UCognito: Private Browsing without Tears

Meng Xu, Yeongjin Jang, Xinyu Xing, Taesoo Kim, and Wenke Lee
Georgia Tech
{meng.xu, yeongjin.jang, taesoo}@gatech.edu, {xinyu.xing, wenke}@cc.gatech.edu

ABSTRACT

While private browsing is a standard feature, its implementation has been inconsistent among the major browsers. More seriously, it often fails to provide the adequate or even the intended privacy protection. For example, as shown in prior research, browser extensions and add-ons often undermine the goals of private browsing. In this paper, we first present our systematic study of private browsing. We developed a technical approach to identify browser traces left behind by a private browsing session, and showed that Chrome and Firefox do not correctly clear some of these traces. We analyzed the source code of these browsers and discovered that the current implementation approach is to decide the behaviors of a browser based on the current browsing mode (i.e., private or public); but such decision points are scattered throughout the code base. This implementation approach is very problematic because developers are prone to make mistakes given the complexities of browser components (including extensions and add-ons). Based on this observation, we propose a new and general approach to implement private browsing. The main idea is to overlay the actual filesystem with a sandbox filesystem when the browser is in private browsing mode, so that no unintended leakage is allowed and no persistent modification is stored. This approach requires no change to browsers and the OS kernel because the layered sandbox filesystem is implemented by interposing system calls. We have implemented a prototype system called UCOGNITO on Linux. Our evaluations show that UCOGNITO, when applied to Chrome and Firefox, stops all known privacy leaks identified by prior work and our current study. More importantly, UCOGNITO incurs only negligible performance overhead: e.g., 0%-2.5% in benchmarks for standard JavaScript and webpage loading.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; K.4.1 [**Computers and Society**]: Public Policy Issues—*Privacy*

Keywords

private browsing; browser implementation; filesystem sandbox

1. INTRODUCTION

Private browsing mode is an essential security feature that has been implemented in major web browsers, such as Firefox, Chrome and Opera. The main goal of private browsing is to let users browse the web without storing local data that could provide some indication of the user's activities during a browsing session. For example, by setting a browser to private browsing mode, a user can browse the Internet without saving any information about which websites and webpages he has visited. A pilot study performed by Mozilla [33] found that users mostly switch into private browsing mode during the lunch break, presumably because users do not want their employers to know what they were looking at during their lunch break.

However, prior studies have shown that an adversary can easily compromise such a privacy goal. For example, Aggarwal et al. [1] demonstrated that browser extensions and add-ons can easily undermine the security of private browsing. Their study also resulted in a change in Google Chrome which disables all extensions while a user remains in private browsing mode.

In this paper, we examine the security of private browsing in Google Chrome and Mozilla Firefox. In particular, we develop an automatic tool to identify the browser traces left behind by a private browsing session. We find that Chrome and Firefox browser do not correctly clear browser traces left by some browser components when they exit a private browsing session. We demonstrate that the failure of handling these components completely undermine the goals of of private browsing. For example, as we will show in §3, an adversary can infer the websites that a user has visited by accessing the OCSP (online certificate status protocol) cache left behind by a private browsing session.

A instinctive reaction to the above finding would be to make the aforementioned components aware of the private browsing mode, in particular, by preventing them from writing anything to disk while in this mode. However, such a *privacy-aware* development approach relies on the developers to carefully insert condition checks on the current browsing mode and implement the correct logic for each mode. Given the complexities of browsers (and its extensions and add-ons), developers are prone to make mistakes and as a result, some browser components would remain *privacy-unaware*, as shown in several recent reports [8, 37].

In order to overcome the limitation of the piracy-aware development approach, we design and implement UCOGNITO, a universal framework for private browsing mode. With UCOGNITO, inadvertent disclosure of private information can be avoided in spite of privacy-unaware implementation by browser developers. UCOGNITO achieves this by overlaying the actual filesystem with a sandbox filesystem and restricting all the modifications made by a web browser to the sandbox filesystem when the browser is running in private browsing mode. Once the browser exists its private browsing mode, UCOGNITO discards the sandbox filesystem and no persistent modification is stored. Given that different browsers define private browsing differently, UCOGNITO also provides a set of privacy policies, thus allowing each browser to implement its private browsing mode according to its own definition.

UCOGNITO does not require any change to browser implementation. As a result, any browsers or browser-support applications (e.g., Chrome apps) can be run in private browsing mode no matter if the browsers or applications support private browsing mode. Since UCOGNITO implements the layered sandbox filesystem by interposing system calls, it does not require any modification to OS kernel.

In summary, this paper makes the following contributions.

- We propose an automatic tool, called UVERIFIER, that examines the security of private browsing and identifies previously unknown privacy violations in major browsers.

- We design UCOGNITO and demonstrate how it helps to implement private browsing mode for both Chrome and Firefox and how users can customize the system to suit their privacy needs.

- We implement UCOGNITO and evaluate its functionality and performance on Linux. We show that UCOGNITO is effective in preventing privacy violations with negligible overheads.

The rest of the paper is organized as follows. §2 discusses the threat model and the security goals of private browsing. §3 examines the security of private browsing and identifies privacy unaware implementation in web browsers. §4 presents the design of UCOGNITO followed by its implementation and evaluation in §5 and §6, respectively. §7 and 8 discuss the extensibility of UCOGNITO and related work. Finally, we conclude our work in §9.

2. BACKGROUND: PRIVATE BROWSING

In this section, we begin with the threat model and security goals of private browsing. Then, we discuss how web browsers achieve these security goals from the perspective of their implementations. In particular, we review the implementation of private browsing in two open source browsers: Chrome and Firefox.

2.1 Private Browsing Mode

According to [11], private mode is commonly perceived by normal users as a mode that could *provide additional privacy protection on users' browsing activities compared with public mode*. In practice, however, it is up to the browser vendor to define what these additional privacy protection are and some of them may not even align with an individual user's expectation or privacy needs.

Table 1 shows the differences in the interpretation of private mode by five mainstream browsers. Most notably, Chrome provides two implementations of private mode, termed Incognito Mode and Guest Mode, respectively, and states that the Guest Mode provides stronger privacy protection than the Incognito Mode in the sense that accesses to persistent data stored in existing user profile, such as browsing history, autofill etc, are blocked [12]. All other browsers we surveyed do not provide such a Guest Mode. There are other differences. For example, Safari allows persisting per-site permission (such as using Notification API [34]) learned in private mode in favor of usability while other browsers disallows such operation in favor of privacy.

We also observed conflicts between users' privacy needs and browser vendors' decisions in terms of what traces should be stored and used in private mode. For example, in one Firefox bug report in 2009 [23], the user is having concerns that SSL client certificate obtained in private mode should not be persisted while Firefox until now is still persisting such data. In a recent survey on private browsing [11], some participants even indicate that they would like to keep previously acquired cookies in private mode to enable auto-login in most websites, trading privacy for usability. In current private mode implementation, there is no way to resolve these conflicts because developers' decisions are in fact hard coded in the browser and the only option left to users is to either manually clean this trace (which is non-trivial) or accept developers' decisions. Our goal is to not only respect each browser vendor but also give freedom and control back to users for their pleasure.

2.2 Privacy Goals

Given the heterogeneity in the interpretation of private browsing mode, it is not possible to define private browsing mode in terms of a set of specific data allowed to be stored or used. In this paper, we respect this heterogeneity and instead define the high-level properties of private browsing. More specifically, we say that the implementation of a private browsing mode achieves the intended privacy protection if both goals are satisfied:

- **Stealthiness**: any data in private mode should not be stored unless explicitly communicated to and agreed by the user. If this goal fails, knowing of such persistent data would increase the probability of recovering users' online activities in private browsing session.

- **Freshness**: any persistent data obtained from previous browsing sessions should not be used in private mode unless explicitly communicated to and agreed upon by the user. If this goal fails, knowing of such trace would increase the probability of recovering users' online activities in previous browsing sessions.

