sample size is necessarily small. This is a threat to external validity because it makes it difficult to find statistically significant results. In the worst case, we can consider this a case study that lets us generate hypotheses to test in future research.

Generalization to other applications. People by Temperament is one web application, and any findings with respect to it may not hold true with respect to other web applications, especially those with vastly different requirements or of much larger scale. The teams had only 30 hours to complete their implementation, which is not representative of most real software development projects. Despite these facts, the application does have a significant amount of functionality and a large enough attack surface to be worth examining.

Number of vulnerabilities. We would like to find the total number of vulnerabilities present in each implementation, but each analysis (manual and black-box) finds only some fraction of them. If the detection rate of our manual analysis is better for one language or one implementation than it is for another, this is a possible threat to validity. However, we have no reason to expect a systematic bias of this nature, as the reviewer's level of experience in manual source code review is approximately equivalent for all three languages. At no time did the reviewer feel that any one review was easier or more difficult than any other.

Similarly, if the detection rate of our black-box tool is better for one language or implementation than it is for another, this could pose a threat to validity. We have no reason to believe this is the case. Because black-box testing examines only the input-output behavior of a web application and not its implementation, it is inherently language- and implementation-agnostic, which leads us to expect that it has no bias for any one implementation over any other.

Vulnerability severity. Our analysis does not take into account any differences in vulnerability severity. Using our analysis, an implementation with many low-severity vulnerabilities would appear less secure than an implementation with only a few very high-severity vulnerabilities, though in fact the latter system may be less secure overall (e.g., expose more confidential customer data).

We have no reason to believe that average vulnerability severity varies widely between implementations, but we did not study this in detail.

Vulnerabilities introduced later in the product cycle. Our study considers only those vulnerabilities introduced during initial product development. Continued development brings new challenges for developers that simply were not present in this experiment. Our results do not answer any questions about vulnerabilities introduced during code maintenance or when features are added after initial product development.

Framework documentation. If a framework's documentation is incomplete or incorrect, or if we misread or misunderstood the documentation, we may have mislabeled the level of support offered by the framework. However, the documentation represents the level of support a developer could reasonably be expected to know about. If we were unable to find documentation for protection against a class of vulnerabilities, we expect that developers would struggle as well.

Awareness of frameworks and levels of support. There may exist frameworks that we are not aware of that provide strong framework support for a vulnerability class. If this is the case, our labeling of vulnerabilities with the nearest level at which framework support exists (Section 3.4) may be incorrect. We have made every effort to consider all frameworks with a significant user base in order to mitigate this problem, and we have consulted several lists of frameworks (e.g., [14]) in order to make our search as thorough as reasonably possible.

4 Results

We look for patterns in the data and analyze it using statistical techniques. We note that we do not find many statistically significant results due to the limited size of our data set.

4.1 Total number of vulnerabilities

Figure 1 displays the total number of vulnerabilities found in each implementation, including both integer-valued and binary vulnerability classes (we count a binary vulnerability as one vulnerability in these aggregate counts).

Every implementation had at least one vulnerability. This suggests that building secure web applications is difficult, even with a well-defined specification, and even for a relatively small application.

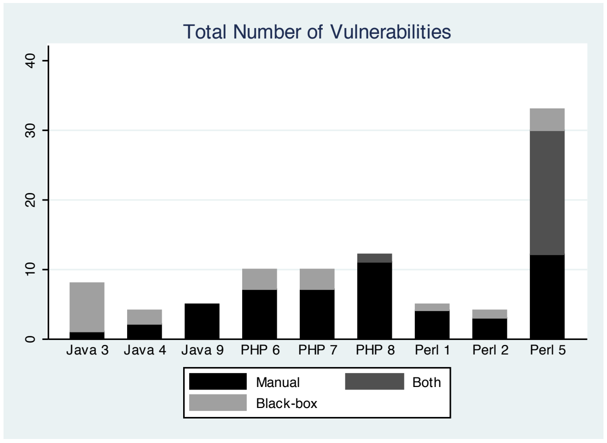


Figure 1: The total number of vulnerabilities found in the 9 implementations of People by Temperament. The x-axis is labeled with the language and team number.

One of the Perl implementations has by far the most vulnerabilities, primarily due to its complete lack of XSS protection.[3] This does not seem to be related to the fact that Perl is the language used, however, since the other two Perl implementations have only a handful of vulnerabilities, and few XSS vulnerabilities.

The Java implementations have fewer vulnerabilities than the PHP implementations. In fact, every Java implementation contains fewer vulnerabilities than each PHP implementation.

A one-way ANOVA test reveals that the overall relationship in our data set between language and total number of vulnerabilities is not statistically significant ($F = 0.57$, $p = 0.592$). Because this appears to be largely due to the obvious lack of a difference between Perl and each of the other two languages, we also perform a Student's t-test for each pair of languages, using the Bonferroni correction to correct for the fact that we test 3 separate hypotheses. As expected, we do not find a significant difference between PHP and Perl or between Perl and Java. We find a statistically significant difference between PHP and Java ($p = 0.033$).

4.2 Vulnerability classes

Figure 2 breaks down the total number of vulnerabilities into the separate integer-valued vulnerability classes, and the shaded rows in Table 5 present the data for the binary vulnerability classes.

XSS. A one-way ANOVA test reveals that the relationship between language and number of stored XSS vulnerabilities is not statistically significant ($F = 0.92$, $p = 0.4492$). The same is true for reflected XSS ($F = 0.43$, $p = 0.6689$).

[3]None of our conclusions would differ if we were to exclude this apparent outlier.

		CSRF		Session Management		Password Storage	
Team Number	Language	Vulnerable?	Framework Support	Vulnerable?	Framework Support	Vulnerable?	Framework Support
1	Perl	•	none		opt-in	•	opt-in
2	Perl	•	none	•	none	•	none
5	Perl	•	none	•	none		opt-out
3	Java		manual		opt-out	•	none
4	Java		always on		opt-in	•	opt-in
9	Java	•	none		opt-in		none
6	PHP	•	none		opt-out	•	opt-in
7	PHP	•	none		opt-out	•	none
8	PHP	•	none		opt-out	•	opt-in

Table 5: Presence or absence of binary vulnerability classes, and framework support for preventing them.

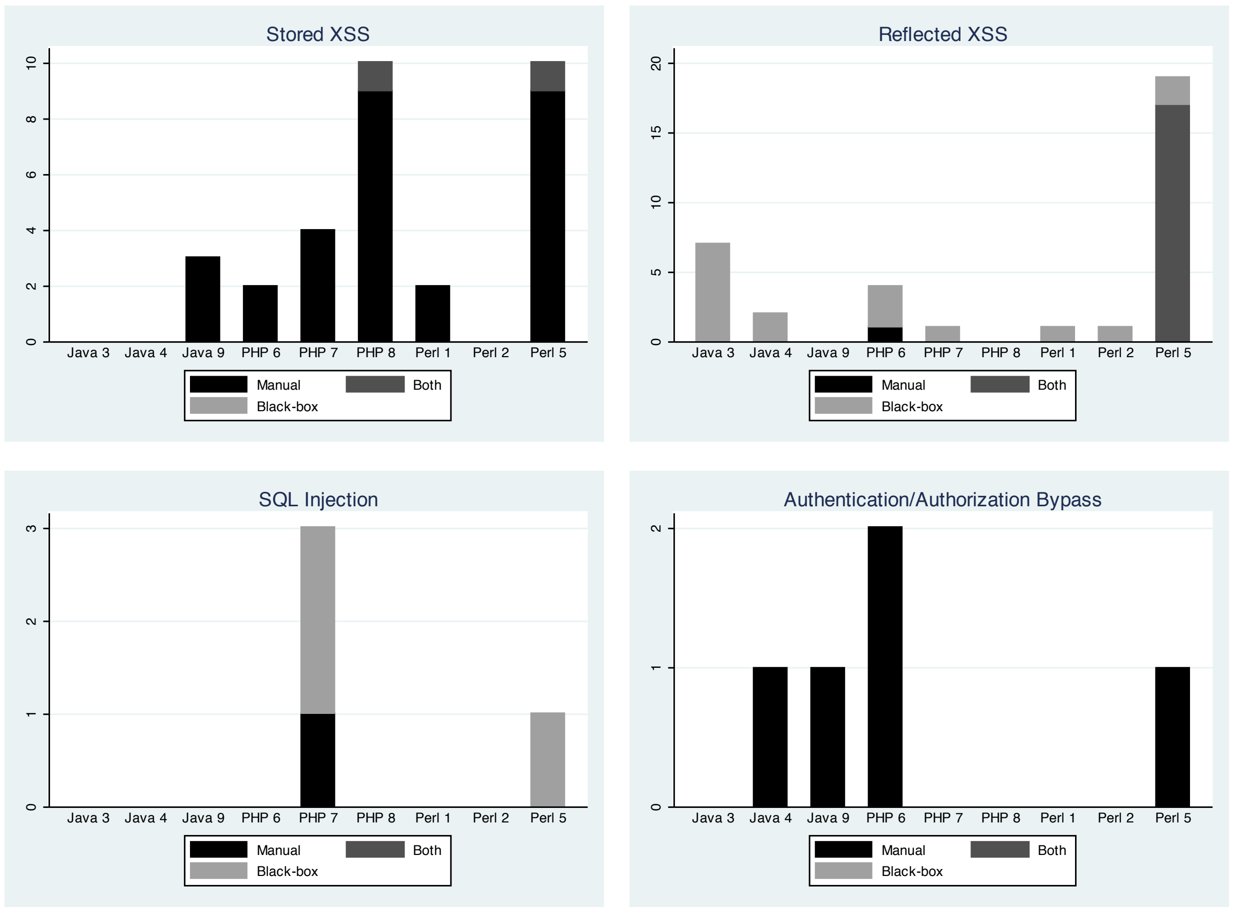


Figure 2: Vulnerabilities by vulnerability class.

SQL injection. Very few SQL injection vulnerabilities were found. Only two implementations had any such vulnerabilities, and only 4 were found in total. The difference between languages is not statistically significant ($F = 0.70$, $p = 0.5330$).

Authentication and authorization bypass. No such vulnerabilities were found in 5 of the 9 implementations. Each of the other 4 had only 1 or 2 such vulnerabilities. The difference between languages is not statistically significant ($F = 0.17$, $p = 0.8503$).

CSRF. As seen in Table 5, all of the PHP and Perl implementations, and 1 of 3 Java implementations were vulnerable to CSRF attacks. Fisher's exact test reveals that the difference between languages is not statistically significant ($p = 0.25$).

Session management. All implementations other than 2 of the 3 Perl implementations were found to implement secure session management. That is, the Perl implementations were the only ones with vulnerable session management. Fisher's exact test reveals that the difference is not statistically significant ($p = 0.25$).

		Vulnerabilities found by			
Team Number	**Language**	**Manual only**	**Black-box only**	**Both**	**Total**
1	Perl	4	1	0	5
2	Perl	3	1	0	4
5	Perl	12	3	18	33
3	Java	1	7	0	8
4	Java	2	2	0	4
9	Java	5	0	0	5
6	PHP	7	3	0	10
7	PHP	7	3	0	10
8	PHP	11	0	1	12

Table 6: Number of vulnerabilities found in the implementations of People by Temperament. The "Vulnerabilities found by" columns display the number of vulnerabilities found only by manual analysis, only by black-box testing, and by both techniques, respectively. The final column displays the total number of vulnerabilities found in each implementation.

Insecure password storage. Most of the implementations used some form of insecure password storage, ranging from storing passwords in plaintext to not using a salt before hashing the passwords. One Perl and one Java implementation did not violate current best practices for password storage. There does not, however, appear to be any association between programming language and insecure password storage. Fisher's exact test does not find a statistically significant difference ($p = 0.999$).

4.3 Manual review vs. black-box testing

Table 6 and Figure 1 list how many vulnerabilities were found only by manual analysis, only by black-box testing, and by both techniques. All vulnerabilities in the binary vulnerability classes were found by manual review, and none were found by black-box testing.

We observe that manual analysis fared better overall, finding 71 vulnerabilities (including the binary vulnerability classes), while black-box testing found only 39. We also observe that there is very little overlap between the two techniques; the two techniques find different vulnerabilities. Out of a total of 91 vulnerabilities found by either technique, only 19 were found by both techniques (see Figure 3). This suggests that they are complementary, and that it may make sense for organizations to use both.

Organizations commonly use only black-box testing. These results suggest that on a smaller budget, this practice makes sense because either technique will find some vulnerabilities that the other will miss. If, however, an organization can afford the cost of manual review, *it should supplement this with black-box testing.* The cost is small relative to that of review, and our results suggest that black-box testing will find additional vulnerabilities.

Figure 2 reveals that the effectiveness of the two techniques differs depending upon the vulnerability class.

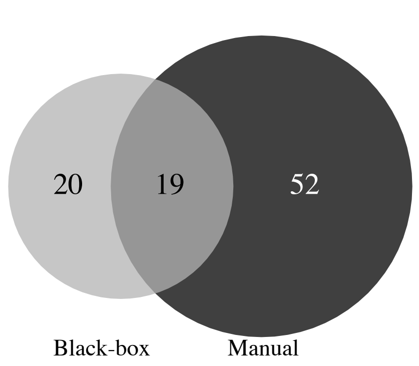


Figure 3: Vulnerabilities found by manual analysis and black-box penetration testing.

Manual review is the clear winner for authentication and authorization bypass and stored XSS vulnerabilities, while black-box testing finds more reflected XSS and SQL injection vulnerabilities. This motivates the need for further research and development of better black-box penetration testing techniques for stored XSS and authentication and authorization bypass vulnerabilities. We note that recent research has made progress toward finding authentication and authorization bypass vulnerabilities [9, 13], but these are not black-box techniques.

Reviewer ability. We now discuss the 20 vulnerabilities that were not found manually. Our analysis of these vulnerabilities further supports our conclusion that black-box testing complements manual review.

For 40% (8) of these, the reviewer found at least one similar vulnerability in the same implementation. That is, there is evidence that the reviewer had the skills and knowledge required to identify these vulnerabilities, but overlooked them. This suggests that we cannot expect a reviewer to have the consistency of an automated tool.

For another 40%, the vulnerability detected by the tool was in framework code, which was not analyzed by the reviewer. An automated tool may find vulnerabilities that reviewers are not even looking for.

The remaining 20% (4) represent vulnerabilities for which no similar vulnerabilities were found by the reviewer in the same implementation. It is possible that the reviewer lacked the necessary skills or knowledge to find these vulnerabilities.

4.4 Framework support

We examine whether stronger framework support is associated with fewer vulnerabilities. Figure 4 displays the relationship for each integer-valued vulnerability class between the level of framework support for that class and the number of vulnerabilities in that class. If for some vulnerability class there were an association between the level of framework support and the number of vulnerabil-

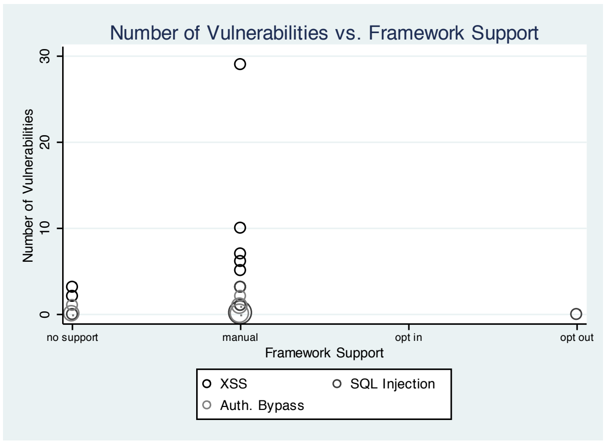


Figure 4: Level of framework support vs. number of vulnerabilities for integer-valued vulnerability classes. The area of a mark scales with the number of observations at its center.

ities, we would expect most of the points to be clustered around (or below) a line with a negative slope.

For each of the three[4] classes, we performed a one-way ANOVA test between framework support for the vulnerability class and number of vulnerabilities in the class. None of these results are statistically significant.

Our data set allows us to compare only frameworks with no support to frameworks with manual support because the implementations in our data set do not use frameworks with stronger support (with one exception). We found no significant difference between these levels of support. However, this data set does not allow us to examine the effect of opt-in, opt-out, or always-on support on vulnerability rates. In future work, we would like to analyze implementations that use frameworks with stronger support for these vulnerability classes. Example frameworks include CodeIgniter's `xss_clean` [1], Google Ctemplate [3], and Django's `autoescape` [2], all of which provide opt-out support for preventing XSS vulnerabilities. A more diverse data set might reveal relationships that cannot be gleaned from our current data.

Table 5 displays the relationship between framework support and vulnerability status for each of the binary vulnerability classes.

There does not appear to be any relationship for password storage. Many of the implementations use frameworks that provide opt-in support for secure password storage, but they do not use this support and are therefore vulnerable anyway. This highlights the fact that manual framework support is only as good as developers' awareness of its existence.

Session management and CSRF do, on the other hand, appear to be in such a relationship. Only the two implementations that lack framework support for session management have vulnerable session management. Similarly, only the two implementations that have framework support for CSRF were not found to be vulnerable to CSRF attacks. Both results were found to be statistically significant using Fisher's exact test ($p = 0.028$ for each).

[4]The level of framework support for stored XSS and reflected XSS is identical in each implementation, so we combined these two classes.

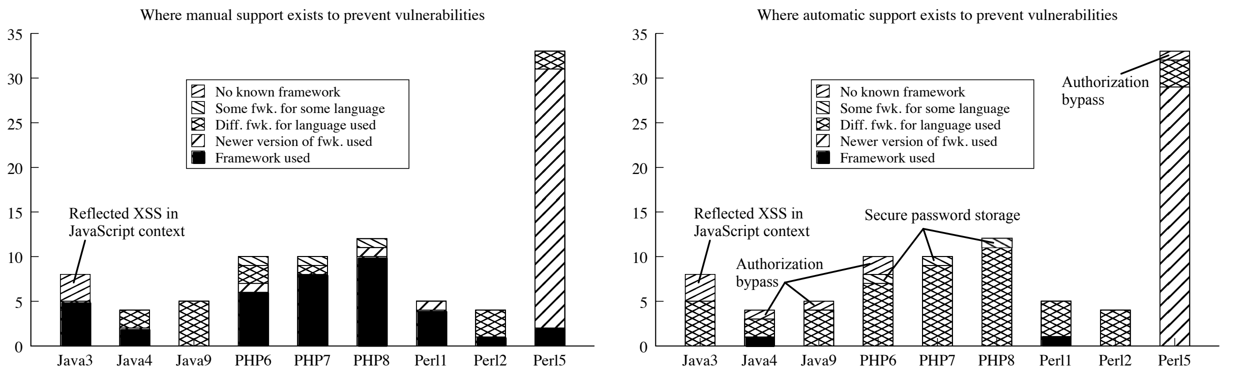
The difference in results between the integer-valued and binary vulnerability classes suggests that manual support does not provide much protection, while more automated support is effective at preventing vulnerabilities. During our manual source code review, we frequently observed that developers were able to correctly use manual support mechanisms in some places, but they forgot or neglected to do so in other places.

Figure 5 presents the results from our identification of the lowest level at which framework support exists that could have prevented each individual vulnerability (as described in Section 3.4).

It is rare for developers not to use available automatic support (the darkest bars in Figure 5b show only 2 such vulnerabilities), but they commonly fail to use existing manual support (the darkest bars in Figure 5a, 37 vulnerabilities). In many cases (30 of the 91 vulnerabilities found), the existing manual support was correctly used elsewhere. This suggests that no matter how good manual defenses are, they will never be good enough; developers can forget to use even the best manual framework support, even when it is evident that they are aware of it and know how to use it correctly.

For both manual and automatic support, the majority of vulnerabilities could have been prevented by support from another framework for the same language that the implementation used. That is, it appears that strong framework support exists for most vulnerability classes for each language in this study.

The annotations in Figure 5 point out particular shortcomings of frameworks for different vulnerability classes. We did not find any framework that provides any level of support for sanitizing untrusted output in a JavaScript context, which Team 3 failed to do repeatedly, leading to 3 reflected XSS vulnerabilities. We were also unable to find a PHP framework that offers automatic support for secure password storage, though we were able to find many tutorials on how to correctly (but manually) salt and hash passwords in PHP. Finally, we are not aware of any automatic framework support for preventing authorization bypass vulnerabilities. Unlike the other vulnerability classes we consider, these require correct policies; in this sense, this vulnerability class is fundamentally different, and harder to tackle, as acknowledged by recent work [9, 13].



(a) Manual framework support *(b)* Automatic framework support

Figure 5: For each vulnerability found, how far developers would have to stray from the technologies they used in order to find framework support that could have prevented each vulnerability, either manually (left) or automatically (right).

4.5 Limitations of statistical analysis

We caution the reader against drawing strong, generalizable conclusions from our statistical analysis, and we view even our strongest results as merely suggestive but not conclusive. Although we entered this study with specific goals and hypotheses (as described in Section 2), results that appear statistically significant may not in fact be valid – they could be due to random chance.

When testing 20 hypotheses at a 0.05 significance level, we expect one of them to appear significant purely by chance. We tested 19 hypotheses in this study, and 3 of them appeared to be significant. Therefore, we should not be surprised if one or two of these seemingly-significant associations are in fact spurious and due solely to chance. We believe more powerful studies with larger data sets are needed to convincingly confirm the apparent associations we have found.

5 Related work

In this section, we survey related work, which falls into 3 categories: (1) studies of the relationship between programming languages and application security, (2) comparisons of the effectiveness of different automated black-box web application penetration testing tools, and (3) comparisons of different bug- and vulnerability-finding techniques.

Programming languages and security. The 9th edition of the WhiteHat Website Security Statistic Report [26] offers what we believe is the best insight to date regarding the relationship between programming language and application security. Their data set, which includes over 1,500 web applications and over 20,000 vulnerabilities, was gathered from the penetration-testing service WhiteHat performs for its clients.