Threat Model. We assume a same computer host-guest threat model which both can be malicious: (1) A guest launches private mode to prevent the host from inferring his browsing activity. The host is assumed to have full control of filesystem **after** the guest browsing session. (2) A host restricts a guest in private mode to prevent his browsing activity from being inferred by the guest. A guest is allowed to perform any browsing permitted by browser **during** the private browsing session.

2.3 Complexity of Implementation

A naive implementation of private mode would be to disable all features that could persistently store data in filesystem, such as cookies, HTML5 local storage etc. However, such an approach would allow the website to easily detect whether the user is in private mode by testing whether such feature is accessible. In fact, for features that persist data in public mode, browsers tend to implement a similar but non-persistent version to give website the impression that such features is available and hence hide any visible effect of browsing in private mode. Such "mimicking" approach is to satisfy the indistinguishability requirement which states that the website should not be able to distinguish which mode a user visits in. However, it also makes the private mode implementation inherently complex and leads to issues that defeat the two privacy goals.

Indeed, when we reviewed the source code of Chrome 37.0.1 and Firefox 42.0.2331, we found that both browsers follow the "mimicking" approach. Moreover, the way the "mimicking" is implemented is ad-hoc with heavy use of if-else branches and polymorphism to achieve logic separation between public mode and private mode, which exponentially increase code complexity.

Category	**Persistent data**	**Use**						**Store**					
		Firefox	**Chrome**		**Opera**	**Safari**	**IE**	**Firefox**	**Chrome**		**Opera**	**Safari**	**IE**
			Incognito	Guest					Incognito	Guest			
Transparent to user	Browsing history	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗
	Cookies	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Cache	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	HTML5 local storage	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Flash storage	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
User action involved	Download entries	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗
	Autofills	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗
	Bookmarks	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓
	Per-site zoom level	✓	✓	✗	✓	-	-	✗	✗	✗	✗	-	-
	Per-site permission	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗
	SSL self-signed cert	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗
	SSL client cert	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-
Add-on support	Add-on storage	✓	✓	-	✓	✓	✓	✓	✓	-	✓	✓	✓
	Add-on enabled by default	✓	✗	✗	✗	✓	✓	-	-	-	-	-	-

Table 1: A comparison of policies for private browsing modes in five major browsers in their default settings. **Use** means accessing persistent data generated in previous browsing sessions. **Store** means storing persistent data generated in private browsing session. Interestingly, each browser has a different set of privacy policies. For example, private browsing in Firefox enables its plugins but incognito mode in Chrome disables the use of add-ons.

```
1  // @netwerk/cookie/nsCookieService.cpp
2  DBState *mDBState;
3  nsRefPtr<DBState> mDefaultDBState; // DB for normal mode
4  nsRefPtr<DBState> mPrivateDBState; // DB for private mode
5
6  // invoked when initializing session
7  void nsCookieService::InitDBStates() {
8    ...
9    mDefaultDBState = new DBState(); // DB for normal mode
10   mPrivateDBState = new DBState(); // DB for private mode
11
12   // default: normal mode
13   mDBState = mDefaultDBState;
14   ...
15 }
16
17 // invoked when storing cookies
18 void nsCookieService::SetCookieStringInternal() {
19   ...
20   // decide which cookie DB to use, depending on the mode
21   mDBState = aIsPrivate ? mPrivateDBState : mDefaultDBState;
22   ...
23 }
```

Figure 1: Firefox maintains user's objects for each mode, public and private mode. For example, Firefox's cookie service has two database instances, namely `mDefaultDBState` and `mPrivateDBState`. This implementation practice makes the code base complex: for example, developers have to distinguish which user's mode is activated whenever it updates a cookie value (e.g., `SetCookieStringInternal()`.)

For example, as illustrated in Figure 1, Firefox maintains a global variable `aIsPrivate` that identifies if a browsing session is in private mode or not. The variable is initialized while a browsing session starts. Firefox checks the value of `aIsPrivate` whenever it needs to access or modify browser cookie. Firefox defines a set of different processes for private browsing and encapsulates them in `mPrivateDBState`. As is shown in line 28 in Figure 1, Firefox uses `aIsPrivate` to determine if executing the logic encapsulated in `mPrivateDBState` when cookie storage or access is needed. The situation is even worse for Chrome as it has two private mode implementations: Incognito mode and Guest mode.

The code snippet in Figure 1 is just an example of how Chrome and Firefox add complexity to their browser implementation in order to support private browsing. In addition to cookie management, the full implementation of private browsing in Chrome and Firefox also makes many other components aware of the private browsing mode, such as cache service, form auto-complete, history, SSL

```
// 1. Detecting private browsing mode @MDN
Components.utils.import(
    "resource://gre/modules/PrivateBrowsingUtils.jsm");
if (!PrivateBrowsingUtils.isWindowPrivate(window)) {
  ...
}

// 2. Detecting mode changes @MDN
function pbObserver() { /* clear private data */ }
var os = Components.classes["@mozilla.org/observer-service;1"]
          .getService(Components.interfaces.nsIObserverService);
os.addObserver(pbObserver, "last-pb-context-exited", false);
```

Figure 2: Each addon has to take a special care on private mode: not only by checking the current browsing mode, but also listening to the context changes. This amount of complexity results in many privacy issues in popular addon (see. §6.2)

certificate store etc. In short, it is a major undertaking to make changes throughout the browser code base in order to make the browser *privacy-aware*.

Given the huge code base of browser and the richness of user data, it is unlikely that developers can correctly manage the exponential growth of complexity. As shown in Table 2, this implementation strategy increases the chances of bugs and privacy unawareness, and makes browsers cumbersome and hard to respond to users' privacy needs. We systematically evaluate the privacy unawareness issue caused by such implementation practice in §3.

2.4 Caveat Interface for Add-on

Although browser add-ons have been an indispensable part of modern browsers, the automatic enabling of private browsing mode to add-ons is not supported at all. Instead, similar to the "mimicking" approaches used in instrumenting browser engine, Firefox and Chrome only provide add-ons an interface to check the mode of current browsing session and expect add-on developers to implement different logic for different modes (as shown in Figure 2).

As add-ons running in private mode might cause unintended privacy violations, Firefox takes the approach of manual app review [35], which states that each add-on must pass at least a preliminary review to be listed and a full review to be ranked. Since browser add-ons are still small in code size, such review practices are still effective in regulating add-on developers to respect the private-browsing policy. However, with increasingly complicated add-ons being developed, manual review might soon reach to its

Category	Bug ID	Browser	Description
Privacy unawareness	967812 37238	Firefox Chrome	Permissions Manager writes to disk in Private Browsing Mode Cookie exception recorded while in private browsing mode
Implementation bug	553773 21974	Firefox Chrome	Entering private browsing aborts active downloads Private Browsing download window shows wrong data
Reluctance in responding to user needs	1074150 471597	Firefox Chrome	Second instance of incognito mode remembers the log-in session Sessions are not "private" when open two or more private (incognito) windows

Table 2: Bug report samples related to private mode implementation in Firefox and Chrome. It is interesting to see that both Firefox and Chrome developers introduced similar bugs, indicating they all have no effective ways to control the implementation complexity.

limitation.

Chrome takes the "use-it-at-your-own-risk" approach. When a user wants to enable an add-on in incognito mode, a message (see Figure 3) will popup and alert user on the risks of their decision. Such an approach essentially alleviate Chrome from the task of ensuring that add-ons respect incognito mode.

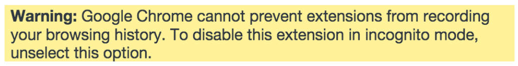
Warning: Google Chrome cannot prevent extensions from recording your browsing history. To disable this extension in incognito mode, unselect this option.

Figure 3: Alert displayed when enabling incognito mode for an extension.

3. TESTING PRIVATE BROWSING

Knowing the security goals of private browsing and their implementation in mainstream browsers, we developed a technical approach to examine the security of private browsing in Firefox and Chrome. In this section, we describe our approach and summarize the privacy-unaware implementation that our approach identifies.

3.1 Privacy-Unaware Implementation

As we describe in §2.3, developers take the "mimicking" approach in extending the browsing engine to support private mode. Initially, we expected browser vendors to employ a systematic approach to identify the features/components that persist data during a browsing session, i.e., those in need of "mimicking". Unfortunately, we are unable to find any hint that Firefox or Chrome follows such an approach. On the contrast, as evidenced in their design documents [2, 3], most of these features are identified based on developers' judgment.

Such an approach might be feasible with a small code base and in a slow development cycle. However, in the case of browser, with the constant introduction of new techniques and standards, web applications are given increasing power in interaction with client machines, among which many of them introduce new opportunities to persist data on-disk, such as HTML 5 local storage [15], and Geolocation API [29]. Developers might not be fully aware of such potential privacy leakage when instrumenting browser engine for those functionalities and hence, either causing users' browsing activities in private mode be persisted to disk (violation of stealthiness goal) or causing previous browsing data be carried over to private session (violation of freshness goal). Therefore, it is not uncommon to see the pattern shown in Table 3 in the browser's release cycle: a new web standard is mis-implemented by developers, causing a privacy leakage bug which takes years to be patched.

3.2 Uverifier: Privacy Violation Detector

Motivated by the long cycle to patch a privacy violation case (as shown in Table 3) as well as the concern that there might be more privacy-unaware implementations, we develop UVERIFIER to systematically check for violations of stealthiness and freshness goals defined in §2.2 in Chrome and Firefox.

Date	Event	Note
Dec 2008	New standard proposed	Geolocation API
May 2010	Implementation and public release	Chrome 5.0
Aug 2010	Violation of incognito mode reported	Issue id 51204
Apr 2013	Issue patched	Revision id 192540

Table 3: The initial integration of a new web standard, Geolocation API, caused permission settings saved in incognito session to be persisted and have an effect in public session. It was reported after 3 months of official release and was not patched for 3 years.

UVERIFIER consists of three major components:

- a script-based driver that drives a browsing session, including creation of new profiles, starting/stopping browser, visiting suspicious website and initializing plug-ins.
- a system call tracer, based on `strace`, that captures browsers behavior in terms of system calls.
- an analyzer that extracts privacy violation patterns from the system call traces.

We start each test by creating a fresh browser profile and:

- To test stealthiness goal, we run one private session (A) only.
- To test freshness goal, we run one public session (A) and one private session (B) consecutively.

For each browsing session, the web driver starts the browser in the desired mode, performs browsing activities and finally shuts down the browser. The logger captures all accesses and modifications to the underlying filesystem by recording all file-related system calls. Unlike previous study [1] that only focused on changes in the browser profile directory, we capture any changes made to the filesystem.

We define any file that 1) is opened/created with write flag, 2) has data inflow and 3) is not deleted after browsing session A as a trace stored, denoted by t_s. Any file that 1) is opened with read flag and 2) has data outflow in browsing session B is considered a trace used, denoted by t_u. A potential violation of the stealthiness goal is detected if any t_s is found while a potential violation of the freshness goal is detected if any $t_s = t_u$ is found. We then manually analyze these potential violation cases and filter out those cases that cannot be used to recover users' browsing activities, such as changes of timestamps values.

3.3 Privacy Violations

Using UVERIFIER, we discover several previously unknown privacy violations in Chrome and Firefox and we have reported those issues to Mozilla and Google accordingly. Here, we summarize the components that the browser implementation is not made aware of private browsing mode, and highlight how they compromise the security goals of private browsing.

OCSP cache. The Online Certificate Status Protocol (OCSP) is an Internet protocol used for obtaining the revocation status of a certificate. Today, OCSP has been supported by many web browsers. Using this protocol, web browsers can ensure that their users connect to the domains that they intend to. In the Firefox case, upon receipt of a certificate, Firefox sends an OCSP request with the certificate serial number to the issuing Certificate Authority (CA) to query certificate validity and by default, caches the OCSP response received at `/<firefox-profile>/cache2/entries/<hash>` even in private mode. This violates the stealthiness goal of private mode because an adversary could infer the websites that a user has visited in private browsing mode with various information in cache, including the certificate serial number which can be uniquely mapped to a web domain.

PNaCl translation cache. Native Client is a sandbox for running compiled C and C++ code in Chrome browser efficiently and securely. PNaCl is a portable version of Native Client. It allows developers to compile their code once to run in any website with ahead-of-time (AOT) translation. As is described in [36], PNaCl speeds up the loading time of PNaCl applications by caching the translation of portable bitcode files at `/<chrome-user-dir>/PnaclTranslationCache`. However, these caches are carried over to the two private modes implemented: Incognito mode and even Guest mode, hence providing potential mechanisms for web tracking. For example, the developer can embed a PNaCl application on his site, which sends a message to the site when it starts to run on a user's browser. The developer could measure the interval between the time of HTTP request and the time of receiving the message from the PNaCl application. If a relatively short interval is observed, it is highly likely that the user is a returning visitor even though he visits the site in private browsing mode. As an illustration on the feasibility of such inference attack, the loading time for PNaCl app at `http://gonativeclient.appspot.com/demo/lua` is about 3 seconds without cache and 0.1 seconds with cache.

Nvidia's OpenGL cache. Similar to the PNaCl translation cache, Nvidia's OpenGL shader disk cache allows compiled shaders to be cached to the system disk so that they do not need to be recompiled again later on, which can potentially save time by just pulling these binaries from the disk instead. On Linux system, the cache is usually stored at `/<user-home>/.nv/GLCache/`. Such caching is on regardless of which mode the browser is running on. Therefore, by probing the content in their cache, it is possible to infer which website a user has visited, especially with websites that contains rich WebGL contents. We show a simple inference attack based on WebGL cache size only given the fact that the cache size is highly correlated to the richness of WebGL contents. For example, visiting a lightweight WebGL demo[1] yield a 52,911 bytes increase in cache size while a heavyweight WebGL demo[2] yield a 176,777 bytes increase in cache size. Hence, by mere measuring the increase in cache size, we could infer whether the webpage user visited is rich in WebGL contents. Such inference is reasonable as websites with rich WebGL contents tend to be more gaming or video oriented, implying that the user might played a web game in the browsing session.

3.4 Unit and Regression Testing

UVERIFIER, like other dynamic tools, requires external triggering of privacy-related features to be tested. Therefore, it makes perfect sense for browser developers or add-on developers to integrate UVERIFIER into their unit and regression testing procedures to maximize code coverage by UVERIFIER and eliminate unintentional mistakes that lead to privation violations.

For cautious users whose browsing habits are fairly stable, they can employ UVERIFIER to check against privacy violation cases in their normal browsing and add-on usage pattern.

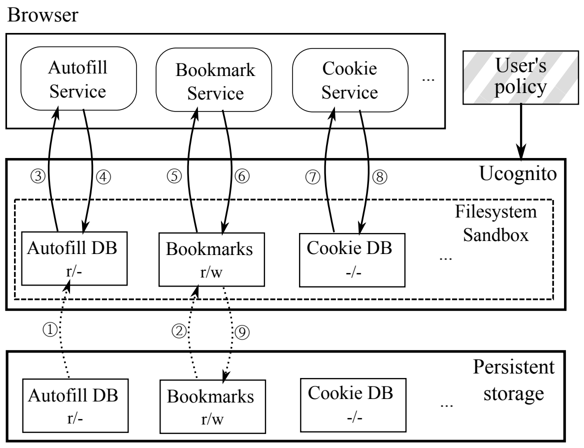

Figure 4: UCOGNITO architecture and its UCOGNITO interaction with browser and persistence layer. Upon initialization of UCOGNITO, data in persistence layer is brought to the filesystem sandbox based on the policy specified. In this case, a `COPY` policy is attached to both autofill DB and bookmarks to allow them being transferred to the filesystem sandbox ①, ② while transferring of cookie DB is prohibited by a `CLEAN` policy, therefore, an empty cookie DB is created. During the private browsing session, browser services will only interact with the data in filesystem sandbox (③-⑧), which gives the browser an impression that it is interaction with the underlying persistence layer. Upon finishing of the browsing session, data is written back to the persistence layer (⑨) if it is attached with the `WRITE` policy.