Their report found differences between languages in the prevalence of different vulnerability classes as well as the average number of "serious" vulnerabilities over the lifetime of the applications. For example, in their sample of applications, 57% of the vulnerabilities in JSP applications were XSS vulnerabilities, while only 52% of the vulnerabilities in Perl applications were XSS vulnerabilities. Another finding was that PHP applications were found to have an average of 26.6 vulnerabilities over their lifetime, while Perl applications had 44.8 and JSP applications had 25.8. The report makes no mention of statistical significance, but given the size of their data set, one can expect all of their findings to be statistically significant (though not necessarily practically significant).

Walden et al. [25] measured the vulnerability density of the source code of 14 PHP and 11 Java applications, using different static analysis tools for each set. They found that the Java applications had lower vulnerability density than the PHP applications, but the result was not statistically significant.

While these analyses sample across distinct applications, ours samples across implementations of the same application. Our data set is smaller, but its collection was more controlled. The first study focused on fixed combinations of programming language and framework (e.g., Java JSP), and the second did not include a framework comparison. Our study focuses separately on language and framework.

Dwarampudi et al. [12] compiled a fairly comprehensive list of pros and cons of the offerings of several different programming languages with respect to many language features, including security. No experiment or data analysis were performed as a part of this effort.

Finally, the Plat_Forms [19] study (from which the present study acquired its data) performed a shallow security analysis of the data set. They ran simple black-box tests against the implementations in order to find indications of errors or vulnerabilities, and they found minor differences. We greatly extended their study using both white- and black-box techniques to find vulnerabilities.

Automated black-box penetration testing. We are aware of three separate efforts to compare the effectiveness of different automated black-box web application security scanners. Suto [22] tested each scanner against the demonstration site of each other scanner and found differences in the effectiveness of the different tools. His report lists detailed pros and cons of using each tool based on his experience. Bau et al. [5] tested 8 different scanners in an effort to identify ways in which the state of the art of black box scanning could be improved. They found that the scanners tended to perform well on reflected XSS and (first-order) SQL injection vulnerabilities, but poorly on second-order vulnerabilities (e.g., stored XSS). We augment this finding with the result that manual analysis performs better for stored XSS, authentication and authorization bypass, CSRF, insecure session management, and insecure password storage, and black-box testing performs better for reflected XSS and SQL injection.

Doupé et al. [11] evaluated 11 scanners against a web application custom-designed to have many different crawling challenges and types of vulnerabilities. They found that the scanners were generally poor at crawling the site, they performed poorly against "logic" vulnerabilities (e.g., application-specific vulnerabilities, which often include authorization bypass vulnerabilities), and that they required their operators to have a lot of knowledge and training to be able to use them effectively.

While these studies compare several black-box tools to one another, we compare the effectiveness of a single black-box tool to that of manual source code analysis. Our choice regarding which black-box scanner to use was based in part on these studies.

Bug- and vulnerability-finding techniques. Wagner et al. [24] performed a case study against 5 applications in which they analyzed the true- and false-positive rates of three static bug-finding tools and compared manual source code review to static analysis for one of the 5 applications. This study focused on defects of any type, making no specific mention of security vulnerabilities. They found that all defects the static analysis tools discovered were also found by the manual review. Our study focuses specifically on security vulnerabilities in web applications, and we use a different type of tool in our study than they use in theirs.

Two short articles [8, 15] discuss differences between various tools one might consider using to find vulnerabilities in an application. The first lists constraints, pros, and cons of several tools, including source code analysis, dynamic analysis, and black-box scanners. The second article discusses differences between white- and black-box approaches to finding vulnerabilities.

6 Conclusion and future work

We have analyzed a data set of 9 implementations of the same web application to look for security differences associated with programming language, framework, and method of finding vulnerabilities. Each implementation had at least one vulnerability, which indicates that it is difficult to build a secure web application – even a small, well-defined one.

Our results provide little evidence that programming language plays a role in application security, but they do suggest that the level of framework support for security may influence application security, at least for some classes of vulnerabilities. Even the best manual support is likely not good enough; frameworks should provide automatic defenses if possible.

In future work, we would like to evaluate more modern frameworks that offer stronger support for preventing vulnerabilities. We are aware of several frameworks that provide automatic support for avoiding many types of XSS vulnerabilities.

We have found evidence that manual code review is more effective than black-box testing, but combining the two techniques is more effective than using either one by itself. We found that the two techniques fared differently for different classes of vulnerabilities. Black-box testing performed better for reflected XSS and SQL injection, while manual review performed better for stored XSS, authentication and authorization bypass, session management, CSRF, and insecure password storage. We believe these findings warrant future research with a larger data set, more reviewers, and more black-box tools.

We believe it will be valuable for future research to test the following hypotheses, which were generated from this exploratory study.

- *H1: The practical significance of the difference in security between applications that use different programming languages is negligible.* If true, programmers need not concern themselves with security when choosing which language to use (subject to the support offered by frameworks available for that language).
- *H2: Stronger, more automatic, framework support for vulnerabilities is associated with fewer vulnerabilities.* If true, recent advances in framework support for security have been beneficial, and research into more framework-provided protections should be pursued.
- *H3: Black-box penetration testing tools and manual source code review tend to find different sets of vulnerabilities.* If true, organizations can make more informed decisions regarding their strategy for vulnerability remediation.

We see no reason to limit ourselves to exploring these hypotheses in the context of web applications; they are equally interesting in the context of mobile applications, desktop applications, and network services.

Finally, we note that future work in this area may benefit from additional data sources, such as source code repositories. These rich data sets may help us answer questions about (e.g.,) developers' intentions or misunderstandings when introducing vulnerabilities and how vulnerabilities are introduced into applications over time. A deeper understanding of such issues will aid us in designing new tools and processes that will help developers write more secure software.

Acknowledgments

We thank Adrienne Felt, Erika Chin, and the anonymous reviewers for their thoughtful comments on earlier drafts of this paper. We also thank the Plat_Forms team for their hard work in putting together the Plat_Forms contest. This research was partially supported by National Science Foundation grants CNS-1018924 and CCF-0424422. Matthew Finifter was also supported by a National Science Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] CodeIgniter User Guide Version 1.7.3: Input Class. `http://codeigniter.com/user_guide/libraries/input.html`.

[2] django: Built-in template tags and filters. `http://docs.djangoproject.com/en/dev/ref/templates/builtins`.

[3] google-ctemplate. `http://code.google.com/p/google-ctemplate/`.

[4] perl.org glossary. `http://faq.perl.org/glossary.html#TMTOWTDI`.

[5] BAU, J., BURSZTEIN, E., GUPTA, D., AND MITCHELL, J. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 332–345.

[6] BISHOP, M. *Computer Security: Art and Science*. Addison-Wesley Professional, Boston, MA, 2003.

[7] COHEN, J. *Best Kept Secrets of Peer Code Review*. Smart Bear, Inc., Austin, TX, 2006, p. 117.

[8] CURPHEY, M., AND ARAUJO, R. Web application security assessment tools. *IEEE Security and Privacy 4* (2006), 32–41.

[9] DALTON, M., KOZYRAKIS, C., AND ZELDOVICH, N. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *USENIX Security Symposium* (2009), USENIX Association, pp. 267–282.

[10] DONALD, K., VERVAET, E., AND STOYANCHEV, R. Spring Web Flow: Reference Documentation, October 2007. `http://static.springsource.org/spring-webflow/docs/1.0.x/reference/index.html`.

[11] DOUPÉ, A., COVA, M., AND VIGNA, G. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)* (Bonn, Germany, July 2010).

[12] DWARAMPUDI, V., DHILLON, S. S., SHAH, J., SEBASTIAN, N. J., AND KANIGICHARLA, N. S. Comparative study of the Pros and Cons of Programming languages: Java, Scala, C++, Haskell, VB.NET, AspectJ, Perl, Ruby, PHP & Scheme. `http://arxiv.org/pdf/1008.3431`.

[13] FELMETSGER, V., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium* (Washington, DC, August 2010).

[14] JASPAL. The best web development frameworks, June 2010. `http://www.webdesignish.com/the-best-web-development-frameworks.html`.

[15] MCGRAW, G., AND POTTER, B. Software security testing. *IEEE Security and Privacy 2* (2004), 81–85.

[16] PETERS, T. PEP 20 – The Zen of Python. `http://www.python.org/dev/peps/pep-0020/`.

[17] PORTSWIGGER LTD. Burp Suite Professional. `http://www.portswigger.net/burp/editions.html`.

[18] PRECHELT, L. Plat_Forms 2007 task: PbT. Tech. Rep. TR-B-07-10, Freie Universität Berlin, Institut für Informatik, Germany, January 2007.

[19] PRECHELT, L. Plat_Forms: A Web Development Platform Comparison by an Exploratory Experiment Searching for Emergent Platform Properties. *IEEE Transactions on Software Engineering 99* (2010).

[20] ROBERTSON, W., AND VIGNA, G. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium* (Montreal, Canada, August 2009).

[21] SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium* (2001), pp. 201–220.

[22] SUTO, L. Analyzing the Accuracy and Time Costs of Web Application Security Scanners, February 2010. `http://www.ntobjectives.com/files/Accuracy_and_Time_Costs_of_Web_App_Scanners.pdf`.

[23] THIEL, F. Personal Communication, November 2009.

[24] WAGNER, S., JRJENS, J., KOLLER, C., TRISCHBERGER, P., AND MNCHEN, T. U. Comparing bug finding tools with reviews and tests. In *In Proc. 17th International Conference on Testing of Communicating Systems (TestCom05), volume 3502 of LNCS* (2005), Springer, pp. 40–55.

[25] WALDEN, J., DOYLE, M., LENHOF, R., AND MURRAY, J. Java vs. PHP: Security Implications of Language Choice for Web Applications. In *International Symposium on Engineering Secure Software and Systems (ESSoS)* (February 2010).

[26] WHITEHAT SECURITY. WhiteHat Website Security Statistic Report: 9th Edition, May 2010. `http://www.whitehatsec.com/home/resource/stats.html`.

Integrating Long Polling with an MVC Web Framework

Eric Stratmann, John Ousterhout, and Sameer Madan
Department of Computer Science
Stanford University
{*estrat,ouster,sameer27*}*@cs.stanford.edu*

Abstract

Long polling is a technique that simulates server push using Ajax requests, allowing Web pages to be updated quickly in response to server events. Unfortunately, existing long poll approaches are difficult to use, do not work well with server-side frameworks based on the Model-View-Controller (MVC) pattern, and are not scalable. Vault is an architecture for long polling that integrates cleanly with MVC frameworks and scales for clustered environments of hundreds of application servers. Vault lets developers focus on writing application-specific code without worrying about the details of how long polling is implemented. We have implemented Vault in two different Web frameworks.

1 Introduction

In its earliest days the Web supported only static content, but over the years it has evolved to support a variety of dynamic effects that allow Web pages to interact with users and update themselves incrementally on the fly. In the past most of these updates have been user-driven: they occurred in response to user interactions such as mouse clicks and keystrokes. However, in recent years more and more applications require *server push*, where Web pages update without any user involvement in response to events occurring remotely on a Web server or even other browsers. Server push is particularly important for collaboration applications where users want to be notified immediately when other users perform actions such as typing messages or modifying a shared document. Server push is also useful for a variety of monitoring applications.

The most common way to implement server push today is with a mechanism called *long polling*. Unfortunately, long polling is not easy to use. Existing implementations tend to be special-purpose and application-specific. They do not integrate well with Model-View-Controller (MVC) Web frameworks that are commonly used to build Web applications, often requiring separate servers just for long polling. Finally, it is challenging to create a long polling system that scales to handle large Web applications.

In this paper we describe an architecture for long polling called Vault, along with its implementation. Vault has three attractive properties:

- It integrates with MVC frameworks in a natural and simple fashion using a new construct called a *feed*, which is an extension of the model concept from MVC.
- It generalizes to support a variety of long polling applications (using different kinds of feeds), and to support multiple uses of long polling within a single Web page.
- It includes a scalable notification system that allows Vault to be used even for very large applications and that spares Web developers from having to worry about issues of distributed notification. There are two interesting architecture decisions in the design of the notification system. The first is that Vault separates notification from data storage, which makes it easy to create a scalable notifier and to incorporate a variety of data storage mechanisms into long polling. The second is that extraneous notifications are allowed in Vault, which helps simplify the implementation of the system and crash recovery.

Overall, Vault improves on existing work by making it easier to create Web applications that use long polling.

We have implemented Vault in two Web frameworks. We first built Vault in Fiz [10], a research framework, and then ported it to Django [3] to demonstrate the general applicability of Vault in MVC frameworks. This paper uses the Django implementation for examples, but the concepts and techniques could be used in most other MVC frameworks.

The remainder of this paper is organized as follows. We present background information on long polling and the problems it introduces in Section 2. Section 3 describes the major components of Vault and how they work together to implement long polling. Section 4 presents a few examples of feeds. Section 5 addresses the issue of long polling in a clustered environment with potentially thousands of application servers. Section 6 analyzes the performance of the Vault notification mechanism. We discuss integrating Vault with Web frameworks in Section 7. Section 8 describes the limitations of Vault, and Section 9 compares Vault with other approaches to long polling. Finally, Section 10 concludes.

2 Background

2.1 Long polling

The goal of long polling is to allow a Web server to initiate updates to existing Web pages at any time. The updates will reflect events occurring on the server, such as the arrival of a chat message, a change to a shared document, or a change in an instrument reading.

Unfortunately, the Web contains no mechanism for servers to initiate communication with browsers. All communication in the Web must be initiated by browsers: for example, a browser issues an HTTP request for a new Web page when the user clicks on a link, and Javascript running within a Web page can issue a request for additional information to update that page using mechanisms such as Ajax [8]. Web servers are generally *stateless*: once a server finishes processing a browser request it discards (almost) all information about that request. Although a server may retain a small amount of information about each active client (using *session* objects) the server typically doesn't retain the addresses of all its clients; even if it did there is no way for it to initiate a connection with the browser. A Web server can only respond to requests initiated by the browser.

Thus Web servers cannot update Web page content without browser assistance. A simple approach used by many early applications is polling: once a Web page is loaded, it issues Ajax requests to the server at regular intervals to check for updates. When an interesting event happens on the server it cannot immediately notify the browser; it must save information about the event until the next poll, at which point the Web page gets updated. Although polling can provide an appearance to the user much like true server push, it requires a trade-off between fidelity and efficiency. A short interval for polling provides high fidelity (events are reflected quickly on the browser) but wastes server resources and bandwidth responding to poll requests. A long interval for polling reduces overhead but may result in long delays before an event is reflected in the browser.

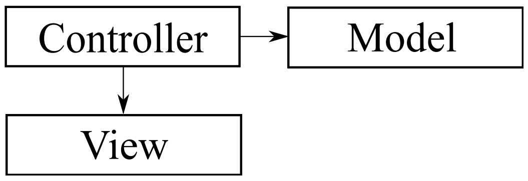


Figure 1: The Model-View-Controller pattern for Web applications. Each HTTP request is handled by a controller, which fetches data for the request from models and then invokes one or more views to render the Web page.

Long polling, also known as Comet [14] or reverse Ajax, is a variation on the polling approach that improves both fidelity and efficiency. The browser polls the server as described above, but if no updates are available the server does not complete the request. Instead it holds the Ajax request open for a period of time (typically tens of seconds). If an event occurs during this time then the server completes the request immediately and returns information about the event. If no event occurs for a long period of time the server eventually ends the request (or the browser times out the request). When this happens the browser immediately initiates a new long poll request. With long polling there is almost always an Ajax request active on the server, so events can be reflected rapidly to the browser. Since requests are held on the server for long periods of time, the overhead for initiating and completing requests is minimized.

However, long polling still has inefficiencies and complexities. It results in one open Ajax connection on the server for every active client, which can result in large numbers of open connections. Browsers typically limit the number of outstanding HTTP connections from a given page, which can complicate Web pages that wish to use long polling for several different elements. But the most challenging problem for long polling is that it does not integrate well with modern Web frameworks; these problems are introduced in the next section, and solving them is the topic of the rest of the paper.

2.2 Model-View-Controller Frameworks

Most of today's popular server-side Web frameworks are based on the Model-View-Controller (MVC) pattern [12] [13]. MVC divides Web applications into three kinds of components (see Figure 1): *models*, which manage the application's data; *views*, which render data into HTML and other forms required by the browser; and *controllers*, which provide glue between the other components as well as miscellaneous functions such as ensuring that users are logged in. When a request arrives at the Web server, the framework dispatches it to a method in a controller based on the URL. For example, a request for the URL `/students/display/47` might be dispatched to the `display` method in the `Student` controller. The controller fetches appropriate data from one or more models (e.g., data for the student whose id is 47), then invokes a view to render an HTML Web page that incorporates the data.

Unfortunately, MVC frameworks were not designed with long polling in mind. As a result, it is difficult to use long polling in Web applications today. Most frameworks assume that requests finish quickly so they bind a request tightly to a thread: the thread is occupied until the request completes. Some frameworks have only a single thread, which means there can be only one active request at a time; this can result in deadlock, since an active long poll request can prevent the server from handling another request that would generate the event to complete the long poll. If a framework is multi-threaded, it can use one thread for each active long poll request while processing other requests with additional threads. However, the presence of large numbers of threads can lead to performance problems. Fortunately, some frameworks (such as Tomcat 7.0) have recently added mechanisms for detaching a request from its thread, so that long poll requests can be put to sleep efficiently without occupying threads.

Another problem with MVC frameworks is that they were not designed for the flow of control that occurs in long polling. In traditional Web applications requests are relatively independent: some requests read information and pass it to the browser while other requests use information from the browser to update data, but there is no direct interaction between requests. With long polling, requests interact: one request may cause an event that is of interest to one or more active long polls; the framework must provide a mechanism for managing these events and moving information between requests. Notifications become even more complicated in clustered Web servers where an action on one Web server may impact long poll requests on other servers in the cluster.

Because of these problems, applications that need long polling typically use special-purpose solutions today. In many cases long polling is handled by different Web servers than the main application, with a special (non-MVC) framework used for the long poll servers (see Section 9.3). The internal mechanisms used for long polling are often application-specific, so that each long polling application must be built from scratch. For example, some implementations of long polling tie the notification mechanism to a particular data model such as message channels, which requires the notification mechanism to be reimplemented if a different data model is used.

Our goal for Vault was to create an architecture for long polling that integrates naturally with existing MVC frameworks, generalizes to support a variety of long polling applications and encourage code reuse, and provides reusable solutions for many of the problems shared across long polling applications such as managing the long poll protocol and creating a scalable notification system.

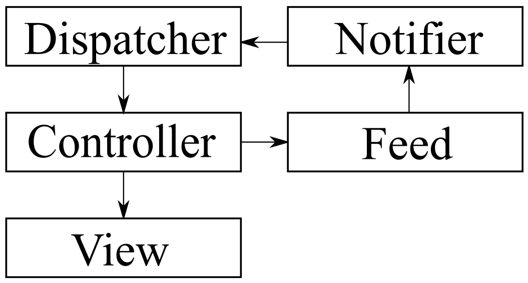


Figure 2: The architecture of Vault. Vault is similar to an MVC framework except models have been replaced with feeds and two new components have been added (the notifier and the dispatcher).

3 The Vault Architecture

Figure 2 shows the overall architecture of Vault. Vault extends an MVC framework with three additional elements: feeds, a notifier, and a dispatcher. A *feed* is similar to a model in a traditional MVC framework except that it provides extra capabilities for long polling. As with traditional models, a feed is responsible for managing a particular kind of data, such as a table in a database, a queue of messages, or an instrument reading. A traditional model answers the question *what is the current state of the data?* and also provides methods to modify the model's data, validate data using business logic, etc. A feed provides these same facilities, but in addition it contains methods to answer questions of the form *how has the data changed since the last time I asked?*. The new methods make it easy to create long polling applications. One of our goals for Vault is to make it as simple as possible to convert a model to a feed; we will illustrate this in Section 3.3.

One of the key issues for a long polling system is notification: when there is a change in the state of data (e.g. a value in a database is modified, or an instrument reading changes) there may be pending long poll requests that are interested in the change. The *notifier* is responsible for keeping track of long poll requests and waking them up when interesting events occur. Feeds tell the notifier which events are relevant for each long poll request, and they also tell the notifier when events have occurred; the notifier is responsible for matching interests with events.

logs.py

```
def show(request):
    logs = Logs.get_new_instances()
    return render_to_response("show.html", {"logs": logs})

def get_latest_logs(request):
    logs = Logs.get_new_instances()
    if logs.count() == 0:
        return
    return append_to_element("logs-div", "log-entries.html", {"logs": logs})
```

show.html

```
...
<% add_long_poll "logs/get_latest_logs" %>
<div id="logs-div">
    <% include "log-entries.html" %>
</div>
...
```

log-entries.html

```
<% for log in logs %>
    <p><%= log.type %>: <%= log.message %></p>
<% endfor %>
```

Figure 3: A simple application that uses Django and Vault to display a list of log entries. `logs.py` is a controller; the `show` method displays the overall page and `get_latest_logs` updates the page with new log entries during long-poll requests. `show.html` is a template that renders the main page, and `log-entries.html` is the template for rendering a list of log entries.

Vault notifications do not contain any data, they simply say that a change has occurred; it is up to feeds to verify exactly what data changed. The Vault notifier is scalable, in that it can support Web applications spanning many application servers: if a feed on one server announces an event to its local notifier, the notifier will propagate information about the event to all other servers that are interested in it. The notifier API is discussed in Section 3.2 and a scalable implementation is described in Section 5.

The third component of Vault is the long poll dispatcher. The dispatcher provides glue between Javascript running on Web pages, controllers, and the notifier. Among other things, it receives long poll requests and invokes controller methods at the right time(s) to complete those requests. Its behavior will be explained in Section 3.4.

Of these three components, feeds are the primary component visible to Web application developers. The dispatcher is almost completely invisible to developers, and the notifier is visible only to developers who create new feeds.

3.1 A Simple Example

This section illustrates how the Vault mechanism is used by a Web application developer (using Django as the framework), assuming that an appropriate feed already exists. Writing an application with Vault is similar to writing an application using MVC. Developers write controller methods that fetch data and use it to render a view. Although they must indicate to Vault that they would like to use long polling to update a page, most of the details of the long poll mechanism are handled invisibly by Vault.