4. UCOGNITO: NEW FRAMEWORK FOR PRIVATE BROWSING

Designing and implementing private mode can be non-trivial as illustrated in §2 and §3. Motivated by this, we propose UCOGNITO that radically changes the view of private mode in browser and the way how it should be implemented.

4.1 Overview

In UCOGNITO, we decouple private mode implementation from browser code base, i.e., we do not consider private mode as a browser-specific feature which, implementation-wise, is indispensable from the browser code base. Instead, we consider it as an customizable overlay that will automatically provide the intended privacy goals of private mode when put on top of any browser implementation.

To achieve this goal, UCOGNITO abstracts out the notion of private mode from browsers and contain all the logic related to private mode within itself. In other words, at runtime, the browser is transparent to which mode it is running in and will always uses the exact same code and logic for both private mode and public mode. In this way, private mode effectively becomes a concrete and separate module instead of pieces of logics scattered across all features that might persist data and UCOGNITO serves as one central point for testing and configuring private mode.

4.2 Architecture

Filesystem sandboxing. As shown in Figure 4, UCOGNITO em-

[1]available at `http://threejs.org/examples/#webgl_animation_cloth`

[2]available at `https://developer.mozilla.org/en-US/demos/detail/the-polar-sea/launch`

ploys filesystem sandboxing to give browsers an impression that it is interacting with the actual filesystem in the persistence layer while in fact, the browser only interacts with the sandboxed filesystem. Such a design creates an isolated filesystem environment for the private browsing session and hence, all traces generated in the session is collected and retained in the sandbox.

Policy system. Policy system kicks in at two points,

- Upon starting the private browsing session, only persistent data that are explicitly given the "read" access is copied into the filesystem sandbox.
- Upon finishing the private browsing session, traces collected in the sandboxed filesystem is conditionally written back to the filesystem in persistence layer if such data is given the "write" access. For the traces that are not written back, they are discarded after the session finishes as the whole filesystem sandbox only lives in memory, which is freed after the private browsing session.

4.3 Challenges and Solutions

The design of UCOGNITO involves two challenges. First, it needs to respect the heterogeneity in the interpretation of private mode by different browser vendors and users, as described in §2.1 Second, it has to address both privacy goals defined in §2.2 and be indistinguishable to websites §2.3.

In accommodating heterogeneity in the interpretation of private mode, UCOGNITO employs a policy system to allow for flexible configuration on the storing or using of each type of persistent data. In the view of the policy system, each file in the filesystem is bundled with two policies:

- Allow/deny write access, corresponding to whether the persistent data is allowed to be stored in the file during private browsing session.
- Allow/deny read access, corresponding to whether persistent data contained in this file can be used during private browsing session.

Hence, by toggling the policies bundled with each file, both users and browser vendors can express their interpretation of private mode.

In guaranteeing the stealthiness and freshness goal, we design the policy system to take the "whitelist" approach. To be specific, the following logic is wired into the policy system:

- Strongest protection by default: without any policy specified, complete isolation, i.e. deny all read/write accesses, must be assumed.
- Conservativeness: any type of persistent data that is not explicitly agreed or expected by the user should not be left or shared. In other words, if storing or using of persistent data is intended, it must be mentioned as a policy.

This ensures that users have full knowledge and control on the types of persistent data stored and used in private mode. Therefore, by definition, both stealthiness and freshness goals are achieved

UCOGNITO also increases the indistinguishability between private mode and public mode by eliminating attacks that rely on the differences in browser implementation of public mode and private mode. For example, the hyperlink attack demonstrated in [1] relies on the fact that most browsers do not render visited link in purple in private mode. With UCOGNITO, such attack will effectively be mitigated.

4.4 Sandboxing Layer

To isolate the trace of a private mode from the system, our general approach is to apply a filesystem sandbox on the commercially off-the-shelf web browsers. We assume that the filesystem is the only persistent storage on the system. In other words, we ignore all in-memory components as it has already been isolated under the process isolation mechanism on the system.

Isolation of the filesystem can be done by rewriting the path to the contained location, in particular, by intercepting system calls on its entrance. When the browser accesses a file with the `open` system call, our sandbox hooks its entrance and changes the "path" argument to the sandbox directory. For example, if the path is to `/home/user/.config/*`, we redirect this to `/tmp/ucognito-PID/home/user/.config/*`. This redirection was originally introduced in MBox [17].

However, MBox only supports the redirection of the filesystem to the ephemeral location (e.g. `/tmp/*`), and it lacks the policy to determine which file to read or not to read, and only has manual options to discard or commit the changes to the host filesystem from the sandbox. In UCOGNITO, we need to support more options such as choosing files to be included or to be cleaned, and then writing back the configurations to the original profile. Accordingly, we add features to MBox to support selective writing on all changes in the filesystem.

Details of the implementation of our sandbox is described in section §5.1.

4.5 Policy System

To satisfy the flexibility requirement, i.e. configurable private mode, in UCOGNITO, we provides a policy system over sandbox isolation. Our policy have three types:

- `CLEAN`: create an empty file to prevent file-copying from the original user profile to the private profile;
- `COPY`: copy a file or sub-directory to the private profile from existing user profile, to employ existing settings;
- `WRITE`: allow data to be written back to the user profile after the session closes.

We use the `CLEAN` policy to support running a browser or an extension at its pristine stage (i.e. first-time execution of the browser). If a path is specified as `CLEAN`, we create an empty directory and files for its sub-directory elements then redirect all further access. Next, `COPY` is for applying existing session information to the sandbox. Either a path from the original profile or a path from a different location can be chosen, to load the clean a profile from the file.

For the `WRITE` policy, we memorize the original location where it is read (i.e. from `COPY`), then write back the trace after the browser instance terminates.

These policies are defined to determine which files is loaded into the profile, and what data is committed back to the profile. `COPY` and `CLEAN` are for "read" property, as it determines how to load a profile into the session. `WRITE` is for "write" property, as it specifies how to deal with trace data created by the session.

Enforcement of these policies can be done as follows. For `CLEAN` files, we create files on the sandbox location; for `COPY`, we copy the file from the location to the sandbox path. After creation of the files, we restrict the browser instance's access (using further `open`, `read`, `write`, etc.) to the file within the sandbox container. The `WRITE` policy is enforced at the end of execution of the web browser session. When exiting the sandbox, all files specified with `WRITE` are written back to the original location of the filesystem with the contents of the files in the sandbox filesystem.

The policy can be defined as INI-style file (a `.cfg` file); each policy is divided into sections (e.g. `[copy] and [clean]`) with the listing of files or directory entries (if ends with "/")

Example: Chrome's Private Browsing. We present examples of our policy definition in for the incognito mode and guest mode of Google Chrome in Figure 5 and Figure 6, respectively.

For the private mode of Google Chrome, it is allowed to read: 1) browsing history, 2) autofills, 3) download entries, 4) per-site preferences, 5) custom SSL certificates, 6) bookmarks, and 7) extension storage. To support each of these, we list the files that belongs to each access at the `COPY` sections. Then, we mark the directory that Google Chrome stores user profile for the normal instance in a `CLEAN` section to prevent accesses to the files that are not specified in the `COPY` sections. The private mode of Google Chrome stores bookmarks and extension storage that have been changed during the private mode execution. Thus, we specify files and directories reserved for those data at the `WRITE` section to store the changes when the session terminates.

The policy definition for guest mode of Google Chrome is quite simple. Since it does not need to access anything from the user's profile except for the certificate store `cert9.db`, we just mark the home directory for the user as `CLEAN` and add both `COPY` and `WRITE` `cert9.db`. These examples show that our policy is flexible enough to easily support existing implementation of private modes of different web browsers.