The example page (see Figure 3) displays a list of log entries and uses long polling to update the page whenever a new log entry is added to a database. New log entries appear to be added to the page instantaneously. A user who opens the page will initially see a list of existing log entries, and new ones will show up when they are created. Log entries are assumed to be generated elsewhere.

The page (URL `/logs/show`) is generated by the `show` method. `show` has the same structure as normal MVC controller methods: it retrieves data from a model and then invokes a view to render a Web page using that data.

The page differs from traditional MVC pages in two ways. First, `show` invokes a new feed method called `get_new_instances` to retrieve its data; `get_new_instances` returns all of the current rows in the table and also makes arrangements for future notifications. Second, to arrange for automatic page updates, the view calls `add_long_poll`. This causes Vault to enable long polling for the page, and it indicates that

```
all() => [Record]
filter(condition) => [Record]
save() => void
get_new_instances(request) => [Record]
```

Figure 4: A few of the methods provided by DatabaseFeed. get_new_instances is the only method that would not be found in a regular model. [Record] indicates that the method returns an array of database records. The request object represents the state of the HTTP request currently being processed.

the method logs/get_new_logs should be invoked to update the page.

Vault arranges for the page to issue long-poll requests, and it invokes get_latest_logs during the requests. The purpose of this method is to update the page if new log entries have been created. get_latest_logs is similar to a method in an MVC framework for handling an Ajax request: it fetches data from a model, invokes a view to render that data, and returns it to the browser (append_to_element is a helper method that generates Javascript to append HTML rendered from a template to an element). However, it must ensure that it generates no output if there are no new log entries. This signals to Vault that the request is not complete yet. Vault then waits and invokes this method again when it thinks there may be new log entries.

The DatabaseFeed (see Figure 4) is used to determine if any new log entries have been created. Its get_new_instances method returns any new log entries that have been created since the last time the method was invoked for the current page, or an empty array if there are none. The first time it is invoked, it returns all existing log entries, as in the show method. All feeds have methods analogous to get_new_instances: these methods either return the latest data, if it has changed recently, or an indication that the data has not changed. Aside from this method, DatabaseFeed also has ordinary model methods such as all and filter, which return subsets of the data, and save, which writes new data to the database.

Vault automatically handles many of the details of long polling so that application developers do not have to deal with them. For example, the dispatcher automatically includes Javascript in the page to issue long-poll requests, the feed and notifier work together to determine when to invoke methods such as get_latest_logs, and the feed keeps track of which log entries have already been seen by the current page. These mechanisms will be described in upcoming sections.

```
create_interest(request, interest) => boolean
remove_interest(request, interest) => void
notify(interest) => void
```

Figure 5: The methods provided by the Vault notifier.

3.2 The Notifier

One of the most important elements of a long polling system is its notification mechanism, which allows long poll requests to find out when interesting events have occurred (such as a new row being added to a table). Most libraries for long polling combine notification with data storage, for example by building the long polling mechanism around a message-passing system. In contrast, Vault separates these two functions, with data storage managed by feeds and notification implemented by a separate notifier. This separation has several advantages. First, it allows a single notifier implementation to be shared by many feeds, which simplifies the development of new feeds. Second, it allows existing data storage mechanisms, such as relational databases, to be used in a long polling framework. Third, it allows the construction of a scalable notifier (described in Section 5) without having to build a scalable storage system as well.

The Vault notifier provides a general-purpose mechanism for keeping track of which long poll requests are interested in which events and notifying the dispatcher when those events occur (see Figure 5). Each event is described by a string called an *interest*. The create_interest method associates an interest with a particular HttpRequest (which represents the state of an HTTP request being processed by the server, usually a long poll request); the remove_interest method breaks the association, if one exists. The notify method will call the dispatcher to wake up all requests that have expressed interest in a particular event. As will be seen in Section 3.3, feeds are responsible for calling create_interest and notify. Interest names are chosen by feeds; they do not need to be unique, but the system will operate more efficiently if they are.

For Web applications that exist entirely on a single server machine the notifier implementation is quite simple, consisting of little more than a hash table to store all of the active interests. However, its implementation becomes more interesting for large Web sites where the sender and receiver of a notification may be on different server machines. Section 5 describes how a scalable distributed notifier can be implemented with the same API described by Figure 5.

Extraneous notifications are acceptable in Vault; they affect only the performance of the system, not its correctness. Extraneous notifications can happen in sev-

```
 1  class DatabaseFeed:
 2      def get_new_instances(self, request):
 3          interest = "DB-" + self.model.name
 4          possibly_ready = Notifier.create_interest(interest, request)
 5          if not possibly_ready:
 6              return []
 7
 8          old_largest_key = PageProperties.get(request, interest)
 9          if old_largest_key == None:
10              old_largest_key = 0
11          current_largest_key = self.get_largest_primary_key()
12
13          PageProperties.set(request, interest, current_largest_key)
14
15          if current_largest_key > old_largest_key:
16              latest = self.filter(primary_key__greater_than=old_largest_key)
17              return latest
18          else:
19              return []
20
21      def on_db_save(table_name, instance, is_new_instance):
22          if is_new_instance:
23              Notifier.notify("DB-" + table_name)
```

Figure 6: A partial implementation of `DatabaseFeed`, showing the code necessary to implement the `get_new_instances` method. `get_new_instances` records the largest primary key seen for each distinct Web page and uses that information in future calls to determine whether new rows have been added to the table since the page was last updated. For brevity, code for the other feed methods is not shown and a few minor liberties have been taken with the Django API, such as the arguments to `on_db_save`.

eral ways. First, it is possible for different feeds to choose the same string for their interests. As a result, an event in either feed will end up notifying both interests. In addition, extraneous notifications can happen when recovering from crashes in the cluster notifier (see Section 5). If an extraneous notification happens in the example of Section 3.1 it will cause the controller method `get_latest_logs` to be invoked again, even though no new rows have been created. The `get_new_instances` method will return an empty array in, so `get_latest_logs` will generate no output and the dispatcher will delay for another notification.

Vault interests are similar to condition variables as used in monitor-style synchronization [9]: a notification means “the event you were waiting for *probably* occurred, but you should check to be certain”. Vault interests can be thought of as a scalable and distributed implementation of condition variables, where each interest string corresponds to a distinct condition variable.

3.3 Implementing Feeds

A feed is a superset of a traditional MVC model, with two additional features. First, it must maintain enough state so that it can tell exactly which data has changed (if any) since the last time it was invoked. Second, it must work with the Notifier to make sure that long poll requests are reawakened when relevant events occur (such as the addition of a row to a database) . The paragraphs below discuss these issues in detail, using the `get_new_instances` method of `DatabaseFeed` for illustration (see Figure 6).

In order to detect data changes, feeds must maintain state about the data they have already seen. For example, `get_new_instances` does this by keeping track of the largest primary key that has been seen so far (`old_largest_key`). In each call, it compares the largest key seen so far with the largest primary key in the database (lines 8-11). If the latter is larger, the feed returns all the new rows; otherwise it returns an empty array. The type of state will very from feed to feed, but a few possibilities are a primary key of a table, the contents of a row, or a timestamp.

Because a user many have several pages open, state such as the largest key seen must be page specific. For example, a user might have the same URL opened in two different browser tabs; they must each update when new rows are created. Vault accomplishes this through the use of *page properties*, which are key-value pairs stored on the server but associated with a particular Web page. Page properties can be thought of as session data at the page level instead of the user or browser level. If a page property is set during one request associated with a particular page (such as when `get_new_instances` is invoked by the `show` method of Figure 3 during the initial page rendering), it will be visible and modifiable in any future requests associated with the same page (such as when get_new_instances is invoked by get_latest_logs during a subsequent long poll request). The DatabaseFeed uses page properties to store the largest primary key seen by the current page. For details on how page properties are implemented, see [11]. The Fiz implementation of Vault uses an existing page property mechanism provided by Fiz; Django does not include page properties, so we implemented page properties as part of Vault.

The second major responsibility for feeds is to communicate with the notifier. This involves two actions: notification and expressing interest. Notification occurs whenever any operation that modifies data is performed, such as creating a new row or modifying an existing row. For example, on_db_save, which is invoked by Django after any row in the table has been modified, calls Notifier.notify (line 23) to wake up requests interested in additions to the table. Expressing an interest occurs whenever any feed method is invoked. get_new_instances invokes Notifier.create_interest to arrange for notification if/when new rows are added to the table. The interest must be created at the beginning of the method, *before* checking to see whether there are any new instances: if the interest were created afterward, there would be a race condition where a new row could be added by a different thread after the check but before the interest was created, in which case the notification would be missed.

The DatabaseFeed uses interests of the form "DB-*ttt*", where *ttt* is the name of the table. This ensures that only requests interested in that table are notified when the new rows are added to the table.

Note that Notifier.remove_interest is not invoked in Figure 6. The feed intentionally allows the interest to persist across long poll requests, so that notifications occurring between requests will be noticed. Old interests are eventually garbage collected by the notifier.

Feeds can take advantage of an additional feature of the notification mechanism in order to eliminate unnecessary work. In many cases the notifier has enough information to tell the feed that there is no chance that the feed's data has changed (e.g., if the interest has existed since the last call to the feed and there has been no intervening notification). In this case create_interest returns false and get_new_instances can return immediately without even checking its data. This optimization often prevents a costly operation, such as reading from disk or querying a remote server. Furthermore, most of the time when the feed is invoked there will have been no change (for example, the first check for each long poll request is likely to fail).

All feeds have methods with the same general structure as get_new_instances. In each case the method must first create one or more interests, then check to see if relevant information has changed. The exact checks may vary from feed to feed, but they will all record information using page properties in order to detect when the information on the page becomes out of date. For example, a feed that implements messages with serial numbers might store the serial number of the last message seen by this page and compare it with the serial number of the most recent message.

3.4 The Dispatcher

The Vault dispatcher hides the details of the long poll protocol and supervises the execution of long poll requests. The dispatcher is invisible to Web application developers except for its add_long_poll method, which is invoked by views to include long polling in a Web page (see Figure 3). When add_long_poll is invoked the dispatcher adds Javascript for long polling to the current Web page. This Javascript will start up as soon as the page has loaded and initiate an Ajax request for long polling back to the server. When the long poll request completes the Javascript will process its results and then immediately initiate another long poll request. Only one long poll request is outstanding at a time, no matter how many long poll methods have been declared for the page.

Vault arranges for long poll requests to be handled by the dispatcher when they arrive on the Web server. In the Django implementation this is done by running Vault as middleware that checks each request for a special long poll URL, and if present sends the request to the dispatcher. The dispatcher finds all of the long poll methods for the current Web page (add_long_poll records this information using the page property mechanism described in Section 3.3) and invokes all of them in turn. If output was generated by any of the long poll methods then the dispatcher returns the output to the browser and completes the request. If none of the methods generated any output then the dispatcher puts the request to sleep until it receives a notification for this request from the notifier. Once a notification is received the dispatcher invokes all of the long poll methods again; once again, it either returns (if output has been generated) or puts the request to sleep.

In normal use it is unlikely that the first invocation of the long poll methods during a request will generate any output, since only a short time has elapsed since the last invocation of those methods. However, it is important to invoke all of the methods, which in turn invoke feeds, in order to ensure that interests have been created for all of the relevant events.

```
create_channel(channel_name) => void
create_user(user_name) => void
get_new_messages(request, user_name) => [Message]
subscribe(user_name, channel_name) => void
post_message(channel_name, message) => void
```

Figure 7: The methods provided by our example Messaging feed .

4 Feed examples

In order to evaluate the architecture of Vault we have implemented two feeds; these feeds illustrate the approaches that we think will be most common in actual Web applications.

The first feed is the `DatabaseFeed` discussed earlier in 3. We implemented two feed methods, `get_new_instances` and `get_modified_instance`, and designed a third, `get_attribute_changes`.

`get_new_instances` has been discussed earlier in Section 3.1. Briefly, `get_new_instances` returns new rows from a database. It uses interests of the form `DB-`*ttt*, where *ttt* is the name of the table. `get_new_instances` detects new rows by saving the highest primary key seen in the past for any row in the table and comparing this with the current highest key in future calls (it assumes that primary keys are chosen in ascending order).

`get_modified_instance` allows a Web page to observe changes to a particular row; it returns the latest value of the row, or `None` if the row has not changed since the last call for this row on the current Web page. It uses interests of the form `DB-`*ttt*`-`*rrr*, where *ttt* is the name of the table and *rrr* is the primary key of the row under observation; all methods in the model that modify existing rows must notify the interest for that row. The implementation of `get_modified_instance` is similar to that of `get_new_instances` in Figure 6: it records the last value of the row seen by the current Web page and compares this against the current contents of the row in future calls.

Although these two methods are simple to implement, one can imagine more complicated methods that are not as easy to implement in Vault. One example is a method to monitor changes to a column (or *attribute* in model terminology), called `get_attribute_changes`. If any values in the column have changed since the last invocation for this table in the current Web page the primary keys for the affected rows are returned. It is difficult for the feed to tell whether a column has changed recently unless it records a copy of the entire column, which would be impractical for large tables. One solution is to create an auxiliary data structure to keep track of changes. A new table can be added to the database with each row representing one change to the given column: the auxiliary row contains the primary key of the row in the main table that changed. This table will need to be updated any time the model modifies a field in the column under observation: once the auxiliary table exists, an approach similar to `get_new_instances` can be used to implement `get_attribute_changes`. The method stores the largest primary key (of the auxiliary table) used by previous method calls and queries the table for all rows with a larger primary key.

The `DatabaseFeed` described above introduces overhead to notify various interests when relevant changes occur in the database; in some cases a single modification to the database might require notification of several different interests (e.g., modifying a row would notify both that the row was modified and any column changes). However, as will be seen in Section 6, notifying an interest is considerably faster than typical database operations. Thus we think a variety of interesting database feeds can be implemented with minimal overhead.

Our second feed implements a publish-subscribe messaging protocol (see Figure 7) somewhat like Bayeux [1] (see Section 9.1). It contains a collection of channels on which text messages can be posted, and each user can subscribe to channels that are of interest to that user. The message feed provides a method `receive_messages` that is analogous to `get_new_instances` in `DatabaseFeed`: it returns all the messages that have arrived for a given user since the last call to this method for the current Web page. The feed uses a separate interest for each user with names of the form `message-`*uuu*, where *uuu* is an identifier for a particular user. `receive_messages` creates an interest for the current user, and `post_message` notifies all of the users that have subscribed to the channel on which the message is posted. In order to tell whether there are new messages for a user, the feed serializes all of the messages for each user and records the serial number of the most recent message that has been delivered to each Web page.

We have created a simple application using the message feed that provides Web pages for posting messages, subscribing to channels, and observing incoming traffic for a particular user.

We believe that many feed implementations are

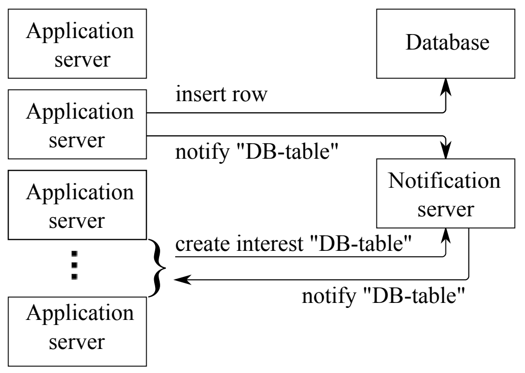


Figure 8: Cluster architecture. Notifications must go through notification servers where they are relayed to servers with matching interests.

likely to be similar to the ones above where they either (a) observe one or more individual values (or rows) like `get_new_instances` and `get_modified_instance`, (b) observe a sequence of messages or changes like `receive_messages` and `get_new_instances`, or (c) make more complex observations like `get_attribute_changes`, in which case they will create an auxiliary data structure in the form of a sequence.

5 Cluster Support and Scalability

Large-scale Web applications must use clusters of application servers to handle the required load; the cluster size can vary from a few servers to thousands. Typically, any application server is capable of handling any client request (though the networking infrastructure is often configured to deliver requests for a particular browser to the same server whenever possible); the data for the application is kept in storage that is shared among all of the servers, such as a database server.

Introducing long polling to a cluster environment complicates the notification mechanism because events originating on one server may need to propagate to other application servers. Many events will be of interest to only a single server (or may have no interests at all), while other events may be of interest to nearly every server.

One possible solution would be to broadcast all notifications to all application servers. This approach behaves correctly but does not scale well since every application server must process every notification: the notification workload of each server will increase as the total number of servers increases and the system will eventually reach a point where the servers are spending all of their time handling irrelevant notifications.

For Vault we implemented a more scalable solution using a separate *notification server* (see Figure 8). When an interest is created on a particular application server, that information gets forwarded to the notification server so that it knows which application servers care about which interests. When an interest is notified on a particular application server, the notification is forwarded to the notification server, which in turn notifies any other application servers that care about the interest. With this approach only the application servers that care about a particular interest are involved when that interest is notified. The notification server is it similar to the local notifier, but it works with interested servers, not interested requests. In particular, both have the same basic API (create interest, remove interest, and notify).

For large Web applications it may not be possible for a single notification server to handle all of the notification traffic. In this case multiple notification servers can be used, with each notification server handling a subset of all the interests. Local notifiers can use a consistent hash function [15] on each interest to determine which notification server to send it to.

One of the advantages of the API we have chosen for notification is that it distributes naturally as described above. Furthermore, the decision to allow extraneous notifications simplifies crash recovery and several other management issues for notification servers.

Crash recovery is simple in Vault due to these properties. If a notification server crashes, a new notification server can be started as its replacement. Each of the application servers can deal with the crash by first recreating all of its local interests on the new server and then notifying all of those interests locally (just in case a notification occurred while the original server was down). Most of the notifications will be extraneous but the feeds will detect that nothing has actually changed. This behavior is correct but may be slow depending on the number of interests.

There is an additional crash recovery issue in Vault, which occurs if a feed crashes after updating its data but before notifying the notification server. In Figure 6, this could happen if the server crashes on line 22. If an application server crashes, there is no way to tell whether it was about to send notifications. To avoid losing notifications, every existing interest must be notified whenever any application server crashes. Another alternative is to use a two-phase notification, but the associated overheard makes the first alternative a more attractive option.

6 Benchmarks

We ran a series of experiments to measure the performance and scalability of the Vault notification mechanism. Our intent was to determine how many servers are needed to handle a given load. We ran our experiments

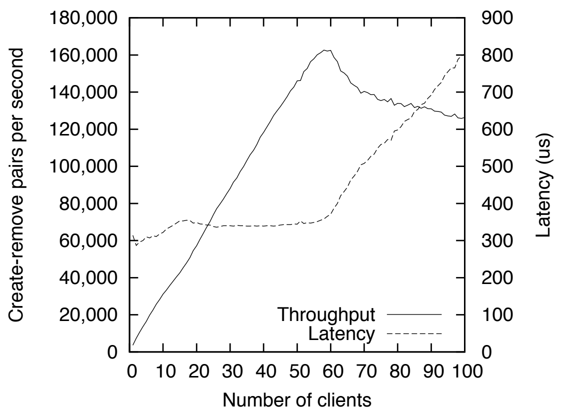


Figure 9: Performance measurements of the notification server for creating and removing interests. Each load generator repeatedly messages the notification server to create an interest and then remove the interest. Throughput measures the number of create-remove pairs handled by the notification server per second and latency measures the time from the start of creating the interest to the end of removing the interest.

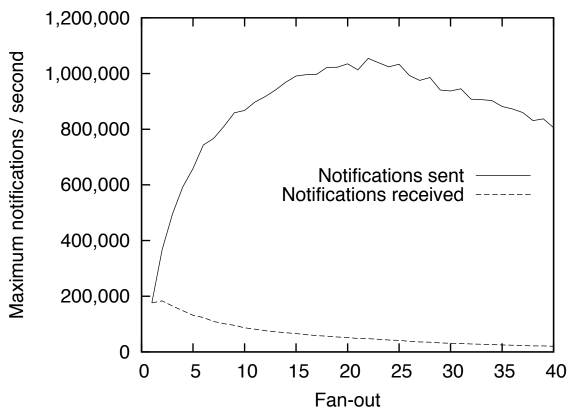


Figure 10: Performance of the notification server for processing notifications. Fan-out indicates the number of application servers that must ultimately receive each notification received by the notification server. The chart shows the maximum rate at which the server can receive notifications (measured by varying the number of issuing clients) and the maximum rate at which it sends them to individual application servers.

on 40 identical servers. Each one has a Xeon X3470 (4x2.93 GHz cores), 24GB of 800MHz DDR3 SDRAM, and an Intel e1000 NIC. For some benchmarks, the number of application servers is larger than the number of machines, in which case some machines run multiple processes (the impact on the measurements is negligible because the notification server bottlenecks before any of the application servers do).

Our first experiment was to determine the number of create-interest and remove-interest operations each notification server can handle per second. Creates are performed by an application server when a long poll creates an interest locally and the application server has not yet relayed the interest to the notification server. Removing an interest occurs when the application server determines that there are no longer any interested long polls. Creating and removing an interest may happen as frequently as several times per page request so they should be fast.

The experiment measured the total number create-remove pairs handled by the notification server per second and the latency from the start of the create to the end of the remove, as shown in Figure 9. Each load generator ran a loop sending a create message and waiting for the response and then sending a remove without waiting for the response. Our results show a linear increase in throughput as the number of load generators is increased. The throughput maximum is around 160,000 create-remove pairs per second, but drops about 10% after the maximum is reached. Latency remains roughly constant until the throughput limit is reached, at which point the latency begins to grow linearly.

Our second experiment measured the cost for the notification server to process events and relay them to interested application servers. We measured the maximum throughput for the notification server with different fanouts per notification, as seen in Figure 10. The figure shows the total number of notifications received by the notification server per second and the number of notifications it passes on to application servers. The notifications received starts at just under 200,000 per second with one interested server, then drops roughly linearly as the fanout increases. The total number of notifications sent rises as fan-out increases, peaking at around 1,000,000 notifications per second with a fan-out of 15-20. As the number of servers continues to increase, the total number of notifications sent drops to around 800,000 per second.