With these two policy files, it is clear to see that the guest mode provides much stronger browsing privacy than the Incognito Mode. It also shows the easiness of customization, i.e., if a user is not satisfied with the policy for incognito mode, he/she could easily modify it to meet his/her own privacy needs. Implementation details of policy are described in §5.2.

5. IMPLEMENTATION

We implemented the sandbox, the policy, and additional UI layer (in §5.3) on the Linux operating system, for the distribution Ubuntu 14.04 LTS that runs Linux Kernel 3.13.0. Since we utilize Computing/Berkeley Packet Filter (`seccomp-bpf` [9]) for system call hooking mechanism, we need the kernel version over 3.5.

Figure 7 shows that the complexity of the implementation of UCOGNITO in terms of lines of code. We only wrote 564 lines of code to support all three layers, which we describes in the following subsections.

5.1 Sandboxing Layer

We implemented the sandboxing layer using the same mechanism in a MBox filesystem sandbox [17]. Basically, the sandbox hooks all file-related system calls, then rewrites the path argument to isolate a file at certain path into an ephemeral location, i.e. under `/tmp`.

System call hooking. System call hooking is done with *seccomp-bpf*, which provides an easy and fast way of intercepting system call entry/exit. We placed hooks on 50 syscalls that deal with file path. For example, `open`, `creat`, `unlink`, `stat`, `mkdir`, `rmdir`, `symlink`, `readlink`, etc. are the system calls that operate on an argument that specifies a location in the filesystem.

In addition to the file-related system calls, we additionally placed a hook on the `bind` syscall. The reason is that when `bind` is called with the `AF_UNIX` argument (to create a Unix Socket), the port that the socket is bounded to is not a network location; instead, it is a file path (e.g. `/tmp/.com.google.chrome.*/SingletonSocket`). Therefore, we place a hook on `bind`, and on its entrance, check if it is called for `AF_UNIX`. Then, we re-write the path if it is required to be contained in the sandbox.

```
# copy section: copying files from the user profiles
[copy]
# Use: browsing history
~/.config/google-chrome/Default/History
~/.config/google-chrome/Default/History-journal
~/.config/google-chrome/Default/Visited Links
~/.config/google-chrome/Default/Favicons
~/.config/google-chrome/Default/Favicons-journal
~/.config/google-chrome/Default/Top Sites
~/.config/google-chrome/Default/Top Sites-journal

# Use: autofill data
~/.config/google-chrome/Default/Login Data
~/.config/google-chrome/Default/Login Data-journal
~/.config/google-chrome/Default/Web Data
~/.config/google-chrome/Default/Web Data-journal

# Use: per-site preferences
~/.config/google-chrome/Default/Preferences
~/.config/google-chrome/Default/Secure Preferences

# Use: SSL certificates
~/.config/google-chrome/Default/TransportSecurity
~/.config/google-chrome/Default/Origin Bound Certs
~/.config/google-chrome/Default/Origin Bound Certs-journal

# Use: SSL client certificates
~/.pki/nssdb/cert9.db

# Use: bookmarks
~/.config/google-chrome/Default/Bookmarks

# copy section: include all subdirectory
[copy]
# Use: extension storage
~/.config/google-chrome/Default/Local Extension Settings/

# clean section: exclude files & sub-directories
[clean]
# exclude all other files in the home directory
~/

# write section: write-back data to the user profile
[write]
# write-back bookmarks
~/.config/google-chrome/Default/Bookmarks
# write-back client certificates
~/.pki/nssdb/cert9.db
# write-back extension storages
~/.config/google-chrome/Default/Local Extension Settings/
```

Figure 5: Policy configuration file for the private mode of Google Chrome.

```
# exclude all files in home directory
[clean]
~/

# Use: SSL client certificates
[copy]
~/,pki/nssdb/cert9.db

# write-back client certificates
[write]
~/,pki/nssdb/cert9.db
```

Figure 6: Policy configuration file for the guest mode of Google Chrome.

Note that we do not place any hook for `read`, `write`, `send`, or `recv`, which are very frequently called in networked applications like web browser. Since we rewrite the "path" argument only at the entrance of the system call, the interception happens very rare; the overhead of hooking is very low. Please refer to §6 for the performance overhead of the sandbox.

Containing file access. To contain file accesses, we rewrite the path argument on each system call entrance. For example, on `open` system call, if the first argument (path) is on `/home/user/.config`, then we overwrite the path into the contained location (e.g. `/tmp`)

Component	Lines of code
Sandbox & Policy	415 lines of ANSI-C
UI Layer	149 lines of Java and Python
UCOGNITO	564 lines of code
MBox [17]	24,311 lines of ANSI-C
MBox + UCOGNITO	24,827 lines of code
UVERIFIER	533 lines of Python

Figure 7: Components of UCOGNITO and an estimate of their complexities in terms of lines of code. UCOGNITO has very few (564) lines. Note that we deleted/commented-out unused part of MBox (48 lines) so that total number of lines are not matched with addition of the two numbers. We also note that our verifier, UVERIFIER has 533 lines of Python code.

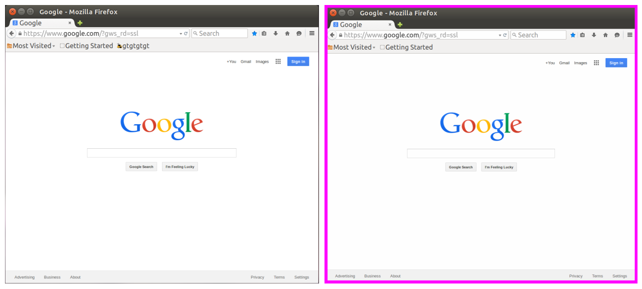

Figure 8: UI highlights for UCOGNITO. Figure on the left shows normal running of Mozilla Firefox; on the right, our UI highlight is shown as pink border line to indicate UCOGNITO is working for the web browser.

by adding a prefix to the path, that is, we rewrite the path to `/tmp/ucognito-pid` `/home/user/.config`. Since the rewriting happens before entering the kernel execution, the file descriptors opened for the redirected path. Thus, further read/write (accessed with file descriptor) automatically happens on the file at the contained location. Note that this rewriting operation is done outside of the sandbox. Since we placed hooks for all file-related system-calls, the web browser instance running in the sandbox environment cannot bypass this re-writing routine.

5.2 Policy System

We implemented our policy system designed in §4.5 on our filesystem sandbox. For each policy section (`CLEAN`, `COPY`, or `WRITE`), they can have two types of path entries: a file or a directory entry. Note that the enforcement is handled differently for files and directories (including all sub-directory entries). We describe the rules on accessing files and directories under our policy system as follows.

File: initialize before start. For the file entry, we initialize them during initialization of the sandbox; i.e., before running the browser application. For files marked as `COPY`, we duplicate the specified file from the host filesystem to the sandbox filesystem. For files marked as `CLEAN`, we simply create an empty file on the sandbox filesystem. After the file is initialized in this way, all further access (both read/write) to those files are contained (by path rewriting) in the sandbox filesystem.

Directory: initialize upon access. For directory entries, we initialize them in lazy way, i.e., only when a system call is issued to access a path under the specified directory. Upon the interception of such system call, we get the path argument and check if the path is under the directory listed in a `COPY` or `CLEAN` section of the policy file. Note that if the path does not belongs to any of directory listed in the policy, we copy the file in the sandbox filesystem. This is to support loading of libraries (e.g. `*.so` files) and other files that is required but does not related to the private mode of the web browser. To prevent this automatic `COPY` rule being applied to the whole filesystem, policy writer must specify the path, which is usually the user's home directory, in the `CLEAN` list to opt-out from this rule. In case of a path is matched with the entry in both `CLEAN` and `COPY` (e.g. `~/.config` is in `COPY` while `~/` is in `CLEAN`), we set `COPY` overrides `CLEAN` to follow the policy of *white-listing* of the read access.