Figure 10 does not contain a data point for interests with a fan-out of zero. For notifications where no server is interested, the notification server can process about 2,000,000 per second. Notification servers may get many notifications for which there are no interested servers so it is important for them to be quick.

7 Framework integration

Although the principles of Vault are applicable to any MVC framework, integrating Vault with a Web framework requires some basic level of support from the framework. We have identified two such requirements: the ability to decouple requests from their threads, and page properties.

Many existing frameworks do not currently allow requests to be put to sleep. For example, Django does not support this, so we modified the Django Web server to implement request detaching. Our initial implementation in Fiz was easier because Fiz is based on Apache Tomcat 7.0, which provides a mechanism to *detach* a request from the thread that is executing it, so that the thread can return without ending the request. The Vault dispatcher saves information about the detached request and then returns so its thread can process other requests (one of which might generate a notification needed by the detached request). When a notification occurs the dispatcher retrieves the detached request and continues processing it as described above.

If a framework does not allow requests to be detached but does support multi-threading, then the dispatcher can use synchronization mechanisms to put the long poll thread to sleep and wake it up later when a notification occurs. However, this means that each waiting long poll request occupies a thread, which could result in hundreds or thousands of sleeping threads for a busy Web server. Unfortunately, many systems suffer performance degradation when large numbers of threads are sleeping, which could limit the viability of the approach.

If a framework only allows a single active request and does not support request detaching, then the framework cannot support long polling; applications will have to use a traditional polling approach. However, Vault could still be used in such a framework and most of Vault's benefits would still apply.

Secondly, a framework must provide page properties or some equivalent mechanism to associate data with a particular Web page and make that data available in future requests emanating from the page. If a framework does not support page properties, they can be built into Vault on top of existing session facilities (a separate object for each page can be created within the session, identified with a unique identifier that is recorded in the page and returned in long poll requests).

8 Limitations

We are aware of two limitations in the Vault architecture. The first limitation is that some styles of notification do not map immediately onto interest strings, such as `get_attribute_changes` in Section 4. We believe these situations can be handled with the creation of auxiliary data structures as described in Section 4, but the creation of these structures may affect the complexity and efficiency of the application. There may be some applications where a better solution could be achieved with a richer data model built into the notification mechanism.

The second limitation of Vault is that the crash recovery mechanism can produce large numbers of extraneous notifications, as described in Section 5. We have experimented with alternatives that reduce the extraneous notifications, but they result in extra work during normal operation, which seems worse than the current overheads during crash recovery.

9 Related Work

9.1 CometD/Bayeux

CometD [2] is the most well known long polling system. It is an implementation of Bayeux [1], a protocol intended primarily for bidirectional interactions between Web clients. Bayeux communication is done through channels and is based on a publish/subscribe model. Clients can post messages to a channel, and other clients subscribed to the channel receive the messages.

CometD works well for publish/subscribe communication but does not generalize to other uses (for example, there is no way to implement database notifications using CometD). Since CometD is a stand-alone framework, it does not work with existing MVC frameworks. By tying the protocol to the message-based model used, CometD limits the ability of developers to write applications using other data models. In addition, the current implementation of CometD does not seem to have a mature cluster implementation.

9.2 WebSockets

The WebSocket [7] API is a proposed addition to Web browsers that allows for bi-directional communication between the browser and the server. Although WebSockets are often seen as a "solution" to long polling, they do not fix any of the major issues associated with long polling. Notification, scalability, and integration with MVC will still be issues with WebSockets. WebSockets only fix some minor inconveniences such as the need to hold Ajax requests at the server and some of the associated complexity of long polling. If widely adopted (which is likely when HTML5 becomes supported by all browsers) WebSockets could become the new transport mechanism for Vault, in which case Vault would not have to worry about Ajax requests timing out or the need to wait for an Ajax request to return from the browser to send another update.

9.3 Event-based Web Frameworks

Due to the difficulty of using long polling in traditional Web frameworks, event-based frameworks have become popular for handling long polling. This is typically done by running two Web frameworks side-by-side, one to handle normal requests and an event-based one to handle long polls, with the two communicating though some backend channel. This approach is easier than trying to include long polling in existing MVC frameworks but is not as clean as keeping all application logic in one framework.

Event-based Web frameworks differ from other frameworks because they only run one thread at a time, eliminating the thread issues traditional MVC frameworks have when implementing long polling. Since there is only one thread, it is important that it does not block. If an expensive operation is performed, such as disk or network access, a callback is specified by the caller and the server stops processing the current request and starts processing a different request. When the operation finishes, the server runs the callback, passing in values generated by the operation. This style of programming makes it easy to resume a request at a later time as is required with Vault. Node.js [5] is an javascript event-based IO library used to write server-side javascript applications. Tornado [6] is a similar Web server for Python, based on FriendFeed [4].

10 Conclusion

Long polling allows for new interactive Web applications that respond quickly to external events. Existing implementations, however, have not been general or scalable, and they require too much glue code for application developers. We have presented a modification of the MVC pattern that allows developers to write their applications in a familiar style without needing to know about the details of how long polling is implemented. Furthermore, Vault makes it easy to develop a variety of feeds that can be reused in many different applications. Internally, the Vault implementation handles long polling in a scalable way using a distributed notifier to minimize the amount of work that is required per request.

We hope that the Vault architecture will be implemented in a variety of mainstream frameworks in order to encourage the development of interesting applications based on long polling.

11 Acknowledgments

This research was supported by the National Science Foundation under Grant No. 0963859. Thanks to Tomer London, Bobby Johnson, and anonymous program committee members for reviewing various drafts of this paper.

References

[1] Bayeux protocol. `http://svn.cometd.com/trunk/bayeux/bayeux.html`.

[2] Cometd homepage. `http://cometd.org/`.

[3] Django. `http://www.djangoproject.com/`.

[4] Friendfeed homepage. `http://friendfeed.com/`.

[5] Node.js homepage. `http://nodejs.org/`.

[6] Tornado web server homepage. `http://nodejs.org/`.

[7] Web socket api. `http://dev.w3.org/html5/websockets/`.

[8] GARRETT, J. J. Ajax: a new approach to web applications, February 2005. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`.

[9] LAMPSON, B. W., AND REDELL, D. D. Experience with processes and monitors in mesa. *Commun. ACM 23* (February 1980), 105–117.

[10] OUSTERHOUT, J. Fiz: A Component Framework for Web Applications. Stanford CSD Technical Report. `http://www.stanford.edu/~ouster/cgibin/papers/fiz.pdf`, 2009.

[11] OUSTERHOUT, J., AND STRATMANN, E. Managing state for ajax-driven web components. *USENIX Conference on Web Application Development* (June 2010), 73–85.

[12] REENSKAUG, T. Models-views-controllers. Xerox PARC technical note `http://heim.ifi.uio.no/~trygver/mvc-1/1979-05-MVC.pdf`, May 1979.

[13] REENSKAUG, T. Thing-model-view-editor. Xerox PARC technical note `http://heim.ifi.uio.no/~trygver/mvc-2/1979-12-MVC.pdf`, May 1979.

[14] RUSSEL, A. Comet: Low latency data for the browser, March 2006. `http://alex.dojotoolkit.org/2006/03/comet-lowlatency-data-for-the-browser/`.

[15] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw. 11* (February 2003), 17–32.

Detecting Malicious Web Links and Identifying Their Attack Types

Hyunsang Choi
Korea University
Seoul, Korea
realchs@korea.ac.kr

Bin B. Zhu
Microsoft Research Asia
Beijing, China
binzhu@microsoft.com

Heejo Lee
Korea University
Seoul, Korea
heejo@korea.ac.kr

Abstract

Malicious URLs have been widely used to mount various cyber attacks including spamming, phishing and malware. Detection of malicious URLs and identification of threat types are critical to thwart these attacks. Knowing the type of a threat enables estimation of severity of the attack and helps adopt an effective countermeasure. Existing methods typically detect malicious URLs of a single attack type. In this paper, we propose method using machine learning to detect malicious URLs of all the popular attack types and identify the nature of attack a malicious URL attempts to launch. Our method uses a variety of discriminative features including textual properties, link structures, webpage contents, DNS information, and network traffic. Many of these features are novel and highly effective. Our experimental studies with 40,000 benign URLs and 32,000 malicious URLs obtained from real-life Internet sources show that our method delivers a superior performance: the accuracy was over 98% in detecting malicious URLs and over 93% in identifying attack types. We also report our studies on the effectiveness of each group of discriminative features, and discuss their evadability.

1 Introduction

While the World Wide Web has become a killer application on the Internet, it has also brought in an immense risk of cyber attacks. Adversaries have used the Web as a vehicle to deliver malicious attacks such as phishing, spamming, and malware infection. For example, phishing typically involves sending an email seemingly from a trustworthy source to trick people to click a URL (Uniform Resource Locator) contained in the email that links to a counterfeit webpage.

To address Web-based attacks, a great effort has been directed towards detection of malicious URLs. A common countermeasure is to use a blacklist of malicious URLs, which can be constructed from various sources, particularly human feedbacks that are highly accurate yet time-consuming. Blacklisting incurs no false positives, yet is effective only for known malicious URLs. It cannot detect unknown malicious URLs. The very nature of exact match in blacklisting renders it easy to be evaded.

This weakness of blacklisting has been addressed by anomaly-based detection methods designed to detect unknown malicious URLs. In these methods, a classification model based on discriminative rules or features is built with either knowledge a priori or through machine learning. Selection of discriminative rules or features plays a critical role for the performance of a detector. A main research effort in malicious URL detection has focused on selecting highly effective discriminative features. Existing methods were designed to detect malicious URLs of a single attack type, such as spamming, phishing, or malware.

In this paper, we propose a method using machine learning to detect malicious URLs of all the popular attack types including phishing, spamming and malware infection, and identify the attack types malicious URLs attempt to launch. We have adopted a large set of discriminative features related to textual patterns, link structures, content composition, DNS information, and network traffic. Many of these features are novel and highly effective. As described later in our experimental studies, link popularity and certain lexical and DNS features are highly discriminative in not only detecting malicious URLs but also identifying attack types. In addition, our method is robust against known evasion techniques such as redirection [42], link manipulation [16], and fast-flux hosting [17].

Identification of attack types is useful since the knowledge of the nature of a potential threat allows us to take a proper reaction as well as a pertinent and effective countermeasure against the threat. For example, we may conveniently ignore spamming but should respond immediately to malware infection. Our experiments on 40,000 benign URLs and 32,000 malicious URLs obtained from real-life Internet sources show that our method has achieved an accuracy rate of more than 98% in detecting malicious URLs and an accuracy rate

This work was done when Hyunsang Choi was an intern at Microsoft Research Asia. Contact author: Bin B. Zhu (binzhu@ieee.org).

of more than 93% in identifying attack types.

This paper has the following major contributions:

- We propose several groups of novel, highly discriminative features that enable our method to deliver a superior performance and capability on both detection and threat-type identification of malicious URLs of main attack types including spamming, phishing, and malware infection. Our method provides a much larger coverage than existing methods while maintaining a high accuracy.

- To the best of our knowledge, this is the first study on classifying multiple types of malicious URLs, known as a multi-label classification problem, in a systematic way. Multi-label classification is much harder than binary detection of malicious URLs since multi-label learning has to deal with the ambiguity that an entity may belong to several classes.

The remainder of this paper is organized as follows. We present the proposed method and the learning algorithms it uses in Section 2, and describe the discriminative features our method uses in Section 3. Evaluation of our method with real-life data is reported in Section 4. We review related work in Section 5, and conclude the paper in Section 6.

2 Our Framework

2.1 Overview

Our method consists of three stages as shown in Figure 1: training data collection, supervised learning with the training data, and malicious URL detection and attack type identification. These stages can operate sequentially as in batched learning, or in an interleaving manner: additional data is collected to incrementally train the classification models while the models are used in detection and identification. Interleaving operations enable our method to adapt and improve continuously with new data, especially with online learning where the output of our method is subsequently labeled and used to train the classification models.

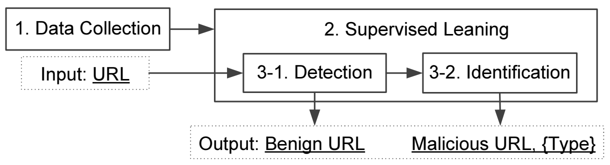


Figure 1: The framework of our method.

2.2 Learning Algorithms

The two tasks performed by our method, detecting malicious URLs and identifying attack types, need different machine learning methods. The first task is a binary classification problem. The Support Vector Machine (SVM) is used to detect malicious URLs. The second task is a multi-label classification problem. Two multi-label classification methods, (RA*k*EL [38] and ML-*k*NN [48]), are used to identify attack types.

Task1: Support Vector Machine (SVM). SVM is a widely used machine learning method introduced by Vapnik *et al.* [8]. SVM constructs hyperplanes in a high or infinite dimensional space for classification. Based on the Structural Risk Maximization theory, SVM finds the hyperplane that has the largest distance to the nearest training data points of any class, called *functional margin*. Functional margin optimization can be achieved by maximizing the following equation

$$\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

subject to

$$\sum_{i=1}^{n} \alpha_i y_i = 0, 0 \leq \alpha_i \leq C, i = 1, 2, ..., n$$

where α_i and α_j are coefficients assigned to training samples x_i and x_j. $K(x_i, x_j)$ is a kernel function used to measure similarity between the two samples. After specifying the kernel function, SVM computes the coefficients which maximize the margin of correct classification on the training set. C is a regulation parameter used for tradeoff between training error and margin, and training accuracy and model complexity.

Task2: RA*k*EL. and ML-*k*NN. RA*k*EL is a high-performance multi-label learning method that accepts any multi-label learner as a parameter. RA*k*EL creates *m* random sets of *k* label combinations, and builds an ensemble of Label Powerset (LP) [47] classifiers from each of the random sets. LP is a transformation-based algorithm that accepts a single-label classifier as a parameter. It considers each distinct combination of labels that exists in the training set as a different class value of a single-label classification task. Ranking of the labels is produced by averaging the zero-one predictions of each model per considered label. An ensemble voting process under a threshold *t* is then employed to make a decision for the final classification set. We use C4.5 [32] as the single-label classifier and LP as a parameter of the multi-label learner.

ML-*k*NN is derived from the traditional *k*-Nearest Neighbor (*k*NN) algorithm [1]. For each unseen instance, its k nearest neighbors in the training set are first identified. Based on the statistical information gained from the label sets of these neighboring instances, maximum a posteriori principle is then utilized to determine the label set for the unseen instance.

3 Discriminative Features

Our method uses the same set of discriminative features for both tasks: malicious URL detection and attack type identification. These features can be classified into six groups: lexicon, link popularity, webpage content, DNS, DNS fluxiness, and network traffic. They can effectively represent the entire *multifaceted* properties of a malicious URL and are robust to known evasion techniques.

3.1 Lexical Features

Malicious URLs, esp. those for phishing attacks, often have distinguishable patterns in their URL text. Ten lexical features, listed in Table 1, are used in our method. Among these lexical features, the average domain/path token length (delimited by '.', '/', '?', '=', '-', '_') and brand name presence were motivated from a study by McGrath and Gupta [24] that phishing URLs show different lexical patterns. For example, a phishing URL likely targets a widely trusted brand name for spoofing, thus contains the brand name. Therefore, we employ a binary feature to check whether a brand name is contained in the URL tokens but not in its SLD (Second Level Domain)[1].

Table 1: Lexical features (LEX)

No.	Feature	Type
1	Domain token count	Integer
2	Path token count	Integer
3	Average domain token length	Real
4	Average path token length	Real
5	Longest domain token length	Integer
6	Longest path token length	Integer
7∼9	Spam, phishing and malware SLD hit ratio	Real
10	Brand name presence	Binary

In our method, the detection model maintains two lists of URLs: a list of benign URLs and a list of malicious URLs. The identification model breaks the list of malicious URLs into three lists: spam, phishing, and malware URL lists. For a URL, our method extracts its SLD and calculates the ratio of the number that the SLD matches SLDs in the list of malicious URLs or a list of specific type of malicious URLs (e.g., spam URL list) to the number that the SLD matches SLDs in the list of benign URLs. This ratio is called the *malicious* or a specific attack type (e.g., *spam*) *SLD hit ratio* feature, which is actually an *a priori* probability of the URL to be malicious or of a specific malicious type (e.g., spam) based on the precompiled URL lists.

Previous methods use URL tokens as the "bag-of-words" model in which the information of a token's position in a URL is lost. By examining a large set of malicious and benign URLs, we observed that the position of a URL token also plays an important role. SLDs are relatively hard to forge or manipulate than URL tokens at other positions. Therefore, we discard the widely used "bag-of-words" approach and adopt several new features differentiating SLDs from other positions, resulting in a higher robustness against lexical manipulations by attackers. Lexical features No. 1 to No. 4 in Table 1 are from previous work. Feature No. 10 is different from the "bag-of-words" model used in previous work by excluding the SLD position. The other lexical features in Table 1 are novel features never used previously.

3.2 Link Popularity Features

One of the most important features used in our method is "link popularity", which is estimated by counting the number of incoming links from other webpages. Link popularity can be considered as a reputation measure of a URL. Malicious sites tend to have a small value of link popularity, whereas many benign sites, especially popular ones, tend to have a large value of link popularity. Both link popularity of a URL and link popularity of the URL's domain are used in our method. Link popularity (LPOP) can be obtained from a search engine[2]. Different search engines may produce different link popularity due to different coverage of webpages each has crawled. In our method, five popular search engines, `Altavista`, `AllTheWeb`, `Google`, `Yahoo!`, and `Ask`, are used to calculate the link popularity of a URL and the link popularity of its domain, corresponding to LPOP features No. 1 to 10 in Table 2.

One problem in using link popularity is "link-farming [16]", a link manipulation that uses a group of webpages to link together. To address this problem, we develop five additional LPOP features by exploiting different link properties between link-manipulated malicious websites and popular benign websites. The first feature, the distinct domain link ratio, is the ratio of the number of unique domains to the total number of domains that link to the targeted URL. The second feature, the max domain link ratio, is the ratio of the maximum number of links from a single domain to the total number of domains that link to the targeted URL. Link-manipulated malicious URLs tend to be linked many times with a few domains, resulting in a low score on the distinct domain link ratio and a high score on the max domain link ratio. A study by Castillo *et al.* [4] indicates that spam pages tend to be linked mainly by spam pages. We believe that a hypothesis to assume that not only spam pages, but also phishing and malware pages tend to be linked by phishing and malware pages, respectively, is plausible. Therefore, we develop the last three features: spam link ratio, phishing link ratio, and malware link ratio. Each represents the ratio from domains of a specific malicious type that link to the targeted URL. To measure these three features, we use the malicious URL lists described in Section 3.1. The link popularity features described in this subsection are all novel

[1]Brand names can be taken from the SLDs of the Alexa [2] top 500 site list.

[2]For example, we can use *Yahoo! site explorer* to get inlinks of target URLs.

features.

Table 2: Link popularity features (LPOP)

No.	Feature	Type
1∼5	5 LPOPs of the URL	Integer
6∼10	5 LPOPs of the domain	Integer
11	Distinct domain link ratio	Real
12	Max domain link ratio	Real
13∼15	Spam, phishing and malware link ratio	Real

3.3 Webpage Content Features

Recent development of the dynamic webpage technology has been exploited by hackers to inject malicious code into webpages through importing and thus hiding exploits in webpage content. Therefore, statistical properties of client-side code in the Web content can be used as features to detect malicious webpages. To extract webpage content features (CONTs), we count the numbers of HTML tags, iframes, zero size iframes, lines, and hyperlinks in the webpage content. We also count the number for each of the following seven suspicious native JavaScript functions: escape(), eval(), link(), unescape(), exec(), link(), and search() functions. As suggested by a study of Hou *et al.* [18], these suspicious JavaScript functions are often used by attacks such as cross-site scripting and Web-based malware distribution. For example, unescape() can be used to decode an encoded shellcode string to obfuscate exploits. The counts of these seven suspicious JavaScript functions form features No. 6 to No. 12 in Table 3. The last feature in this table is the the sum of these function counts, i.e., the total count of these suspicious JavaScript functions. All the features in Table 3 are from the previous work [18].

Table 3: Webpage content features (CONT)

No.	Feature	Type
1	HTML tag count	Integer
2	Iframe count	Integer
3	Zero size iframe count	Integer
4	Line count	Integer
5	Hyperlink count	Integer
6∼12	Count of each suspicious JavaScript function	Integer
13	Total count of suspicious JavaScript functions	Integer

The CONTs may not be effective to distinguish phishing websites from benign websites because a phishing website should have similar content as the authentic website it targets. However, this very nature of being sensitive to one malicious type but insensitive to other malicious types is very much desired in identifying the type of attack that a malicious URL attempts to launch.

3.4 DNS Features

The DNS features are related to the domain name of a URL. Malicious websites tend to be hosted by less reputable service providers. Therefore, the DNS information can be used to detect malicious websites. Ramachandran *et al.* [33] showed that a significant portion of spammers came from a relatively small collection of autonomous systems. Other types of malicious URLs are also likely to be hosted by disreputable providers. Therefore, the Autonomous System Number (ASN) of a domain can be used as a DNS feature.

Table 4: DNS features (DNS)

No.	Feature	Type
1	Resolved IP count	Integer
2	Name server count	Integer
3	Name server IP count	Integer
4	Malicious ASN ratio of resolved IPs	Real
5	Malicious ASN ratio of name server IPs	Real

All the five DNS features listed in Table 4 are novel features. The first is the number of IPs resolved for a URL's domain. The second is the number of name servers that serves the domain. The third is the number of IPs these name servers are associated with. The next two features are related to ASN. As we have mentioned in Section 3.1, our method maintains a benign URL list and a malicious URL list. For each URL in the two lists, we record its ASNs of resolved IPs and ASNs of the name servers. For a URL, our method calculates hit counts for ASNs of its resolved IPs that matches the ASNs in the malicious URL list. In a similar manner, it also calculates the ASN hit counts using the benign URL list. Summation of malicious ASN hit counts and summation of benign ASN hit counts are used to estimate the malicious ASN ratio of resolved IPs, which is used as an *a priori* probability for the URL to be hosted by a disreputable service provider based on the precompiled URL lists. ASNs can be extracted from MaxMind's database file [14].

3.5 DNS Fluxiness Features

A newly emerging fast-flux service network (FFSN) establishes a proxy network to host illegal online services with a very high availability [17]. FFSNs are increasingly employed by attackers to provide malicious content such as malware, phishing websites, and spam campaigns. To detect URLs which are served by FFSNs, we use the discriminative features proposed by Holz *et al.* [17], as listed in Table 5.

Table 5: DNS fluxiness features (DNSF)

No.	Feature	Type
1∼2	φ of N_{IP}, N_{AS}	Real
3∼5	φ of N_{NS}, N_{NSIP}, N_{NSAS}	Real

We lookup the domain name of a URL and repeat the DNS lookup after TTL (Time-To-Live value in a DNS packet) timeout given in the first answer to have consecutive lookups of the same domain. Let N_{IP} and N_{AS} be the total number of unique IPs and ASNs of each IP, respectively, and N_{NS}, N_{NSIP}, N_{NSAS} be the total number of unique name servers, name server IPs, and ASNs of the name server IPs in all DNS lookups. Then, we can estimate *fluxiness* using the acquired numbers. For example, *fluxiness* of the resolved IP address is estimated as follows,

$$\varphi = N_{IP}/N_{single},$$

where φ is the *fluxiness* of the domain and N_{single} is the number of IPs that a single lookup returns. Similarly, all of the other *fluxiness* features are estimated.

3.6 Network Features

Attackers may try to hide their websites using multiple redirections such as iframe redirection and URL shortening. Even though also used by benign websites, the distribution of redirection counts of malicious websites is different from that of redirection counts of benign websites [31]. Therefore, redirection count can be a useful feature to detect malicious URLs. In a HTTP packet, there is a content-length field which is the total length of the entire HTTP packet. Hackers often set malformed (negative) content-length in their websites in a buffer overflow exploit. Therefore, content-length is used as a network discriminative feature. Benign sites tend to be more popular with a better service quality than malicious ones. Web technologies tend to make popular websites quick to look up and faster to download. In particular, benign domains tend to have a higher probability to be cached in a local DNS server than malicious domains, esp. those employing FFSNs and dynamic DNS. Therefore, domain lookup time and average download speed are also used as features to detect malicious URLs. The network features listed in Table 6 except the third and fifth features are novel features.

Table 6: Network features (NET)

No.	Feature	Type
1	Redirection count	Integer
2	Downloaded bytes from content-length	Real
3	Actual downloaded bytes	Real
4	Domain lookup time	Real
5	Average download speed	Real

4 Evaluation

In this section, we evaluate the performance of our method for both malicious URL detection and attack type identification. We also study the effectiveness of different groups of features. The main findings of our experiments include:

- **Link popularity.** Link popularity first used in our method is highly discriminative for both malicious URL detection (over 96% accuracy) and attack type identification (over 84% accuracy). `Google`'s search engine was not suitable to estimate link popularity since it reported just a partial list of link popularity.
- **Link distribution.** Malicious URLs are mainly linked by malicious URLs of the same attack type: about 56% of malicious URLs were found to be linked only by the malicious URLs of the same attack type.
- **Multi-labels.** In our collected malicious URLs, over 45% belong to multiple types of threat. Therefore, malicious URLs should be classified with a multi-label classification method in order to produce a more accurate result on the nature of attack.
- **Identification.** Our method has achieved an accuracy rate of over 93% in attack type identification. It is worth mentioning that novel features used in our method including malicious SLD hit ratio in LEX, three malicious link ratios in LPOP, two malicious ASN ratios in DNS were found to be highly effective in distinguishing different attack types.

4.1 Methodology and Data Sets

Real-life data was collected from various sources to evaluate our method:

- **Benign URLs.** 40,000 benign URLs were collected from the following two sources as used in previous work [49, 43, 21, 22]: 1) randomly selected 20,000 URLs from the DMOZ Open Directory Project [10] (manually submitted by users), 2) randomly selected 20,000 URLs from `Yahoo!`'s directory (generated by visiting `http://random.yahoo.com/bin/ryl`)[3].
- **Spam URLs.** The spam URLs were acquired from jwSpamSpy [19] which is known as an e-mail spam filter for Microsoft Windows. We also used a publicly available Web spam dataset [3].
- **Phishing URLs.** The phishing URLs were acquired from `PhishTank` [29], a free community site where anyone can submit, verify, track and share phishing data.
- **Malware URLs.** The malware URLs were obtained from `DNS-BH` [11], a project creates and maintains a list of URLs that are known to be used to propagate malware.

The data set of malicious URLs is simply the union of the three individual data sets of malicious types. A total of 32,000 malicious URLs was collected. A malicious URL may launch multiple types of attack, *i.e.*, belongs to multiple malicious types. The malicious data sets collected above were marked with only single labels. URLs

[3]Many URLs from 1) and 2) did not have any sub-path. We adjusted the ratio of benign URLs with a sub-path to be half of benign URLs.

of multi-labels were found by querying both McAfee SiteAdvisor[4] [23] and WOT[5] (Web of Trust) [41] for each URL in the malicious URL data set. The two sites provide reputation of a submitted website URL including the detailed malicious types it belongs to. Their information was relatively accurate, although they had made errors (*e.g.*, SiteAdvisor has incorrectly labeled websites[6] and WOT was manipulated by attackers to generate incorrect labels[7]). We use (λ_i) with a single index i to represent a single type: spam (λ_1), phishing (λ_2), malware (λ_3). Multi-labels are represented by the set of their associated indexes, *e.g.*, $\lambda_{1,3}$ represents a URL of both spam and malware. Table 7 shows the resulting distribution of multi-label URLs, where L_{SAd} and L_{WOT} represent the results reported by SiteAdvisor and WOT, respectively, and L_{Both} denotes their intersection. From Table 7, about half of the malicious URLs were classified to be multi-labels: 45% by SiteAdvisor and 46% by WOT. Comparing the labeling results by both L_{SAd} and L_{WOT}, 91% of the URLs were labeled consistently whereas 9% of URLs were labeled inconsistently by the two sites.

Table 7: The collected data set of multi-labels

Label	Attribute	L_{SAd}	L_{WOT}	L_{Both}
λ_1	spam	6020	6432	5835
λ_2	phishing	1119	1067	899
λ_3	malware	9478	8664	8105
$\lambda_{1,2}$	spam, phishing	4076	4261	3860
$\lambda_{1,3}$	spam, malware	2391	2541	2183
$\lambda_{2,3}$	phishing, malware	4729	4801	4225
$\lambda_{1,2,3}$	spam, phishing, malware	2219	2170	2080

Once the URL data sets were built, three crawlers were used to crawl features from different sources. A webpage crawler crawled the webpage content features and the network features by accessing each URL in the data sets. We implemented a module to the webpage crawler using the cURL library [9] to detect redirections (including URL shortening) and find original URLs automatically. A link popularity crawler crawled the link popularity features from the five search engines, Altavista, AllTheWeb, Google, Yahoo!, and Ask, for each URL and collected inlink information. A DNS crawler crawled and calculated the DNS features and DNS fluxiness features by sending queries to DNS servers.

Two-fold cross validation was performed to evaluate our method: the URLs in each data set were randomly split into two groups of equal size: one group was selected as the training set while the other was used as the testing set. Ten rounds of two-fold cross validation were used to obtain the performance for both malicious URL detection and attack type identification. The SVM-light [35] software package was used as the support vector machine implementation in our evaluation.

[4]The SiteAdvisor is a service to report safety of websites using a couple of webpage analysis algorithms.

[5]The WOT is a community-based safe surfing tool that calculates the reputation of a website through a combination of user ratings and data from trusted sources.

[6]http://en.wikipedia.org/wiki/McAfee_SiteAdvisor

[7]http://mashable.com/2007/12/04/web-of-trust/

4.2 Malicious URL Detection Results

The following metrics were used to evaluate the detection performance: accuracy (ACC) which is the proportion of true results (both true positives and true negatives) over all data sets; true positive rate (TP, also referred to as recall) which is the number of the true positive classifications divided by the number of positive examples; false positive rate (FP) and false negative rate (FN) which are defined similarly.

4.2.1 Detection Accuracy

By applying all the discriminative features on the data sets described in Section 4.1, our malicious URL detector produced the following results: 98.2% for the accuracy, 98.9% for the true positive rate, 1.1% for the false positive rate, and 0.8% for the false negative rate. We also conducted the same experiments using only the 20,000 benign URLs collected from Yahoo!'s directory. The results were similar: 97.9% for the accuracy, 98.2% for the true positive rate, 0.98% for the false positive rate, and 1.08% for the false negative rate.

To study the effectiveness of each feature group, we performed detection using only each individual feature group. The resulting accuracy and true positive rate are shown in Figure 2. We can clearly see in this figure that LPOP is the most effective group of features in detecting malicious URLs in terms of both detection accuracy and true positive rate.

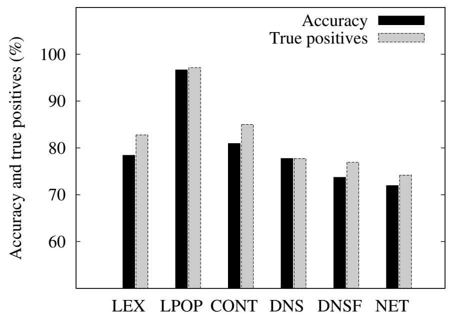


Figure 2: Detection accuracy and true positives for each group of features.

We also compared the performance of each feature group on detecting each type of malicious URLs by mixing the corresponding malicious URL data set with the benign URL data set. The resulting accuracies and true positive rates are shown in Table 8.

As expected, the lexical features (LEX) are effective on detecting phishing URLs, but did a poor job to detect spam and malware URLs. This is because the latter types do not show very different textual patterns as

Table 8: Detection accuracy and true positive rate (%) of individual feature groups for each malicious type

Dataset	Metric	Feature group					
		LEX	LPOP	CONT	DNS	DNSF	NET
Spam	ACC	73.0	97.2	82.8	77.4	**87.7**	72.1
	TP	72.4	97.4	74.2	75.9	**86.3**	77.4
Phishing	ACC	**91.6**	98.1	77.3	76.3	71.8	77.2
	TP	**86.1**	95.1	82.8	76.9	70.1	78.2
Malware	ACC	70.3	96.2	**86.2**	78.6	68.1	73.3
	TP	74.5	93.2	**88.4**	75.1	74.2	78.2

compared with benign URLs. A different sensitivity to a different malicious type is exactly what we want to distinguish one malicious type from other malicious types (phishing from spam and malware for the specific case of lexical features) in the attack type identification to be reported in Section 4.3. These partially discriminative features (effective only for some types of attack) and the features that are effective for all the malicious types form the set of discriminative features for our malicious URL detector.

The link popularity features (LPOP) outperformed all the other groups of features for detecting any type of malicious URLs. Table 8 shows that the webpage content features (CONT) are useful in distinguishing malware URLs from benign ones. This is because malware URLs usually have malicious tags or scripts in their Web content to infect visitors. From Table 8, it seems that the webpage content features are also effective in detecting spam and phishing URLs as malicious URLs from a mixture of malicious and benign URLs. That might be partially due to the fact that many spam or phishing URLs also belonged to malware, as we have seen in Section 4.1. Note that a URL is claimed to be malicious no matter which malicious type it is detected to belong to.

From Table 8, the DNS fluxiness features (DNSF) were effective to detect spam URLs. This should be due to the fact that FFSNs were widely used by spam campaigns, as shown by Moore *et al.* [25]. Malicious network behaviors such as redirections using multiple proxies can be employed by any type of threat. That can explain similar performance of the network features (NET) for detecting each type of malicious URLs.

4.2.2 Link Popularity Feature Analysis

In this section, we study the effectiveness of the link popularity features in detail, and show the effectiveness of our method for two unfavorable scenarios when the link popularity features are not effective:1) the case when malicious websites have high manipulated popularity scores; and 2) the case when newly-setup benign websites do not have high popularity scores.

First, we studied the distribution of the link popularity for each data set. In our data sets, malicious URLs had typically much smaller LPOP than benign URLs. A majority, more precisely 60.35%, of the malicious URLs had 0 link popularity. On the other hand, only a very small portion of benign URLs had almost 0 link popularity. This confirms the observation in Section 4.2.1 that LPOP is effective to differentiate malicious URLs from benign URLs.

Next we studied the quality of the link popularities retrieved from the five different search engines: `Altavista`, `AllTheWeb`, `Google`, `Yahoo!`, and `Ask`. The distribution of LPOP for each search engine over 20,000 benign URLs randomly selected from the collected 40,000 benign URLs is shown in Figure 3, and the distribution over the 32,000 malicious URLs is shown in Figure 4. The x-axis in both figures is the index of the URLs sorted by the link popularity.

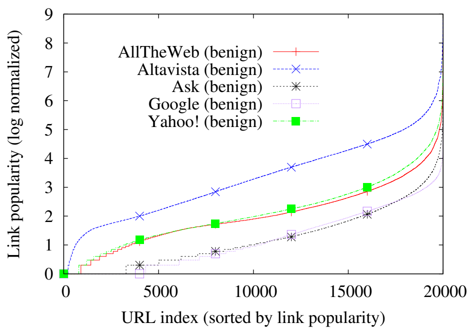


Figure 3: LPOP of benign URLs for each search engine.

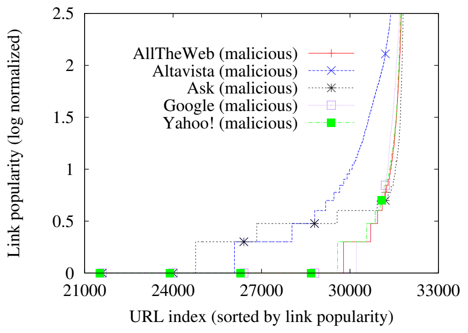


Figure 4: LPOP of malicious URLs for each search engine.

The larger the gap between benign URLs and malicious URLs a search engine reports, the more accurate that the link popularity is in distinguishing malicious URLs from benign URLs. `Google` tends to report a lower link popularity for both benign and malicious URLs and thus should produce higher false positives and lower false negatives. Table 9 shows the measured metrics for the malicious URL detection using only LPOP reported by each individual search engine. From the table, `Google` yielded high false positives (12.3%) and low false negatives (2.1%). `AllTheWeb` showed a link popularity distribution similar to that of `Yahoo!`. They had similar performance on malicious URL detection. This is not a surprise since `AlltheWeb` started to use Yahoo!'s database since March 2004[8].

The result using Google was a surprise to us. We expected that Google would report the same, if not higher, link polarity than other search engines since it should have more comprehensive information of the Web. It turned out that Google just reported a partial list of link popularity, as their official website described[9]. The Google Webmaster Tool provides more comprehensive external link information, but we could not use it since it is available only for the owner's website.

Table 9: Detection accuracy, false positives and false negatives using only LPOP reported by each individual search engine (%)

Metric	AllTheWeb	Altavista	Ask	Google	Yahoo!
ACC	95.1	95.6	84.0	85.7	95.9
TP	95.3	96.3	85.7	86.7	95.7
FP	2.7	2.7	8.4	**12.3**	2.1
FN	2.2	1.6	7.6	**2.1**	2.1

Unpopular legitimate link classification. From the results reported above, we can conclude that LPOP is the most effective discriminative feature for detecting malicious URLs. It outperforms all the other feature groups by a large margin. However, LPOP alone may be ineffective for certain types of URLs, for example, to distinguish malicious URLs from a group of unpopular or newly setup benign URLs which also have low LPOP scores. This is the worst scenario for our malicious URL detector since the most effective feature, LPOP, is ineffective in this case. To conduct a test on the performance for this worst scenario, we used only the benign and malicious URLs which had zero LPOP to evaluate the performance of our detector. We obtained the following results on malicious URL detection: 91.2% for the accuracy, 4.0% for false positives, and 4.8% for false negatives. The accuracy remains high even under this worst scenario.

Popularity-manipulated link classification. As described in Section 3.2, some malicious URLs have high LPOP scores because their links are manipulated using a link farm [16]. We have developed five features, i.e., distinct domain link ratio, max domain link ratio, spam link ratio, phishing link ratio, and malware link ratio, to detect link manipulated malicious URLs. To make our detector light-weight and feasible in real-time applications, we used sampled link information instead of the whole link information to calculate each of these features. To evaluate the performance when the links are manipulated, we collected malicious URLs which had high LPOP scores (LPOP > 10). Among the 32,000 malicious URLs we collected, only 622 URLs could be selected. Their distinct domain link ratio and max domain link ratio are shown against those of benign URLs in Figure 5. This figure indicates that the popularity-manipulated malicious URLs show a different pattern from those of benign URLs. Moreover, about 90% of these malicious URLs have more than 10% malicious link ratio (spam link ratio, phishing link ratio, and malware link ratio), whereas about 5% of benign URLs have more than 10% malicious link ratio. About 56% of these malicious URLs were linked exclusively by malicious URLs of the same type. Consequently, we obtained 90.03% accuracy in detecting link-manipulated malicious URLs with the aforementioned five features.

[8]AlltheWeb was taken over by Yahoo!.

[9]http://sites.google.com/site/webmasterhelpforum/en/faq--crawling--indexing---ranking\#links

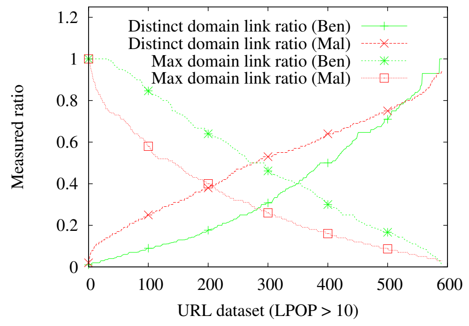


Figure 5: Distinct domain link ratio and max domain link ratio for benign and malicious URLs.

4.2.3 Error Analysis

In this section, both false positives and negatives are further studied to understand why these errors happened in order to further improve our method.

False positives. A false positive is when a benign URL is misclassified as malicious. False positives can be broadly categorized as follows:

- **Disreputable URL.** A benign URL is likely misclassified by our detector if it fits into two or more of the following three cases: 1) the URL's domain has a very low link popularity (LPOP errors), 2) the URL contains a malicious SLD (LEX errors), and 3) the URL's domain is hosted by malicious ASNs (DNS errors). In this case, a benign URL can be considered as a disreputable URL. More than 90% of the false positives belonged to the disreputable case (*e.g.*, 208.43.27.50/~mike).

- **Contentless URL.** Some benign URLs had no content on their webpages. In this case, CONT would fail (*e.g.*, 222.191.251.167, 1traf.com, and 3gmatrix.cn).

- **Brand name URL.** Some benign URLs contained a brand name keyword even they were not related to the brand domain. These URLs could be misclassified as malicious (*e.g.*, twitterfollower.wikispaces.com).

- **Abnormal token URL.** We observed several benign URLs which had unusual long domain tokens typically appearing in phishing URLs (*e.g.*, `centraldevideoscomhomensmaduros.blogspot.com`).

False negatives. A false negative is when a malicious URL is undetected. Most false negatives were hosted by popular social networking sites which had a high link popularity and most URLs they hosted were benign. Most of the false negative URLs were of spam or phishing type. They generated features similar to those of benign URLs. More than 95% of the false negatives belonged to this case (*e.g.*, `blog.libero.it/matteof97/` and `digilander.libero.it/Malvin92/?`). This will be further discussed in Section 4.4.

4.3 Attack Type Identification Results

To evaluate the performance of attack type identification, the following metrics given in [37] for multi-label classification were used: 1) micro and macro averaged metrics, and 2) ranking-based metrics with respect to the ground truth of multi-label data.

Identification metrics. Additional notation is first introduced. Assume that there is an evaluation data set of multi-label examples $(x_i, Y_i), i = 1, ...m$, where x_i is a feature vector, $Y_i \subseteq L$ is the set of true labels, and $L = \{\lambda_j : j = 1...q\}$ is the set of all labels.

- **Micro-averaged and macro-averaged metrics.** To evaluate the average performance across multiple categories, we apply two conventional methods: micro-average and macro-average [45]. The micro-average gives an equal weight to every data set, while the macro-average gives an equal weight to every category, regardless of its frequency. Let tp_λ, tn_λ, fp_λ, and fn_λ denote the number of true positives, true negatives, false positives, and false negatives, respectively, after evaluating binary classification metrics B (accuracy, true positives, etc.) for a label λ. The micro-averaged and macro-averaged version of B can be calculated as follows:

$$B_{micro} = B(\sum_{\lambda=1}^{M} tp_\lambda, \sum_{\lambda=1}^{M} tn_\lambda, \sum_{\lambda=1}^{M} fp_\lambda, \sum_{\lambda=1}^{M} fn_\lambda),$$

$$B_{macro} = \frac{1}{M} \sum_{\lambda=1}^{M} B(tp_\lambda, tn_\lambda, fp_\lambda, fn_\lambda).$$

- **Ranking-based metrics.** Among several ranking-based metrics, we employ the ranking loss and average precision for the evaluation. Let $r_i(\lambda)$ denote the rank predicted by a label ranking method for a label λ. The most relevant label receives the highest rank, while the least relevant label receives the lowest rank. The *ranking loss* is the number of times that irrelevant labels are ranked higher than relevant labels. The ranking loss, denoted as R_{Loss}, is calculated as follows:

$$R_{loss} = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{|Y_i||\overline{Y}_i|} |\{(\lambda_a, \lambda_b) : r_i(\lambda_a) > r_i(\lambda_b),$$

$$(\lambda_a, \lambda_b) \in Y_i \times \overline{Y}_i\}|$$

where $\overline{Y}_i$ is the complementary set of Y_i with respect to L. The *average precision*, denoted by P_{avg}, is the average fraction of labels ranked above a particular label $\lambda \in Y_i$ which are actually in Y_i. It is calculated as follows:

$$P_{avg} = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{|Y_i|} | \sum_{\lambda \in Y_i} \frac{|\{\lambda' \in Y_i : r_i(\lambda') \leq r_i(\lambda)\}|}{r_i(\lambda)}.$$

Table 10: Multi-label classification results (%)

	Label	Averaged			Ranking-based	
		ACC	micro TP	macro TP	R_{loss}	P_{avg}
RA*k*EL	L_{SAd}	90.70	87.55	88.51	3.45	96.87
	L_{WOT}	90.38	88.45	89.59	4.68	93.52
	L_{Both}	**92.79**	**91.23**	**89.04**	**2.88**	**97.66**
ML-*k*NN	L_{SAd}	91.34	86.45	87.93	3.42	95.85
	L_{WOT}	91.04	88.96	89.77	3.77	96.12
	L_{Both}	**93.11**	**91.02**	**89.33**	**2.61**	**97.85**

Identification accuracy. We performed the multi-label classification by using three label sets, L_{SAd}, L_{WOT} and L_{Both} mentioned in Section 4.1. The results for two different learning algorithms, RA*k*EL algorithm and ML-*k*MN, are shown in Table 10, where micro TP and macro TP are micro-averaged true positives and macro-averaged true positives, respectively. The following results were obtained: the average accuracy was 92.95%, whereas the average precision of ranking of the two algorithms was 97.76%. The accuracy on the label set L_{Both} was always higher than that on either L_{SAd} or L_{WOT}. This implies that more accurate label set produces a more accurate result for identifying attack types.