Data write-back. For `WRITE` policy, we enforce the policy when the protected application (in our case, the browser) terminates. Upon exiting the application, we get the list of *modified* files in the sandbox filesystem. Modification is detected by comparing hash of contained file with that of the file in the host filesystem. Then, we check if a path belongs to any of policy definition. For a file entry, if the path is matched, then it is written back to the source file at the host filesystem. Likewise, for the directory entry, if prefix matches, we perform the same operation.

5.3 UI Layer

In UCOGNITO, since it wraps the normal web browser with sandbox to make it as private mode, there is no UI indicator like Mozilla Firefox and Google Chrome. To the user, it is very important to know the mode of the current session. We employ a very simple method: highlighting the border of the window with different colors. Figure 8 shows an example. We keep track of the focus on the X11 window display. Whenever focus changes, we check if the currently focused window is in private mode. If it is, we retrieve the size of the window, then draw a colored, thick borderline rectangle over the window. We implemented the drawing part with JPanel and make the rectangle *always-on-the-top* so that it cannot be hidden by the browser window.

5.4 Launching a Private Mode

Launching a web browser with UCOGNITO is quite simple. The following command launches Google Chrome web browser with the specified policy file:

```
$ ucognito -P chrome_incognito.cfg -- google-chrome
```

The option argument `-P` indicates the location of policy file described above. At the time of launching, before executing the actual web browser program, UCOGNITO applies policy first to create a new file (for `CLEAN` policy) or copy a file (for `COPY`). Afterwards, UCOGNITO applies the sandbox by calling *seccomp* syscall and then executes browser instance. The whole procedure for private browsing under UCOGNITO can be divided into four phases:

1. Initialization phase. Load the policy file (if specified), then applies pre-execution policies (e.g. `COPY` or `CLEAN` file entries);
2. Browser starting phase. Create a new process, place hooks on 50 file-related system calls using `seccomp-bpf`;
3. Browsing phase. Launch the browser to let user browse in private mode;
4. Cleaning phase. Depending on the `WRITE` policy, check if any file belongs to the policy has been changed. If any, then write back the file to the host filesystem to make it persist.

6. EVALUATION

The goal of our evaluation of UCOGNITO is to answer:

1. What are the uses cases for UCOGNITO? (§6.1, §6.2)

2. How flexible and general is UCOGNITO's policy in implementing private browsing schemes of popular browsers? (§6.3)

3. How much performance overheads does UCOGNITO incur? (§6.4)

Experimental setup. To evaluate UCOGNITO, we ran Mozilla Firefox 37.0.2 and Google Chrome 42.0.2311.152, on Ubuntu 14.04 LTS, running 64-bit Linux Kernel 3.19.0. We ran our experiments on commodity hardware, equipped with Intel Xeon E5-1620 (one CPU, quad core, and 3.6GHz) and 16GB of RAM.

6.1 Preventing Privacy Violations

We tested the effectiveness of UCOGNITO in protecting against the privacy violations we found in §3.3. Our results showed that UCOGNITO is able to mitigate all these privacy violation cases:

- OCSP cache. We ran a single private browsing session to visit a HTTPS site. We observed that OCSP cache files generated in the temporary filesystem during the private browsing session are subsequently cleaned after the session finishes. None of these cache files are written back to the original filesystem.

- PNaCl translation cache. We first ran a public browsing session to generate PNaCl translation cache first and subsequently ran a private browsing session to access the same PNaCl app[3]. Unlike in §3.3, we observed no discernible loading time differences between these two sessions.

- Nvidia's OpenGL cache. We first ran a public browsing session to generate Nvidia's OpenGL translation cache and subsequently ran a private browsing session to access the same WebGL app[4]. Unlike in §3.3, we observed no differences in the Nvidia cache file in original filesystem and found that new cache files are generated and subsequently deleted in the temporary filesystem.

6.2 Supporting Add-ons

Add-ons can be a major source of privacy violation in private mode, as evidenced in §2.4. We tested the effectiveness of UCOGNITO in protecting against the privacy violations brought by add-ons. We evaluated UCOGNITO against four popular Chrome extensions without modification to the policy defined in Figure 5. We chose these add-ons not only because of their popularity, but also because we expected them to cause privacy violations as their core functionality, i.e., session, history and autofill management, inevitably involve user private data. As shown in Table 4, UCOGNITO successfully prevented all privacy violations found in those four extensions, indicating that UCOGNITO is a promising candidate in enabling private mode support for add-ons without making any changes to those add-ons.

6.3 Policy Flexibility

We measured the flexibility of our policy system in terms of how fine-grain we can achieve in modelling a specific private mode. Since UCOGNITO only provides per-file granularity for policy specification, if two types of persistent data are contained in one single file and their use/store behavior is different in private mode, our policy system will not be able to model that. Otherwise, we can always define a policy to match the intended definition of private

[3]available at http://gonativeclient.appspot.com/demo/lua

[4]available at https://developer.mozilla.org/en-US/demos/detail/the-polar-sea/launch

Benchmark	Firefox		Chrome	
	Base	UCOGNITO	Base	UCOGNITO
Kraken (ms)	1171.1	1171.2 (0.0%)	1108.6	1115.2 (0.6%)
Sunspider (ms)	158.3	159.8 (0.9%)	173.1	177.4 (2.5%)
Octane (pts)	27164	27013 (-0.6%)	27266	27018 (-0.9%)

Table 6: Performance overheads of standard JavaScript benchmark on Firefox and Chrome running with UCOGNITO: the worst case performance overhead is around 2.5%

Website	Base	UCOGNITO	Overhead
Google.com	193 ms	196 ms	1.55%
Bing.com	190 ms	193 ms	1.58%
Twitter.com	599 ms	614 ms	2.50%
Facebook.com	256 ms	259 ms	1.18%

Table 7: Page loading time in Google Chrome, with and without UCOGNITO. The private-aware browsing with UCOGNITO incurs negligible overhead (<15 ms).

mode. We show the mapping of persistent data to its container in Table 5, and from the table it can be shown that for Firefox, browsing history, downloaded entries and bookmarks are all persisted in the the same file, `places.sqlite`. If we attach the `WRITE` policy to allow saving bookmarks in private mode, then the browsing history will be persisted too, which is not user intended. User or browser vendor is forced to make a trade-off decision on this (although we believe a better design is to have Firefox separate them). However, this is the only case in Table 1 that cannot be expressed as a policy in UCOGNITO. For any other item, enabling loading of it in private mode requires only one `COPY` policy while enabling storing of the item requires only one `WRITE` policy.

6.4 Browsing Performance

Since UCOGNITO heavily places hooks on the system calls, it affects the running time of web browser. To measure the performance overhead, we ran two types test: 1) JavaScript benchmark, and 2) measurement of page load time. While JavaScript benchmark shows the performance overhead in the computational core of the web browser, the page load time exhibits the actual performance of what user would experience because it measures the delay from the user action to the time when the page is ready to serve.

Benchmark setup is done as follows. For the baseline, we ran the browser instance with private mode from the command-line. For UCOGNITO, we ran with the command discussed in §5.4, enabling the policy that is equivalent to private mode of the baseline.

It is worth noting that since our sandbox uses `/tmp`, a kind of ramdisk, for its ephemeral storage, access on those files are much faster than running web browser alone. To avoid performance skew caused by this, for the baseline running, we mapped our home directory (i.e. `/home/user`) as a same ramdisk (mapped as `tmpfs`) during the evaluation[5].

For JavaScript, we ran three benchmark programs: Kraken [24] from Mozilla, Sunspider [38] from Webkit, and Octane [13] benchmark from Google. Table 6 shows the result of the benchmark. We ran the benchmark for 10 times and took the best value for each case. All results except one exhibits less than 1% of overhead and the worst case, Sunspider on Google Chrome, incurs only 2.5% of execution overhead. Since `seccomp-bpf` is very light weight, and we only place hooks on the system calls that initiate file access and

[5]Without this settings, UCOGNITO shows better performance than the baseline.