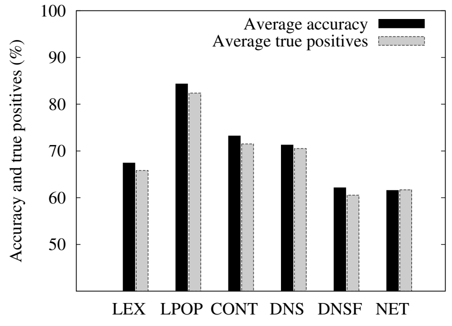


Figure 6: Average accuracy and micro-averaged true positives (%).

Fig. 6 shows effectiveness of each feature group in identifying attack types. Among the top ten most effective features, eight are novel features. They are three SLD hit ratio features in LEX, three malicious link ratios in LPOP, and two malicious ASN ratios in DNS. From this figure, even the link popularity features were also rather effective in distinguishing different attack types. In addition, no single feature group was highly effective in identifying attack types: they all yielded an accuracy lower than 85%. The combination of all the groups of features, however, yielded a much improved performance.

4.4 Evadability Analysis

Existing methods can be evaded by capable attackers. Similarly, our features are also evadable to a certain degree. However, it is an improvement if we can raise the bar of evasion difficulty by either increasing the evasion cost or decreasing the effectiveness of threat. To study evadability of our method, we discuss in this subsection the robustness of our method against known evasions and also possible evasion tactics.

Robust against known evasions. 1) Redirection: One possible evasion tactic is to hide the original URL using multiple redirections (also known as a "drive-by website attack" such as Iframe redirection) or a URL shortening service which makes a webpage available under a very short URL in addition to the original URL. Our method is robust against this kind of URL hiding and embedding evasions because our webpage crawler can automatically detect redirections and find the original URLs. 2) Link manipulation: As mentioned in Section 4.2.2, our method is robust against the link manipulation attack (more than 90% of link-manipulated URLs were detected). 3) Fast-flux hosting: The DNSF features used in our method can detect fast-fluxed domains.

URL obfuscation. If an attacker (or a domain generation algorithm in malware, *e.g.*, Conficker Worm) generates a domain name and path tokens with random length and counts, most statistical features in LEX will be evaded. Therefore, it is easy to evade the statistical features in LEX except our unique feature "malicious SLD hit ratio" since a plenty of domains have to be registered to evade the malicious SLD hit ratio. Evading brand name presence feature is easy but such an evasion will make a malicious URL less likely to be clicked, resulting in a reduced effectiveness of attack. URL obfuscation using IDN (Internationalized Domain Names) spoofing can also be used to evade our detector. For example, `http://www.p&#1072;ypal.com` represents `http://www.paypal.com`. Such an evasion can be easily prevented by adding a module to deobfuscate a URL to find the resulting URL in our webpage crawler.

JavaScript obfuscation. Malicious javascript often utilizes obfuscation to hide known exploits, embed redirection URLs, and evade signature-based detection methods. Particularly, JavaScript obfuscation can make the webpage crawler mislead webpage content features (CONT). To extract webpage content features accurately, the webpage crawler should have an automated deobfuscation functionality. The Firefox JavaScript deobfuscator add-on[10] inspired by "The Ultimate Deobfuscator" [5] can be used in our webpage content crawler as a JavaScript deobfuscation module.

Social network site. Utilizing social network sites (*e.g.*, `Twitter`) to attack can reduce the effectiveness of LEX, LPOP, DNS, and NET features. A possible solution against this evasion tactic is to adopt features which can differentiate hacker's fake accounts from normal users. For example, we can use the number of incoming linked accounts (*e.g.*, "followers" in `Twitter`) as a feature to detect faked accounts. Such a feature is still evadable with more sophisticated attacks which build a fake social network to link each other. Like the five link ratio features in LPOP to deal with the link popularity manipulation, similar linked account ratio features can be used to deal with a fake social network. Other countermeasures against social spam and phishing [20] can also be combined with our detector.

As mentioned in this section, it may cost little to evade a single feature group. However, evading all the features in our method would cost much more and also reduce the effectiveness of attack.

5 Related Work

This section reviews the related work of our method. They can be classified into two categories depending on how the classifier is built: machine learning methods which use machine learning to build classifiers, and other methods which build classifiers with a priori knowledge.

5.1 Non-machine learning approaches

Blacklisting. One of the most popular approaches is to build a blacklist to block malicious URLs. Several websites provide blacklists such as jwSpamSpy [19], PhishTank [29], and DNS-BH [11]. Several commercial products construct blacklist using user feedbacks and their proprietary mechanisms to detect malicious URLs, such as McAfee's SiteAdvisor [23], WOT Web of Trust [41], Trend Micro Web Reputation Query Online System [36], and Cisco IronPort Web Reputation [7]. URL blacklisting is ineffective for new malicious URLs. The very nature of exact match in URL blacklisting renders it easy to be evaded. Moreover, it takes time to analyze malicious URLs and propagate a blacklist to end users. Zhang *et al.* [46] proposed a more effective blacklisting approach, "predictive blacklists", which uses a relevance ranking algorithm to estimate the likelihood that an IP address is malicious.

VM execution. Wang *et al.* [39] detected drive-by exploits on the Web by monitoring anomalous state changes in a Virtual Machine (VM). SpyProxy [26] also uses a VM-based Web proxy defense to block suspicious

[10]The Firefox add-on shows JavaScript runs on a webpage, even if the JavaScript is obfuscated and generated on the fly [28].

Web content by executing the content in a virtual machine first. The VM-based approaches detect malicious webpages with a high accuracy, but only malware exploiting pages can be detected.

Rule-based anti-phishing. Several rule-based anti-phishing approaches have been proposed. Zhang *et al.* [49] proposed a system to detect phishing URLs with a weighted sum of 8 features related to content, lexical and WHOIS data. They used the Google Web search as a filter for phishing pages. Garera *et al.* [13] used logistic regression over manually selected features to classify phishing URLs. The features include heuristics from a URL such as Google's page rank features. Xiang and Hong [43] proposed a hybrid phishing detection method by discovering inconsistency between a phishing identity and the corresponding legitimate identity. PhishNet [30] provides a prediction method for phishing attacks using known heuristics to identify phishing sites.

5.2 Machine learning-based approaches

Detection of single attack type. Machine learning has been used in several approaches to classify malicious URLs. Ntoulas *et al.* [27] proposed to detect spam webpages through content analysis. They used site-dependent heuristics, such as words used in a page or title and fraction of visible content. Xie *et al.* [44] developed a spam signature generation framework called AutoRE to detect botnet-based spam emails. AutoRE uses URLs in emails as input and outputs regular expression signatures that can detect botnet spam. Fette *et al.* [12] used statistical methods to classify phishing emails. They used a large publicly available corpus of legitimate and phishing emails. Their classifiers examine ten different features such as the number of URLs in an e-mail, the number of domains and the number of dots in these URLs. Provos *et al.* [31] analyzed the maliciousness of a large collection of webpages using a machine learning algorithm as a pre-filter for VM-based analysis. They adopted content-based features including presence of obfuscated javascript and exploit sites pointing iframes. Hou *et al.* [18] proposed a detector of malicious Web content using machine learning. In particular, we borrow several webpage contents features from their features. Whittaker *et al.* [40] proposed a phishing website classifier to update Google's phishing blacklist automatically. They used several features obtained from domain information and page contents.

Detection of multiple attack types. The classification model of Ma *et al.* [21, 22] can detect spam and phishing URLs. They described a method of URL classification using statistical methods on lexical and host-based properties of malicious URLs. Their method detects both spam and phishing but cannot distinguish these two types of attack.

Existing machine learning-based approaches usually focus on a single type of malicious behavior. They all use machine learning to tune their classification models. Our method is also based on machine learning, but a new and more powerful and capable classification model is used. In addition, our method can identify attack types of malicious URLs. These innovations contribute to the superior performance and capability of our method.

Other related work. Web spam or spamdexing aims at gaining an undeservedly high rank from a search engine by influencing the outcome of the search engine's ranking algorithms. Link-based ranking algorithms, which our link popularity is similar to, are widely used by search engines. Link farms are typically used in Web spam to affect link-based ranking algorithms of search engines, which can also affect our link popularity. Researches have proposed methods to detect Web spams by using propagating trust or distrust through links [15], detecting bursts of linking activity as a suspicious signal [34], integrating link and content features [4], or various link-based features including modified PageRank scores [6]. Many of their techniques can be borrowed to thwart evading link popularity features in our detector through link farms.

6 Conclusion

The Web has become an efficient channel to deliver various attacks such as spamming, phishing, and malware. To thwart these attacks, we have presented a machine learning method to both detect malicious URLs and identify attack types. We have presented various types of discriminative features acquired from lexical, webpage, DNS, DNS fluxiness, network, and link popularity properties of the associated URLs. Many of these discriminative features such as link popularity, malicious SLD hit ratio, malicious link ratios, and malicious ASN ratios are novel and highly effective, as our experiments found out. SVM was used to detect malicious URLs, and both RA*k*EL and ML-*k*NN were used to identify attack types. Our experimental results on real-life data showed that our method is highly effective for both detection and identification tasks. Our method achieved an accuracy of over 98% in detecting malicious URLs and an accuracy of over 93% in identifying attack types. In addition, we studied the effectiveness of each group of discriminative features on both detection and identification, and discussed evadability of the features.

References

[1] AHA, D. W. Lazy learning: Special issue editorial. *Artifiial Intelligence Review* (1997), 7–10.

[2] ALEXA. The web information company. `http://www.alexa.com`, 1996.

[3] CASTILLO, C., DONATO, D., BECCHETTI, L., BOLDI, P., LEONARDI, S., SANTINI, M., AND VIGNA, S. A reference collection for web spam. *SIGIR Forum 40*, 2 (2006), 11–24.

[4] CASTILLO, C., DONATO, D., GIONIS, A., MURDOCK, V., AND SILVESTRI, F. Know your neighbors: web spam detection using the web topology. In *ACM SIGIR: Proceedings of the conference on Research and development in Information Retrieval* (2007).

[5] CHENETTE, S. The ultimate deobfuscator. http://securitylabs.websense.com/content/Blogs/3198.aspx, 2008.

[6] CHUNG, Y.-J., TOYODA, M., AND KITSUREGAWA, M. Identifying spam link generators for monitoring emerging web spam. In *WICOW: Proceedings of the 4th workshop on Information credibility* (2010).

[7] CISCO IRONPORT. IronPort Web Reputation: Protect and defend against URL-based threat. http://www.ironport.com.

[8] CORTES, C., AND VAPNIK, V. Support vector networks. *Machine Learning* (1995), 273–297.

[9] CURL LIBRARY. Free and easy-to-use client-side url transfer library. http://curl.haxx.se/, 1997.

[10] DMOZ. Netscape open directory project. http://www.dmoz.org.

[11] DNS-BH. Malware prevention through domain blocking. http://www.malwaredomains.com.

[12] FETTE, I., SADEH, N., AND TOMASIC, A. Learning to detect phishing emails. In *WWW: Proceedings of the international conference on World Wide Web* (2007).

[13] GARERA, S., PROVOS, N., CHEW, M., AND RUBIN, A. D. A framework for detection and measurement of phishing attacks. In *WORM: Proceedings of the Workshop on Rapid Malcode* (2007).

[14] GEOIP API, MAXMIND. Open source APIs and database for geological information. http://www.maxmind.com.

[15] GYÖNGYI, Z., AND GARCIA-MOLINA, H. Link spam alliances. In *VLDB: Proceedings of the international conference on Very Large Data Bases* (2005).

[16] GYONGYI, Z., AND GARCIA-MOLINA, H. Web spam taxonomy, 2005.

[17] HOLZ, T., GORECKI, C., RIECK, K., AND FREILING, F. C. Detection and mitigation of fast-flux service networks. In *NDSS: Proceedings of the Network and Distributed System Security Symposium* (2008).

[18] HOU, Y.-T., CHANG, Y., CHEN, T., LAIH, C.-S., AND CHEN, C.-M. Malicious web content detection by machine learning. *Expert Systems with Applications* (2010), 55–60.

[19] JWSPAMSPY. E-mail spam filter for Microsoft Windows. http://www.jwspamspy.net.

[20] LEE, K., CAVERLEE, J., AND WEBB, S. Uncovering social spammers: social honeypots + machine learning. In *ACM SIGIR: Proceeding of the international conference on Research and development in Information Retrieval* (2010).

[21] MA, J., SAUL, L. K., SAVAGE, S., AND VOELKER, G. M. Beyond blacklists: learning to detect malicious web sites from suspicious URLs. In *KDD: Proceedings of the international conference on Knowledge Discovery and Data mining* (2009).

[22] MA, J., SAUL, L. K., SAVAGE, S., AND VOELKER, G. M. Identifying suspicious URLs: an application of large-scale online learning. In *ICML: Proceedings of the International Conference on Machine Learning* (2009).

[23] MCAFEE SITEADVISOR. Service for reporting the safety of web sites. http://www.siteadvisor.com/.

[24] MCGRATH, D. K., AND GUPTA, M. Behind phishing: An examination of phisher modi operandi. In *LEET: Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats* (2008).

[25] MOORE, T., CLAYTON, R., AND STERN, H. Temporal correlations between spam and phishing websites. In *LEET: Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats* (2009).

[26] MOSHCHUK, A., BRAGIN, T., DEVILLE, D., GRIBBLE, S. D., AND LEVY, H. M. Spyproxy: Execution-based detection of malicious web content. In *Security: Proceedings of the USENIX Security Symposium* (2007).

[27] NTOULAS, A., NAJORK, M., MANASSE, M., AND FETTERLY, D. Detecting spam web pages through content analysis. In *WWW: Proceedings of international conference on World Wide Web* (2006).

[28] PALANT, W. JavaScript Deobfuscator 1.5.6. https://addons.mozilla.org/en-US/firefox/addon/javascript-deobfuscator/, 2011.

[29] PHISHTANK. Free community site for anti-phishing service. http://www.phishtank.com/.

[30] PRAKASH, P., KUMAR, M., KOMPELLA, R. R., AND GUPTA, M. PhishNet: Predictive Blacklisting to Detect Phishing Attacks. In *INFOCOM: Proceedings of the IEEE Conference on Computer Communications* (2010).

[31] PROVOS, N., MAVROMMATIS, P., RAJAB, M. A., AND MONROSE., F. All your iFRAMEs point to us. In *Security: Proceedings of the USENIX Security Symposium* (2008).

[32] QUINLAN, J. R. C4.5: Programs for machine learning. *Morgan Kaufmann Publishers* (1993).

[33] RAMACHANDRAN, A., AND FEAMSTER, N. Understanding the network-level behavior of spammers. In *SIGCOMM* (2006).

[34] SHEN, G., GAO, B., LIU, T.-Y., FENG, G., SONG, S., AND LI, H. Detecting link spam using temporal information. *IEEE International Conference on Data Mining 0* (2006), 1049–1053.

[35] T. JOACHIMS. Making large-Scale SVM Learning Practical. Advances in Kernel Methods - Support Vector Learning, B. Scholkopf and C. Burges and A. Smola (ed.). *MIT-Press* (1999).

[36] TREND MICRO. Web reputation query - online system. http://reclassify.wrs.trendmicro.com/.

[37] TSOUMAKAS, G., KATAKIS, I., AND VLAHAVAS, I. *Mining Multi-label Data*. Data Mining and Knowledge Discovery Handbook, O. Maimon, L. Rokach (Ed.), Springer, 2nd edition, 2010.

[38] TSOUMAKAS, G., KATAKIS, I., AND VLAHAVAS, I. Random k-labelsets for multi-label classification. *IEEE Transactions on Knowledge and Data Engineering* (2010).

[39] WANG, Y.-M., BECK, D., JIANG, X., ROUSSEV, R., VERBOWSKI, C., CHEN, S., AND KING, S. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *NDSS: Proceedings of the Symposium on Network and Distributed System Security* (2006).

[40] WHITTAKER, C., RYNER, B., AND NAZIF, M. Large-scale automatic classification of phishing pages. In *NDSS: Proceedings of the Symposium on Network and Distributed System Security* (2010).

[41] WOT. Web of Trust community-based safe surfing tool. http://www.mywot.com/.

[42] WU, B., AND DAVISON, B. D. Cloaking and redirection: A preliminary study. In *AIRWeb: Proceedings of the 1st Workshop on Adversarial Information Retrieval on the Web* (2005).

[43] XIANG, G., AND HONG, J. I. A hybrid phish detection approach by identity discovery and keywords retrieval. In *WWW: Proceedings of the international conference on World Wide Web* (2009).

[44] XIE, Y., YU, F., ACHAN, K., PANIGRAHY, R., HULTEN, G., AND OSIPKOV, I. Spamming botnets: signatures and characteristics. In *SIGCOMM* (2008).

[45] YANG, Y. An evaluation of statistical approaches to text categorization. *Journal of Information Retrieval* (1999), 67–88.

[46] ZHANG, J., PORRAS, P., AND ULLRICH, J. Highly predictive blacklisting. In *Security: Proceedings of the USENIX Security Symposium* (2008).

[47] ZHANG, M.-L., AND ZHOU, Z.-H. A k-Nearest Neighbor based algorithm for multi-label classification. In *IEEE International Conference on Granular Computing* (2005), vol. 2.

[48] ZHANG, M. L., AND ZHOU, Z. H. ML-KNN: A lazy learning approach to multi-label learning. *Pattern Recognition 40*, 7 (July 2007), 2038–2048.

[49] ZHANG, Y., HONG, J., AND CRANOR, L. CANTINA: A content-based approach to detecting phishing web sites. In *WWW: Proceedings of the international conference on World Wide Web* (2007).

Maverick: Providing Web Applications with Safe and Flexible Access to Local Devices

David W. Richardson and Steven D. Gribble
Department of Computer Science & Engineering
University of Washington

Abstract

Web browsers do not yet provide Web programs with the same safe, convenient access to local devices that operating systems provide to native programs. As a result, Web programmers must either wait for the slowly evolving HTML standard to add support for the device classes they want to use, or they must use difficult to deploy browser plug-ins to add the access they need.

This paper describes Maverick, a browser that provides Web applications with safe and flexible access to local devices. Maverick lets Web programmers implement USB device drivers and frameworks, like file systems or streaming video layers, using standard Web programming technologies such as HTML, JavaScript, or even code executed in a native client sandbox. These Web drivers and Web frameworks are downloaded dynamically from Web servers and executed by browsers alongside Web applications. Maverick provides Web drivers with protected access to the USB bus, and it provides Web drivers and frameworks with event-driven IPC channels to communicate with each other and with Web applications.

We prototyped Maverick by modifying the Chrome Web browser and the Linux kernel. Using Maverick, we have implemented: several Web drivers, including a USB mass storage driver and a Webcam driver; several Web frameworks, including a FAT16 filesystem and a streaming video framework; and, several Web applications that exercise them. Our experiments show that Web drivers, frameworks, and applications are practical, easy to author, and have sufficient performance, even when implemented in JavaScript.

1 Introduction

Web browsers do not yet provide Web programs with the same safe, convenient access to local devices that OSs provide to native programs. Digital devices like cameras, printers, scanners, smartphones, and GPS trackers are increasingly pervasive, yet browsers currently provide little support to Web applications for accessing them. The support that does exist is limited to a handful of HTML tags for accessing a small number of common device classes, such as Webcams or microphones.

Today, Web programmers that want to use unsupported or exotic local devices must either wait for HTML standards to evolve to include them, or they must implement, deploy, and support browser plug-ins that users may be reluctant to install. Such poor choices limit the functionality of Web applications and discourage the development and adoption of new, interesting local devices.

In this paper, we describe Maverick, an experimental browser that gives Web applications safe and flexible access to local USB devices. Maverick takes the aggressive approach of removing the responsibility for managing devices and device frameworks from the host operating system and empowering the browser to execute device drivers and frameworks alongside Web applications. Specifically, instead of requiring users to install USB device drivers into their host OS, Maverick dynamically finds, downloads, and executes *Web drivers* that are written with standard Web programming technologies like HTML, JavaScript, or Native Client (NaCl) [24], and that directly communicate with the USB devices they drive.

Similarly, instead of relying on device frameworks within the host OS, such as file systems or video frameworks, Maverick finds, downloads, and executes *Web frameworks* to provide Web applications with convenient high-level abstractions. Maverick permits Web applications to communicate directly with Web frameworks, which in turn communicate with Web drivers.

Maverick's approach has several advantages. Web developers can add support for new USB devices and make them immediately accessible to any Maverick user and Web application without waiting for updates to slowly moving standards bodies, browser vendors, or operating systems. Maverick also inherits many of the safety and security benefits of running drivers in user-level, such as insulating the OS from driver bugs.

Maverick addresses three main challenges:

1. **Security.** Maverick must isolate Web drivers and Web frameworks to prevent access to unauthorized devices or interference with unrelated Web applications and native software. Maverick uses existing JavaScript and NaCl sandboxes to contain Web drivers and frameworks. In addition, Maverick exposes a virtualized USB bus to Web drivers, granting each driver the ability to send and receive USB messages only to the devices for which it is authorized. So that Web drivers, frameworks, and applications can interact flexibly and efficiently, Maverick provides them with protected event-driven IPC channels.
2. **Performance.** Web drivers and frameworks must be efficient. We prototyped Maverick by modifying the Chrome browser and the Linux kernel, implemented several drivers and frameworks in both JavaScript and NaCl, and compared them to native Linux equivalents. Not surprisingly, user-level drivers in general, and JavaScript drivers and frameworks in particular, are significantly slower than their Linux counterparts. Nonetheless, our experiments demonstrate they are fast enough to support many interesting USB devices and applications. As well, we show that NaCl can achieve performance closer to that of in-kernel drivers and frameworks.
3. **Usability**. Maverick must avoid burdening users with making confusing and error-prone decisions on how to select trustworthy and compatible drivers and frameworks for Web applications. To do this, Maverick allows users to configure their browsers to trust one or more Maverick *domain providers.* A domain provider is a trusted third-party like Google or Microsoft that is responsible for selecting and bundling together a set of interoperable Web frameworks and drivers.

We have prototyped several Web drivers and frameworks and applications that exercise them, including: a USB mass storage driver, a Webcam driver, a FAT filesystem framework, and a streaming video framework. Overall, our experience with Maverick suggests that Web drivers and frameworks are straightforward to implement, and are safe and practical from a performance, security, and usability standpoint.

The rest of this paper is structured as follows. In Section 2, we present a brief overview of USB. Section 3 describes the architecture of Maverick, and Section 4 presents our prototype implementation. In Section 5, we evaluate the performance and security of Maverick and we showcase several applications. After discussing related work in Section 6, we conclude.

2 A Brief Overview of USB

Maverick exposes USB devices to Web drivers and applications. We chose to focus on USB for two reasons; first, it has become the predominant interconnect for most consumer devices, making it an attractive target. Second, since USB is message-oriented, it was relatively straightforward to expose USB message transmission and reception to JavaScript and NaCl. In contrast, we believe it would be much less natural to expose complex device interconnects, such as PCI, that use more architecturally-dependent features like DMA and memory-mapped I/O.

In this section, we provide a brief overview of USB device abstractions and protocols. Readers familiar with USB may choose to skip to Section 3.

2.1 USB Devices and Communication Channels

A USB bus connects a host, such as a laptop or desktop, to multiple peripheral devices over a star topology. Some USB devices consist of more than one *logical device*; for example, a Web camera might consist of a video camera and a microphone packaged together into a single physical box. Each of these logical devices would appear as a separate addressable entity on the USB bus.

A logical device consists of one or more communication *endpoints* associated with some specific function of the logical device. A host establishes a *pipe* to an endpoint to communicate with it. There is a one-to-one mapping between pipes and endpoint. Each endpoint and its corresponding pipe are typed. USB supports four kinds of pipes:

- *Control*. A control pipe facilitates the bidirectional exchange of small control messages used to query or control a device. The USB specification mandates that hosts must reserve 10% of the USB bus bandwidth for control traffic. All devices have at least one control endpoint.
- *Interrupt*. An interrupt pipe is a unidirectional channel used to convey messages from a device to a host. For example, USB keyboards generate interrupt messages when key press events occur. Interrupt messages are latency sensitive; hosts must poll interrupt endpoints sufficiently frequently to ensure responsiveness.
- *Isochronous*. An isochronous pipe is a unidirectional channel used to transfer a continuous stream of data, such as video frames or audio packets. Hosts

must schedule USB messages to provide a guaranteed amount of bandwidth to an established isochronous pipe. Isochronous pipes may experience occasional data loss.

- *Bulk.* Bulk pipes are unidirectional channels that provide reliable data transfer but no bandwidth guarantees. USB bulk storage devices and printers often use bulk pipes.

2.2 Host OS Abstractions and Duties

USB bus bandwidth is allocated and scheduled into time slices called *frames*. For high speed USB 2.0 devices, frames have a 125 micro-second interval, permitting up to 8,000 frames per second. The host operating system is responsible for scheduling USB packets into frames. Because USB mandates reserved bandwidth for isochronous pipes and responsiveness for interrupt pipes, the OS must queue USB packets, potentially delaying some to meet the scheduling demands of isochronous and interrupt traffic. Packets associated with bulk and control transfers are scheduled whenever a frame has available bandwidth not already consumed by interrupt or isochronous transfers.

Operating systems typically abstract USB transactions into a data structure called a USB request block (URB). A URB encapsulates a single, asynchronous interaction between the host and a device. The URB structure accommodates a request, data to be transferred, and a completion status message. Device drivers allocate and submit URBs to low-level, device-independent USB processing code within an OS. The OS schedules the transmission of messages and data indicated by the URB, and upon completion, notifies the driver.

While programmers can manually construct URBs for any type of data transfer, operating systems typically supply USB libraries with higher-level abstractions for constructing, sending, and receiving URBs based on the specified type of data transfer.

3 Architecture

Maverick splits a computer into two worlds (Figure 1): a legacy *desktop world* that contains the underlying host operating system, its drivers and frameworks, and applications, and the *Maverick world* that executes on top of a browser. Each world is isolated from the other with respect to the USB devices they can access: a given device is assigned to one of the two worlds, and the other cannot observe or influence it.

The assignment of USB devices to worlds is managed by the *USB world splitter* in the host OS. When a new USB device is detected, the splitter prompts the user

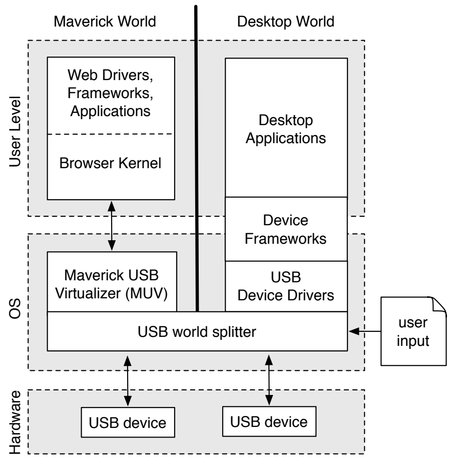


Figure 1: **Maverick "World Splitting."** Maverick splits computers into the legacy desktop world and the Maverick web world. USB devices are partitioned between the two worlds; each world runs its own drivers, frameworks, and applications.

to assign the device to either the desktop or Maverick world. The splitter then enforces this assignment when routing messages between devices and the two worlds.

3.1 The Desktop World

The desktop world exists to facilitate incremental deployment and backwards compatibility. Users can run unmodified legacy device drivers, frameworks, and applications in it: besides the presence of the USB world splitter, the desktop world is unaware of the existence of the Maverick world. Thus, in the desktop world, USB device drivers are installed into the host OS and communicate with devices using the USB core libraries provided by the OS. Drivers abstract away the details of specific devices and interfaces with frameworks, such as file systems, network stacks, and video frameworks. A framework, which is typically implemented partially in the host OS kernel and partially as sets of user-mode libraries, provides high-level, device-independent abstractions to applications.

3.2 The Maverick World

The Maverick world has some similarity to the legacy desktop world, in that the structural relationship between drivers, frameworks, and applications is the same in both cases. However, unlike the legacy world, in Maverick these components are (1) dynamically downloaded rather than installed, (2) implemented using Web programming technologies such as JavaScript and NaCl, and (3) executed by a browser kernel entirely at the user-level.

Figure 2 shows the architecture of the Maverick world.

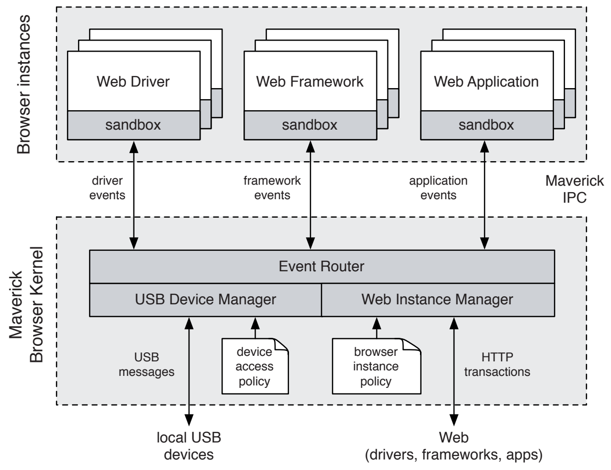


Figure 2: **The "Maverick World."** The Maverick world consists of a trusted browser kernel and untrusted browser instances. The kernel manages instances, provides IPC channels, and relays USB messages between authorized drivers and local USB devices.

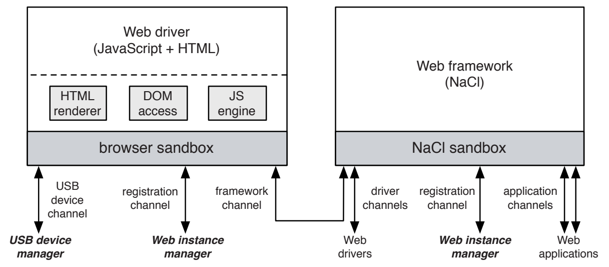


Figure 3: **Maverick Web Browser Instances.** Web drivers, frameworks, and applications execute inside browser instances. Instances contain JavaScript+HTML or NaCl code, and they interact with each other and the browser kernel via IPC channels.

At a high-level, the world is deconstructed into two components: untrusted *browser instances* and the trusted *browser kernel*. We describe each component below.

3.2.1 Maverick Browser Instances

Maverick browser instances contain untrusted code provided by a remote Web service. Browser instances are sandboxed from each other, the browser kernel, and the legacy desktop world. Instances can implement device driver functionality, framework functionality, or application-level functionality using standard Web programming technologies. We focus on two variants: JavaScript and NaCl instances (Figure 3). Similar to browsers like Chrome [20], a JavaScript instance contains DOM bindings, a JavaScript interpreter, and an HTML renderer, in addition to the browser instance's code itself. A NaCl instance contains x86 code that is verified and contained by the NaCl sandbox.

When an instance is instantiated, it has access to a registration IPC channel provided to it by the browser kernel. When the instance has initialized itself, it uses the registration channel to alert the browser kernel, which then establishes point-to-point IPC channels between the instance and other instances with which it must communicate. We discuss the policy by which the browser kernel interconnects devices, frameworks, and applications in Section 3.3.

3.2.2 The Maverick Browser Kernel

The Maverick browser kernel serves two main roles. First, it provides the standard functions of a "typical" browser kernel, including: managing standard Web storage like cookies, cache, and history; providing network access to browser instances; enforcing the same-origin policy; and, implementing the browser's user interface [20]. We have not modified these functions, and will not describe them further in this paper.

Second, the browser kernel provides Web drivers with safe access to local USB devices and facilitates secure communication between drivers, frameworks, and applications. The Maverick browser kernel consists of three modules: (1) the USB device manager, (2) the Web instance manager, and (3) the event router. The USB device manager stores device information such as vendor and device IDs for devices made available to Maverick via the world splitter. As well, the device manager establishes the channel between a Web driver and its local USB device: messages transmitted by a driver are routed to device manager, which verifies that they are properly formatted and addressed to the driver's authorized device before relaying them down to the host OS.

The Web instance manager downloads and instantiates Web driver and framework instances. The Web instance manager is also responsible for establishing the point-to-point IPC channels between drivers, frameworks, and applications. The IPC channels are managed by the event router: each channel implements a reliable, point-to-point FIFO queue. Browser instances communicate over these channels using events that contain a name, untyped variable-length payload, and destination. Within these constraints, browser instances are free to define and use any communication protocol.

3.2.3 The Maverick USB Virtualizer

The Maverick USB Virtualizer (MUV) is a device-agnostic USB driver that lives inside the host OS. It virtualizes the USB API provided by the core USB libraries in the kernel, translating and packaging up USB messages from the kernel's USB core into events that delivered to the browser kernel, and from there, routed to the appropriate Web driver. USB events sent from Web drivers are routed from the browser kernel to the MUV and translated into appropriate calls into the host OS's USB core.

3.3 Naming and Binding of Web Drivers and Frameworks

As previously mentioned, Maverick must decide how to select, download, and interconnect Web driver and framework instances to applications. The policies it uses to do this are critical, as they impact safety, reliability, and compatibility: the policy must prevent users from being exposed to malicious drivers and frameworks, ensure that the set of drivers and frameworks that are instantiated are compatible with each other and with the applications, and that the applications can depend on a stable, coherent set of framework abstractions and interfaces.

We have experimented with several policies, and we describe two below. However, we feel that we are still just beginning to explore this topic: there are still many difficult and interesting research challenges to solve.

Application-driven. Under this policy, each Web application declares to Maverick the URLs of the frameworks that it requires. Similarly, each instantiated framework declares to Maverick the URLs of the drivers that it trusts and is compatible with. When a user navigates to an application, Maverick instantiates its declared frameworks, observes which drivers the framework is willing to have loaded, and identifies the subset of drivers that match available, underlying USB devices. Before the drivers are loaded and bound to USB devices, Maverick prompts the user for authorization: the prompt informs the user of the URLs of the application, frameworks, drivers, and devices that are involved.

This policy is simple, but has many disadvantages. First, since different applications may select different frameworks, there is no opportunity for frameworks (and their drivers) to form a coherent, interoperable set of abstractions. Two different applications may cause vastly different frameworks to load, and those frameworks will have no basis for interoperating or sharing the underlying devices between applications. Second, users likely have no basis to make reasonable authorization decisions: users know they want to use an application, but they cannot know whether the frameworks and drivers selected are trustworthy, especially since URLs in and of themselves are not particularly informative.

Domain-driven. Under this policy, users can configure their browsers to trust one or more Maverick *domain providers.* A domain provider is a trusted third-party that is responsible for selecting and bundling together a set of interoperable Web frameworks and drivers. Users name domain providers by URL; the document behind this URL contains the list of authorized framework and driver URLs. We anticipate that organizations like Google, Microsoft, or the FSF could be domain providers.

When loaded, an application declares to Maverick the domain provider it wants to use, and the frameworks within that domain that it requires. If the user has authorized the specified domain provider, Maverick will demand load the required frameworks. Once loaded, the framework declares the Web drivers that it requires; Maverick verifies that they are on the domain's authorized list, and loads them if so. Frameworks are responsible for prompting users before granting a Web application access to particular device or device class.

A given framework within a domain is loaded only once. Multiple applications that use the same domain are bound to that single instance, permitting the frameworks to facilitate sharing across those applications, as appropriate. Of course, this implies that the frameworks must provide adequate protection as well!

The benefit of the domain-driven approach is that the user makes a single higher-level trust decision and delegates to the trusted domain provider the responsibility for selecting appropriate, safe, and interoperable frameworks and drivers. As well, the domain provider can engineer its frameworks to be interoperable with each other, providing much of the same kind of API coherence, resource sharing, and protection that today's operating systems provide to applications. A disadvantage of this approach is that it may cause the user to place too much trust in a small number of domain providers, and it could lead to significant fragmentation of applications across domain providers.

4 Implementation

We implemented a prototype of Maverick by modifying the Chrome Web browser and the Linux kernel. Since Maverick uses Chrome, we inherit its process model: each browser instance (driver, framework, or application) executes in its own, separate Linux process. As well, Chrome's browser kernel executes as its own separate, trusted process. Rather than attempting to integrate our Maverick browser kernel components into Chrome's browser kernel, we bundled them into their own trusted process.

Using our prototype, we implemented several Web drivers, frameworks, and applications, both in JavaScript and in NaCl. In this section, we describe our aspects of the implementation, focusing on non-obvious issues.

4.1 Event Framework

Maverick browser instances communicate with each other and with the browser kernel using an asynchronous event-driven model, facilitating a natural integration with today's event-driven Web programming languages and browser abstractions. Our events are untyped, meaning that Web drivers, frameworks, and applications can exchange arbitrary data with each other, giving them the flexibility to define and implement their own high-level interfaces and protocols.

Maverick events have three fields: the name of the event, unstructured data payload, and a routing target address. The sender provides each of these fields, specifying a routing target of muv, $driverID$, $frameworkID$, or $applicationID$. This target address tells Maverick to route the event to either the MUV (via the USB device manager), or to a Web driver, framework, or application with the provided ID. The Maverick event router maintains routing tables to determine whether the sender has a valid communication channel to the target Maverick instance with the specified ID. If so, Maverick routes the event to the target's event queue.

4.1.1 Events in NaCl Instances

A programmer might choose to implement a Web driver, framework, or application using NaCl. To integrate our event framework into NaCl, we had to solve three problems. First, we needed to create IPC channels between NaCl instances and our Maverick browser kernel process; we implemented UNIX domain sockets as our channel transport. Second, we needed to be able to marshal Maverick events over that transport; we created a RPC layer using Google's protobuf library to do this [9].

Third, we needed to exchange events with the Web instance code. To do this, we extended the NaCl system call interface to permit sandboxed code to send events over the IPC channel, and to synchronously receive the next event from the channel. We also provide instance programmers with an (untrusted by Maverick) support library that spawns a NaCl thread to loop, issuing blocking receives on the IPC channel and dispatching events to a programmer-supplied callback function. This library provides programmers with convenient `createEvent()` and `postEvent()` functions, as well as functions for registering event handlers.

4.1.2 Events in JavaScript Instances

JavaScript programmers that want to use Maverick can include a convenience JavaScript library that we supply. This library provides an API that is syntactically similar to the API provided by our NaCl support library. This simplifies porting a NaCl Web driver to JavaScript, and vice-versa. Specifically, our library implements the following functionality:

1. To implement the IPC channel to the browser kernel, the library includes a hidden NaCl component that implements the IPC mechanisms we described above. We could have instead modified Chrome's JavaScript interpreter to expose this IPC channel, eliminating the need for NaCl support in the browser, but for the sake of simplicity we chose to leverage our existing NaCl code.
2. To deliver events to the instance programmer's JavaScript, we take advantage of the NPAPI interface provided by NaCl to invoke a callback handler on an object exposed to the instance through the DOM. The library creates a separate DOM object for each channel made available to the Web instance by the browser kernel.
3. The JavaScript library provides the programmer with convenience routines for base64 encoding and decoding binary data within event payloads, as well as protobuf support for exchanging structured messages with other browser instances.

4.2 The Maverick Browser Kernel

As previously mentioned, we implemented the trusted Maverick browser kernel to run as a separate process, independent of the Chrome browser kernel. At its heart, the event router component within the Maverick browser kernel is a threaded RPC server that establishes IPCs to browser instances and the MUV, and processes and routes events between them. We implemented the kernel in C++; for each browser instance, we allocate an event queue and spawn dedicated send and receive threads.

The Web instance manager component defines and processes browser instance registration events on behalf of the browser kernel. When a browser instance begins executing, it is expected to send one of these events over its registration channel. Similarly, the USB device manager component defines and processes USB message events, relaying them between authorized Web drivers and the MUV. Web drivers can create control or bulk URBs, start or terminate isochronous streams, and receive isochronous stream data (see Section 2). We have not yet implemented support for USB interrupt messages, as our experimental drivers have not required them, but doing so would be simple.

4.3 The Maverick USB Virtualizer and World Splitter

The Maverick world splitter is implemented as a dynamically loaded Linux USB device driver. When a USB device is attached to the host, the world splitter prompts the user to decide whether to associate the device with Maverick or not. For devices associated by the user with Maverick, the Linux USB subsystem relays the device's hardware IDs to the browser kernel which stores these tags in a registered device ID table that can later be used to bind compatible Web drivers to registered devices. The world splitter then routes all received USB callbacks up to the Maverick USB virtualizer. Devices not associated with Maverick by the user are bound to native Linux device drivers.

For every attached USB device, the MUV spawns a kernel thread devoted to handling that device. This thread maintains an RPC connection to the Maverick browser kernel, which it initially uses to register the device's vendor and product IDs. USB events are shuttled between the browser kernel and the MUV over these RPC connections; we ported a subset of protobuffer support into the Linux kernel to marshal events over these connections. The MUV dispatches events to the Linux core USB library.

To optimize the transmission of isochronous data from the Linux kernel to a Web driver, the MUV shortcuts the process of sending an isochronous URB to the device. When an isochronous URB completes, the MUV packages up the URB into a *completed_urb* event and sends it to the Web driver via the browser kernel. The MUV then immediately submits the next isochronous URB on behalf of the Web driver. As we show in Section 5, this optimization helps to improve bandwidth for streaming devices by eliminating some of the additional USB latency added by Maverick.

4.4 Example Drivers, Frameworks, and Applications

We have implemented several experimental Web drivers and frameworks. We built two versions of each, one in JavaScript and one in C++ using NaCl. Note that due to JavaScript's lack of raw data support, raw data in JavaScript drivers and frameworks are formatted as base64-encoded strings:

- **USB mass storage driver.** The USB mass storage specification consists of a "bulk-only" transport protocol used to initialize a device, exchange data, and handle error conditions. The SCSI command protocol is layered on top of this; SCSI commands are embedded inside bulk-only protocol messages. Our drivers implement enough of the SCSI protocol to initialize attached drives, probe for capacity, and read and write blocks. We have primarily experimented with SanDisk USB flash drives.
- **Logitech C200 Webcam driver.** Our Webcam drivers interact with the Logitech camera using Logitech's proprietary protocol; we ported a subset of this protocol using a Linux driver as reference. The drivers extract 320x240 resolution video frames at a rate of 30 frames per second over an isochronous stream, and post events containing raw frame data to an attached video framework.
- **FAT16 file system framework.** We implemented FAT16 compatible filesystem frameworks that expose file create, read, write, and delete operations to attached Web applications. The frameworks are also able to format and partition flash drives.
- **Video stream rendering framework.** Our video frameworks first convert raw frame data into a JPEG image; this only requires reformatting the frame data by adding an appropriate JPEG header. Next, the frameworks encode the images into base64 data URI strings [12], and post events containing them to attached Web applications. Web applications can then blit JPEGs to the screen by updating an HTML image tag's source attribute with the received data URI string.