Add-on	# users	Violation trigger	Behavior in Chrome Incognito Mode	Behavior in UCOGNITO
Session Buddy	373409	Visit any website	Almost everything persisted, including history, caches, cookies, extension settings, etc.	Only changes in `Local Extension Settings` are written back to original filesystem as per policy, other changes are discarded
StayFocusd	600944	Start the timer on a website	Log file in `Sync Extension Settings` contains the website url	The log file is deleted in the temporary filesystem upon completion of the session
Better History	248112	Visit any website	Website URL saved in log file in `Extension State`	The log file is deleted in the temporary filesystem upon completion of the session
Lazarus Form Recovery	125709	Fill in a HTML form	Form entries saved in extension database in `databases/chrome-extension-<uuid>_0/1`	Form entries saved in database file in temporary filesystem and is not written back to the original filesystem

Table 4: Using UCOGNITO for automatically enabling private mode for add-ons.

Category	Persistent data	Firefox	Chrome
Transparent to user	Browsing history	`/<firefox-profile>/places.sqlite`	`/<chrome-profile>/History`
	Cookies	`/<firefox-profile>/cookies.sqlite`	`/<chrome-profile>/Cookies`
	Cache	`/<firefox-profile>/cache2`	`/<chrome-profile>/Cache`
	HTML5 local storage	`/<firefox-profile>/webappsstore.sqlite`	`/<chrome-profile>/Local Storage`
	Flash storage	`/<user-home>/.macromedia/Flash_Player`	`/<chrome-profile>/Pepper Data/Shockwave Flash`
User action involved	Download entries	`/<firefox-profile>/places.sqlite`	`/<chrome-profile>/History`
	Autofills	`/<firefox-profile>/{key3.db, formhistory.sqlite}`	`/<chrome-profile>/{Login Data, Web Data}`
	Bookmarks	`/<firefox-profile>/places.sqlite`	`/<chrome-profile>/Bookmarks`
	Per-site zoom level	`/<firefox-profile>/content-pref.sqlite`	`/<chrome-profile>/Preferences`
	Per-site permission	`/<firefox-profile>/permissions.sqlite`	`/<chrome-profile>/Preferences`
	SSL self-signed cert	`/<firefox-profile>/cert8.db`	`/<chrome-profile>/Origin Bound Certs`
	SSL client cert	`/<user-home>/.pki/nssdb/cert9.db`	`/<user-home>/.pki/nssdb/cert9.db`
Add-on support	Add-on storage	`/<user-home>/` with XPCOM components	`/<user-home>/` with NaCl support
	Add-on installation	`/<firefox-profile>/{extensions, extensions.json}`	`/<chrome-profile>/{Extensions, Local Extension Settings}`

Table 5: Mapping of each type of persistent data in Table 1 to the underlying file that contains it. It is possible that multiple types of persistent data is mapped to the same file, for example, for both Firefox and Chrome, browsing history and download entries are all persisted in the same file.

Website	Base	UCOGNITO	Overhead
Google.com	277.4 ms	279.6 ms	0.79%
Bing.com	207.8 ms	208.4 ms	0.29%
Twitter.com	1020.9 ms	1030.3 ms	0.92%
Facebook.com	443.9 ms	446.7 ms	0.63%

Table 8: Page loading time in Mozilla Firefox, with and without UCOGNITO. The private-aware browsing with UCOGNITO incurs negligible overhead (<10 ms).

not for the subsequent frequently invoked system calls that perform actual access (e.g. read/write), the overhead is very low.

To measure the delay that the user might experience, we compared the page load time of popular websites. To measure page load time from Google Chrome, we used an extension named "Page load time [7]" developed by `avflance`. In Mozilla Firefox, we used a different extension called "app.telemetry Page Speed Monitor 14.0.7 [22]" because "Page load time" is not available. We measured the average loading time, by accessing the page 10 times per each website. We discarded the loading time of the first access to eliminate performance skew of caching (in effect, we only measured the subsequent, cached accesses).

Table 7 and Table 8 show the benchmark results from Google Chrome and Mozilla Firefox, respectively. All results except one show around 1% of overhead, while the worst case - the loading of `twitter.com` takes 15 ms (2.5%) more time to load. Both JavaScript and page load time benchmarks exhibit fairly consistent result. Most of the time, it shows around 1% overhead; even the worst case only incurs 2.5% overhead. This shows that the overhead introduced by UCOGNITO is very negligible in practice.

7. DISCUSSION

In addition, we further discuss the advantages of UCOGNITO and its other potential applications.

Personalized private mode. UCOGNITO aims to provide personalized private mode instead of setting a model solution for a single private mode implementation. We respect that every user has his or her unique privacy needs, which might not be satisfied by the default policy provided by browser vendors. Therefore, a primary design goal and advantage of UCOGNITO is that complete control is given to the end-users with regard to private browsing.

Portable architecture. UCOGNITO is not designed solely for browsers, in fact, we believe that UCOGNITO can be readily ported to support other applications that are yet to have an incognito mode available. For example, similar to not leaving traces about browsing activities, users of video players might not want the player to leave traces about the video played, for example, recording the filename in the playlist. But to the best of our knowledge, there is no player that provides this functionality. Users have to manually delete the entry from playlist and still they are not assured that there might be traces left in other places. UCOGNITO can handle this situation perfectly by redirect any write to filesystem to a temporary location and delete this location after the incognito session. In fact, we believe UCOGNITO can be ported to multiple applications with minimal modifications.

Cross-platform design. All of the underlying techniques used by UCOGNITO are readily available since Linux kernel 3.17, hence, there is no technical obstacle to port UCOGNITO to other Linux-based OSes such as Android whose users might have more privacy incentives to use incognito mode. In addition, most non-Linux based OSes have similar substitutes of the core techniques we use. For example, `ptrace` used in UCOGNITO for system call hooking is readily available in Mac OS and can be substituted by API hooking [5] in Microsoft Windows.

8. RELATED WORK

Since UCOGNITO deals with problems in private browsing using filesystem isolation, we discuss the related works previously done as follows.

Private browsing. Private browsing is the first line of work most closely related to our work, and research in this domain mainly focus on two aspects. First, previous studies focus on measuring and preventing privacy leak of a web browser to the persistent storage. Aggarwal et al. [1] reviewed how private mode is implemented in mainstream browsers. They found the implementation of private mode in the browser does not provide enough privacy guarantee. Browsing history, DNS cache, swap files, and extensions could undermine the privacy guarantees. They developed a technical mechanism that prevents browser extensions from unintentionally leaving traces about private activities. Heule et. al. [14] proposed a new extension system design based on MAC to protect users' privacy. While these works only handles unintentional leaving of trace from the extensions, UCOGNITO can protect both reading of profile data and leaving of traces; also, not only for the extensions, the protection from UCOGNITO works on the whole web browser. Lerner el. al. [18] analyzed browser extensions to check whether the extensions are violating privacy under private mode. While their work is a static tool that only detects the violation, UCOGNITO is a runtime tool that works on both detection and prevention of privacy violation during the execution of the web browser. Gao et. al. [11] did a survey on the user perception of private browsing. They discovered several mismatches between private browsing implementation and the user expectations. In our work, we did through analysis on the privacy leaks and traces of private mode that supports their study, and we tries to solve the problem of mismatch by building a user-configurable policy to get back the control of data from the web browser developer to the user, in order to meet their expectation.

On the other hand, previous studies attempt to perform web browsing without tracking. To achieve this goal, a number of studies have been performed [10, 21, 28, 30]. However, these works are orthogonal to our work in that they attempt to prevent privacy leak to the network while our work addresses the problem of privacy leak to the persistent storage. In addition, previous works for attacking privacy using browser fingerprinting based on software/hardware configurations [25, 26] are out-of-scope, while UCOGNITO can defend against user fingerprinting based on the traces such as cookie and extension storage like Evercookie [16]. UCOGNITO can be configured to delete all traces after the browser session closes.