To test these drivers and frameworks, we wrote a video streaming Web application called PhotoBooth. PhotoBooth allows users to view live video from an attached Webcam, capture the Webcam's live video stream and save it to a flash drive, or read and display a previously captured video stream from a flash drive.

5 Evaluation

In this section, we examine three aspects of our Maverick prototype: (1) its performance, (2) its security implications, and (3) its suitability for building device-enabled Web applications. We have not yet optimized our prototype implementation, so performance numbers should be considered an upper bound for a Maverick system.

5.1 Performance

We evaluate the performance of our prototype in two parts. First, to understand the overhead of moving device drivers out of the kernel and into the Web browser, we provide microbenchmarks that compare the performance of Web drivers to their native Linux drivers. Then, we evaluate the end-to-end performance of Maverick Web applications. Our measurements were gathered on an 8-core, 2GHz Intel Xeon machine with 6GB of RAM, running our modified Linux kernel.

5.1.1 Microbenchmarks

Our first set of experiments quantify the latency, throughput, and CPU utilization of Maverick Web drivers. To do this, we constructed a series of microbenchmarks that exercise both the JavaScript and NaCl versions of our

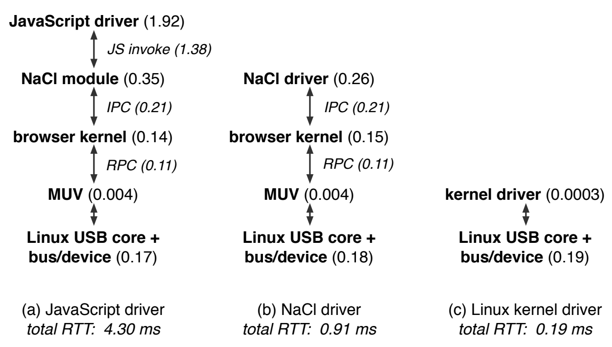


Figure 4: **Driver Latency.** The event flow and latencies of transacting a USB bulk URB with a USB flash drive, using (a) a JavaScript driver, (b) a NaCl driver, and (c) a Linux kernel driver. Numbers in parenthesis indicate time spent in that component or channel, in milliseconds.

storage and Webcam drivers. For a fair comparison, we also back-ported our NaCl drivers to run as native Linux drivers.

To measure the latency of performing USB operations from Web drivers, we instrumented our system to measure the time spent in various Maverick components when sending a single bulk message URB to a USB flash drive and receiving the response. Figure 4 depicts the involved components, the time spent in them, and the total round-trip time between the driver and device. Results are reported for each of the JavaScript, native client, and kernel drivers, averaged over 600 trials.

Not surprisingly, the JavaScript driver has the highest total latency. Its dominant factors are the latency of dispatching an event from the NaCl glue to JavaScript and the JavaScript driver execution itself. The NaCl glue overhead could be eliminated by modifying Chrome's JavaScript interpreter to deliver events rather than using our indirect NaCl route. The NaCl driver avoids these sources of overhead, but is roughly four times slower than the kernel driver due to the marshaling and transport of events from the Linux kernel to the browser.

To measure Web driver throughput and CPU utilization, we tested whether our Logitech C200 drivers were efficient enough to keep up with the isochronous URB transaction rate of 250 URB/s corresponding to a frame rate of 30 frame/s. For this experiment, we isolated driver performance by modifying our drivers to simply receive and drop URBs without further processing. Table 1 shows the URB rate sustained by each of our drivers, as well as the total CPU utilization (out of 8 cores). Only the JavaScript driver is unable to sustain the full streaming rate, as it saturates a single core, resulting in a loss of roughly one frame per second. However, future improvements to JavaScript execution engines should easily nudge this driver above the required 250 URB/s.

	sustained URB rate	CPU utilization
JavaScript Web driver	240 URB/s	13.4%
NaCl Web driver	250 URB/s	2.7%
Linux kernel driver	250 URB/s	0.25%

Table 1: **USB Webcam Driver Throughput.** This table shows the URB transaction rate that each driver could sustain, as well as the CPU utilization of the machine. The camera emits isochronous messages at a rate of 250 URBs per second.

5.1.2 End-to-End Performance

We now evaluate the end-to-end performance of Web applications in Maverick. Our application benchmarks exercise two drivers (USB storage and Webcam) and two frameworks (FAT16 and video). As before, we will compare using JavaScript, NaCl, and native kernel versions of the drivers.

To measure end-to-end latency, we instrumented the system to measure time spent in each major Maverick component when sending a storage operation from PhotoBooth and receiving a response. These components consist of the application, framework, driver, and the rest of Maverick. Time spent in Maverick includes all IPC and RPC channels, the browser kernel, the Linux kernel, and the USB bus and device itself. All reported measurements represent the average of 600 trials.

Table 2 shows our results. Not surprisingly, the JavaScript stack has the highest latency. Similar to our microbenchmark results, this overhead is mostly due to the 10 expensive IPC crossings between native client and JavaScript that are required because our prototype does not implement event delivery directly in Chrome. The JavaScript driver is also slow at processing byte-level data. The native client stack avoids most of these IPC overheads, but is still slower than the native desktop stack because of the cost of routing events from the kernel into user-space drivers and frameworks.

To measure Maverick's end-to-end throughput and CPU utilization, we benchmarked file create+write and file read operations on 20KB-sized files in PhotoBooth with the FAT16 framework and USB flash driver. As in earlier experiments, we compare three cases: the PhotoBooth application driving a JavaScript framework and a JavaScript driver, PhotoBooth driving a NaCl framework and a NaCl driver, and a native C application that issues similar file workload as PhotoBooth driving an in-Linux-kernel combined framework and driver. Our benchmarks leveraged Maverick's ability to run drivers and frameworks in parallel in separate Chrome processes and pipelines file system operations through them.

Table 3 shows our results. For all three versions, the file create+write benchmark achieved lower throughput than file reads, since file creation and file append both re-

PhotoBooth latency	time spent in component				end-to-end latency
	application	framework	driver	Maverick	
JavaScript driver/framework	0.8ms	1.7ms	3.3ms	8.7ms	14.5ms
NaCl driver/framework	0.8ms	0.35ms	0.44ms	3.1ms	4.7ms
native "app/FW" and Linux kernel driver	0.05 ms ("app")	0.22ms (kernel driver + RPC)			0.27ms

Table 2: **PhotoBooth Latency.** The total roundtrip time for sending an event from the PhotoBooth Web application through a framework and driver to a USB flash device. We compare a JavaScript driver and framework, a NaCl driver and framework, and a native desktop "application" using a Linux kernel driver.

quire additional FAT16 operations to update file system metadata. As before, the JavaScript driver and framework are the slowest, while NaCl achieves closer performance to the in-kernel driver and framework.

As a final test, we measured the frame rate at which PhotoBooth could render a live video stream for the JavaScript and NaCl versions of the driver and framework. The NaCl version could render at the full rate of 30 frames per second, but inefficient, byte-level operations to process URBs and video frames in the driver became a bottleneck of the JavaScript version, achieving only 14 frames per second.

5.1.3 Performance summary

Unsurprisingly, JavaScript and the use of user-level drivers introduce significant performance overhead relative to in-kernel, native device drivers. However, we believe that JavaScript and NaCl drivers are sufficiently fast to be practical for many USB device classes, including storage devices and video cameras, in spite of the fact that we have not attempted to optimize them, or our user-mode driver framework, in any significant way.

5.2 Security Implications

There are serious security implications to exposing local USB devices to Web programs using Maverick. In this section, we define Maverick's threat model and use it to describe the possible attacks against Maverick. For each attack, we discuss how Maverick defends against the attack and identify where our current security mechanisms are lacking.

5.2.1 Threat Model

Our threat model is similar to that assumed by modern browsers when handling untrusted extensions [1]. We assume that the browser kernel, the browser instance sandboxes, and the Maverick components inside the host OS are correctly implemented and vulnerability-free. Thus, Web drivers, frameworks, and application instances can only be directly attacked through Maverick's IPC channels; we assume that attackers cannot directly access or modify DOM objects in Web instances, or attack Web instances by corrupting the browser kernel.

	20KB file create+write		20KB file read	
	throughput	CPU util.	throughput	CPU util.
JavaScript Web driver	11.2 files/s	27%	28.0 files/s	24%
NaCl Web driver	44.1 files/s	18%	134.4 files/s	18%
Linux kernel driver	65.5 files/s	0.4%	509 files/s	1.1%

Table 3: **File Throughput.** The throughput and CPU utilization of two file benchmark applications, 20KB file creates+writes and 20KB file reads, in three cases.

5.2.2 Attacks, Defenses, and Limitations

Given this threat model, Maverick is vulnerable to two broad classes of attacks: (1) attacks aimed at compromising *trusted* Web instances that are benign but buggy, and (2) attacks aimed at tricking Maverick into running *malicious* Web instances.

Benign-But-Buggy Web Instances. This class of attacks exploits bugs in trusted Web instances to expose a user's local devices to an attacker. Local devices often contain highly sensitive user information that an attacker might like to access, delete, modify, or corrupt. Examples include data stored on USB storage devices, environmental information obtained via audio, video, and GPS input devices, and sensitive information transmitted to printers or other peripheral devices.

Because trusted frameworks and device drivers in Maverick are Web programs, buggy implementations can be vulnerable to conventional Web-based attack techniques such as cross-site scripting, cross-site request forgery, and framing attacks. Additionally, if trusted Web instances are served over an insecure channel, attackers could mount man-in-the-middle attacks to eavesdrop on and hijack local devices.

Maverick does not provide any additional mechanisms for defending against such attacks beyond those already provided by browsers, nor does it make mounting these attacks any easier. Instead, Maverick elevates the consequences of writing buggy Web programs. Web developers will need to adopt best practices for eliminating common vulnerabilities, and ensure that Web instances are transmitted only through SSL-protected channels.

Maverick's domain-driven naming, binding, and authorization policy described in Section 3.3, in which a user configures their browser to trust a domain provider's choice of frameworks and drivers, makes it so that the average user need only trust a small number of well-known Web instance providers such as Google or Microsoft. Of course, power users that opt-out of this default setting and choose to adopt the more flexible application-driven policy will face an increased burden in vetting the quality of Web instances.

Malicious Web Instances. This second class of attacks relies on an attacker being able to run a malicious Web driver or framework in the user's browser. If successful, the consequences could be serious: because Maverick exposes the USB subsystem to Web drivers, a malicious Web driver can not only control the associated local device, but can inject arbitrary device commands into the USB subsystem. This could allow the attacker to introduce malware onto the device, or exploit bugs in the USB software stack that could give the attacker unfettered access to the user's system.

Maverick's security mechanisms are designed primarily to prevent attackers from running malicious Web instances in the first place. To try to trick Maverick into running a malicious Web instance, attackers might leverage existing network-based and social engineering attacks.

Example network-based attacks include DNS rebinding and DNS cache-poisoning which redirect valid DNS mappings in the browser to malicious URLs. Maverick could defend against these attacks by using stronger naming mechanisms such as secure DNS or certificate-based naming. Maverick also protects against social engineering attacks by relying on the domain-driven naming, binding, and authorization policy to prevent users from authorizing malicious drivers or frameworks.

If a malicious Web instance somehow does manage to run in the browser despite these safeguards, Maverick currently has limited ability to protect the local system. At best, Maverick's IPC channels serve to help mitigate the damage an attacker can do by limiting the compromised device driver's access to a single USB device. In general, once an attacker has the ability to run a malicious Web driver or framework, we consider the user's computing environment to be compromised.

5.3 Experiences with Maverick

Our experience with programming Maverick drivers and frameworks has been positive. JavaScript and NaCl drivers and frameworks enjoy Maverick's clean event model. Programmers do not need to worry about kernel synchronization, pitfalls of interrupt-context code, or the vagaries of kernel memory allocation. If a driver is buggy, that browser instance will crash, but other browser instances and the host OS continue to execute. NaCl makes it easy to port existing C kernel drivers and frameworks into Maverick. While JavaScript is slower and not yet ideally suited for manipulating raw, binary data, JavaScript code is dynamic and flexible, so managing complexity like callback function pointers is straightforward. Overall, we found that for the majority of devices we considered, the advantages of programming in higher-level languages and abstractions far outweighed any performance limitations.

To showcase the power and flexibility of Maverick, we built two additional Web applications. The first, called SquirtLinux, makes it trivial to install Linux onto a USB flash drive. Linux thumbdrives are useful for rescuing data from a damaged system or carrying a portable, secure boot environment. SquirtLinux consists of combined Web application and framework instance that exploits our existing JavaScript-based USB mass storage driver. SquirtLinux communicates with a Web server, having it prepare a "Puppy Linux" USB installation image on behalf of the user; it then uses AJAX to pipeline downloading blocks of the image with writing it to an attached drive. As a result, the user simply needs to find a Maverick browser, attach their flash drive, browse to the SquirtLinux service, and press a single button to manufacture a bootable thumbdrive.

Our second application, WebAmbient, uses a Delcomm USB LED indicator to build an ambient display. The LED supports a custom USB protocol that lets it be programmed to emit any color as a combination of red, green, and blue light. We wrote a JavaScript Web driver and Web framework for it that exposes a high-level color toggling interface to applications. Next, we wrote a Web mashup that fetches real-time stock prices, making the LED grow redder as price drops, greener as it increases, and blue if it is flat. Maverick's flexibility made it simple for us to support the custom USB protocol and expose high level functions without requiring any change to the host OS, browser, or HTML standards.

6 Related Work

Maverick builds upon architectural features and techniques explored by prior work. We discuss related work below.

6.1 User-Level Drivers and Frameworks

Maverick moves device driver and framework code out of the OS and into (user-level) Web applications. Our approach of running drivers and frameworks as Web applications is new, but the general approach of deconstructing a monolithic OS and moving its components to the user-level is well studied. Mach [6] and L3 [16] explored microkernels, small OS kernels that provide basic hardware abstraction and rely on user-level servers to implement major OS subsystems. Maverick also shares features with exokernels [14], which allocate and protect hardware resources within a small, trusted kernel and delegate the higher-level abstractions to user-level programs.

Prior work has explored OS support for user-level drivers. Decaf [22] and Microdrivers [7] use static program analysis and code annotations to automatically partition kernel drivers into user-space and kernel components, leaving just the performance critical components (like I/O) in the kernel. Other user-level driver frameworks have been implemented on top of Linux [15, 5] and Windows [17].

Maverick's virtualization of the USB hub is perhaps closest to the open source libusb [4] and Javax.USB [11] projects, which expose the Linux USB core API to user-level code. Like libusb and Javax.USB, Maverick exposes familiar USB API functions, but unlike libusb and Javax.USB, Maverick Web drivers are untrusted and dynamically located, downloaded, and executed.

6.2 Browser Architecture

Maverick inherits many of the safety and reliability benefits of the Chrome browser. Chrome maps Web applications into OS processes, providing better isolation and resource management [20, 19, 21]. Chrome's process model allows Maverick instances to run in parallel on multicore systems, and its sandbox isolates untrusted code.

Other research browsers have similar properties. For example, Gazelle [23] isolates Web applications into processes using the same origin policy (SOP), placing mashup content from separate Web origins in different trust domains. Gazelle plugins execute in their own processes, and the browser kernel protects updates to the display. OP decomposed the browser architecture into multiple trusted components, but it did not provide full compatibility with existing sites [10]. SubOS provided a process abstraction for Web applications, but did not explore in detail a process model or the interactions between processes [13]. Tahoma provided safe and flexible isolation between Web applications by embedding each in a Xen virtual machine [2]. Tahoma did not consider the problem of exposing local devices to Web applications, but rather provides them with a limited set of standard virtual devices.

6.3 Web Application Access to Devices

ServiceOS's browser architecture is designed to isolate Web applications and allow them to directly monitor and manage device resources [18]. As with Chrome OS [8], ServiceOS envisions itself as a browser OS. However, ServiceOS exposes devices to Web applications by including DOM objects and system calls in the browser kernel on a device-per-device basis. This solution is clean and simple, but suffers from the same drawbacks as adding new HTML tags to the HTML specification: new device support requires modifications to the browser. In contrast, Maverick lets Web developers add support for new devices and functionality *without* requiring browser vendor or OS cooperation.

The Javax.USB [11] project allows signed Java applets to access USB devices. This work is complimentary to the Web drivers piece of Maverick. However, Javax.USB requires use of the heavy-weight (and potentially unsafe) Java browser plugin, and provides no explicit support for safe IPC channels for direct framework and application communication with drivers.

6.4 Client-Side Code Sandboxes

Maverick uses Native Client [24] to provide a safe, client-side sandbox for executing x86 Web drivers and frameworks built with languages like C and C++. These languages handle certain aspects of driver development such as byte-level data manipulation more efficiently than JavaScript, and supporting them in Maverick makes porting existing kernel drivers much easier. Similar technologies to Native Client exist, most notably Xax [3]. Other client-side sandboxes include Flash, Java, Silverlight, and ActiveX. Although our current implementation focuses only on JavaScript and Native Client, Maverick could support these sandboxes in the future.

7 Conclusions

The Maverick browser gives Web applications safe access to local USB devices, and permits programmers to implement USB device drivers and frameworks using standard Web programming technologies like JavaScript and native client (NaCl). With Maverick, Web drivers and Web frameworks are downloaded dynamically from servers and safely executed by the Maverick browser kernel, allowing new drivers and frameworks to be implemented, distributed, and executed as conveniently as Web applications.

The main challenges faced by Maverick are safety and performance. Our prototype system, built by extending the Chrome browser and Linux kernel, relies on existing sandboxes to isolate Web drivers and frameworks from each other, the Maverick browser, and the host operating system and applications. As well, our system is architected to be flexible in supporting a range of policies and trust models for authorizing Maverick driver, framework, and application access to specific USB devices, and for resolving which drivers and frameworks ought to be downloaded and executed to satisfy application dependencies.

We prototyped several JavaScript and NaCl drivers and frameworks, and evaluated them using microbenchmarks and application workloads. While our measurements confirm that JavaScript Web drivers and frameworks suffer from much higher latency and lower throughput than NaCl or Linux equivalents, we also showed that they perform sufficiently well to drive interesting USB devices, including Webcams and USB flash storage devices. Finally, we described two Web applications that showcase the flexibility and power of the Maverick approach.

Acknowledgments

We thank Paul Gauthier, Charlie Reis, Roxana Geambasu, and Ed Lazowska for their feedback and help. We also thank our anonymous reviewers, and our shepherd Jon Howell, for their guidance. This work was supported by NSF grants CNS-0627367 and CNS-1016477, the Torode Family Career Development Professorship, and gifts from Intel Corporation and Nortel Networks.

References

[1] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS 2010)*, San Diego, CA, February 2010.

[2] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A safety-oriented platform for Web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Washington, DC, May 2006.

[3] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. Leveraging legacy code to deploy desktop applications on the Web. In *Proceedings of OSDI 2008*, San Diego, CA, December 2008.

[4] Daniel Drake and Peter Stuge. libusb. `http://www.libusb.org/`.

[5] J. Elson. FUSD: A Linux framework for user-space devices. `http://www.circlemud.org/~jelson/software/fusd/`.

[6] Alessandro Forin, David Golub, and Brian Bershad. An I/O system for Mach 3.0. In *Proceedings of the USENIX Mach Symposium*, Monterey, CA, November 1991.

[7] Vinod Ganapathy, Matthew J. Renzelmann an Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of Microdrivers. In *Proceedings of ASPLOS 2008*, Seattle, WA, March 2008.

[8] Google. Chromium OS. `http://www.chromium.org/chromium-os`.

[9] Google. Protocol buffers. `http://code.google.com/p/protobuf/`.

[10] Chris Grier, Shuo Tang, and Samuel T. King. Secure Web browsing with the OP Web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, Washington, DC, May 2008.

[11] IBM. Javax.USB. `http://www.javax-usb.org/`.

[12] IETF. RFC 2397: The data URL scheme. `http://tools.ietf.org/html/rfc2397`.

[13] Sotiris Ioannidis and Steven M. Bellovin. Building a secure Web browser. In *Proceedings of the FREENIX of the USENIX ATC*, Boston, MA, October 2001.

[14] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hctor M. Briceo, Russell Hunt, David Mazires, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of SOSP 1997*, Saint Malo, France, September 1997.

[15] Ben Leslie, Peter Chubb, Nicholas Fitzroy-dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5), 2005.

[16] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a μ-kernel based OS. *SIGOPS Operating System Review*, 25(2), 1991.

[17] Microsoft. Architecture of the user mode driver framework. `http://www.microsoft.com/whdc/driver/wdf/UMDF-arch.mspx`, February 2007.

[18] Alexander Moshchuk and Helen J. Wang. Resource management for Web applications in ServiceOS. In *MSR-TR-2010-56*, Redmond, WA, May 2010.

[19] Charles Reis, Brian Bershad, Steven D. Gribble, and Henry M. Levy. Using processes to improve the reliability of browser-based applications. In *University of Washington Technical Report UW-CSE-2007-12-01*, Seattle, WA, 2007.

[20] Charles Reis and Steven D. Gribble. Isolating Web programs in modern browser architectures. In *Proceedings of EuroSys 2009*, Nuremberg, Germany, April 2009.

[21] Charles Reis, Steven D. Gribble, and Henry M. Levy. Architectural principles for safe Web programs. In *Proceedings of HotNets 2007*, Atlanta, GA, November 2007.

[22] Matthew J. Renzelmann and Michael M. Swift. Decaf: Moving device drivers to a modern language. In *Proceedings of the USENIX ATC*, San Diego, CA, June 2009.

[23] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal OS construction of the Gazelle Web browser. In *Proceedings of USENIX Security 2009*, Montreal, Canada, August 2009.

[24] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.