Sandbox and isolation mechanisms. In terms of technical approach, our work resembles application sandboxing mechanisms that separate code execution in an isolated environment and undo its effect. In the past, many of such mechanisms have been developed, e.g., Cowdancer [32], FL-COW [20], Alcatraz [19], and MBox [17]. They could prevent an untrusted program from modifying filesystem by layering a sandbox filesystem on top of actual filesystem. Although sharing a similar idea – layering filesystem – with these previous mechanisms, our work has a completely different focus. Rather than preventing an untrusted program from modifying filesystem, our work primarily utilizes layered filesystem to provide privacy guarantees. There are several other solutions that isolate the whole environment of the application from the other applications, or even from the operating system. Onarlioglu et al. introduce PRIVEXEC [27], which provides a privacy guarantee for the application execution environment. Virtual machines are frequently used for the isolation mechanisms. Previous works such as Overshadow [6], Storage Capsule [4], and Qubes OS [31] are mechanisms that provides strong isolation based on the virtual machine. From the perspective of implementation, many of aforementioned mechanisms incur OS kernel modification which significantly narrows their compatibility, (e.g., FL-COW, and PRIVEXEC), or requires virtualization that incurs a high runtime performances penalty. In contrast, we emphasize that UCOGNITO is a light weight scheme which requires no changes to OS kernel or applications.

9. CONCLUSION

In this paper, we have presented a new approach to implement private browsing. Our work was motivated by the observations that private browsing is not implemented consistently and correctly in major browsers. We developed a systematic approach to identify that browsers such as Chrome and Firefox do not clear some of the traces left behind by a private browsing session, and thus compromising privacy goals. We analyzed the browser source code to learn that developers have to put in many conditional checks to invoke the appropriate logic for the current browsing mode (i.e., private or public) .

Our new approach relieves developers from having to carefully consider private browsing, and more importantly, produces a consistent and correct private browsing mode across browsers. The main idea is to overlay the actual filesystem with a sandbox filesystem when the browser is in private browsing mode, so that no unintended leakage is allowed and no persistent modification is stored. We have implemented a prototype system called UCOGNITO on Linux. UCOGNITO requires no change to browsers and the OS kernel because the layered sandbox filesystem is implemented by interposing system calls. Our evaluations show that UCOGNITO, when applied to Chrome and Firefox, stops all known privacy leaks identified by prior work and our current study. In addition, UCOGNITO incurs only negligible performance overhead (1%-2.5%).

References

[1] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2010.

[2] E. Akhgari and M. Connor. Firefox3.1 / PrivateBrowsing / FunctionalSpec, Sept. 2008. https://wiki.mozilla.org/PrivateBrowsing.

[3] P. Battre. Chrome / Preferences / Incognito-Profile, May 2015. https://www.chromium.org/developers/design-documents/preferences#TOC-Incognito-Profile.

[4] K. Borders, E. Vander Weele, B. Lau, and A. Prakash. Protecting confidential data on personal computers with storage capsules. Aug. 2009.

[5] J. Bremer. Intercepting System Calls on x86_64 Windows, May 2012. http://jbremer.org/intercepting-system-calls-on-x86_64-windows/.

[6] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 2–13. ACM, 2008.

[7] Chrome Web Store. Page Load Time. https://chrome.google.com/webstore/detail/page-load-time/fploionmjgeclbkemipmkogoaohcdbig, July 2014. Accessed: 2015-05-16.

[8] M. Davidov. The Double-edged Sword of HSTS Persistence and Privacy, Apr. 2012. https://www.leviathansecurity.com/blog/the-double-edged-sword-of-hsts-persistence-and-privacy/.

[9] W. Drewry. SECure COMPuting with filters, Jan. 2012. http://lwn.net/Articles/498231/.

[10] M. Fredrikson and B. Livshits. RePriv: Re-imagining Content Personalization and In-browser Privacy. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*, pages 131–146, Oakland, CA, May 2011.

[11] X. Gao, Y. Yang, H. Fu, J. Lindqvist, and Y. Wang. Private browsing: An inquiry on usability and privacy protection. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society (WPES)*, Nov. 2014.

[12] Google Inc. Let others browse Chrome as a guest. https://support.google.com/chrome/answer/6130773?p=ui_guest&rd=1, May 2015. Accessed: 2015-05-16.

[13] Google, Inc. Octane 2.0 JavaScript Benchmark. http://octane-benchmark.googlecode.com/svn/latest/index.html, May 2015. Accessed: 2015-05-15.

[14] S. Heule, D. Rifkin, A. Russo, and D. Stefan. The most dangerous code in the browser. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV).*, May 2015.

[15] I. Hickson. Web storage. W3C recommendation, W3C, July 2013. http://www.w3.org/TR/2013/REC-webstorage-20130730/.

[16] S. Kamkar. evercookie – never forget. http://samy.pl/evercookie/, Sept. 2010. Accessed: 2015-05-02.

[17] T. Kim and N. Zeldovich. Practical and effective sandboxing for non-root users. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, San Jose, CA, June 2013.

[18] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *European Symposium on Research in Computer Security (ESORICS)*, Sept. 2013.

[19] Z. Liang, W. Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An isolated environment for experimenting with untrusted software. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):14:1–14:37, Jan. 2009. ISSN 1094-9224.

[20] D. Libenzi. FL-COW 0.10. http://xmailserver.org/flcow.html. January 2013.

[21] N. Mor, O. Riva, S. Nath, and J. Kubiatowicz. Bloom Cookies: Web search personalization without user tracking. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[22] Mozilla. app.telemetry page speed monitor. https://addons.mozilla.org/en-US/firefox/addon/apptelemetry/, Dec. 2013. Accessed: 2015-05-16.

[23] Mozilla. Private browsing mode warning doesn't mention that newly-installed client certificates are not cleared when exiting private browsing mode. https://bugzilla.mozilla.org/show_bug.cgi?id=475881, 2015. Accessed: 2015-05-02.

[24] Mozilla. Kraken JavaScript Benchmark (version 1.1). http://krakenbenchmark.mozilla.org/, May 2015. Accessed: 2015-05-15.

[25] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, volume 5, 2013.

[26] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[27] K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda. Privexec: Private execution as an operating system service. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[28] X. Pan, Y. Cao, and Y. Chen. I do not know what you visited last summer: Protecting users from third-party web tracking with TrackingFree browser. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[29] A. Popescu. Geolocation API specification. W3C recommendation, W3C, Oct. 2013. http://www.w3.org/TR/2013/REC-geolocation-API-20131024/.

[30] F. Roesner, T. Kohno, and D. Wetherall. Detecting and Defending Against Third-party Tracking on the Web. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, Apr. 2012.

[31] J. Rutkowska. Qubes OS. http://qubes-os.org. January 2013.

[32] J. Uekawa. Cowdancer: copy-on-write data access completely in userland. http://www.netfort.gr.jp/~dancer/software/cowdancer.html.en. January 2013.

[33] H. Ulmer. Understanding private browsing, Aug. 2010. https://blog.mozilla.org/metrics/2010/08/23/understanding-private-browsing/.

[34] A. van Kesteren and J. Gregg. Web notifications. Last call WD, W3C, Sept. 2013. http://www.w3.org/TR/2013/WD-notifications-20130912/.

[35] J. Villalobos and K. Maglione. AMO review policies, Apr. 2015. https://developer.mozilla.org/en-US/Add-ons/AMO/Policy/Reviews.

[36] J. Voung. PNaCl Translation Caching: In Javascript or In Browser, Apr. 2011. https://code.google.com/p/nativeclient/wiki/PNaClTranslationCache.

[37] T. Warren. Chrome for iOS' incognito mode isn't private, bug reveals, Oct. 2013. http://www.theverge.com/2013/10/3/4797968/chrome-for-ios-incognito-mode-not-private-bug.

[38] WebKit.org. SunSpider 1.0.2 JavaScript Benchmark. https://www.webkit.org/perf/sunspider/sunspider.html, May 2015. Accessed: 2015-05-15.