Feedback-Directed Instrumentation for Deployed JavaScript Applications

Magnus Madsen*
University of Waterloo
Waterloo, Ontario, Canada
mmadsen@uwaterloo.ca

Frank Tip
Samsung Research America
Mountain View, CA, USA
ftip@samsung.com

Esben Andreasen*
Aarhus University
Aarhus, Denmark
esbena@cs.au.dk

Koushik Sen
EECS Department
UC Berkeley, CA, USA
ksen@cs.berkeley.edu

Anders Møller
Aarhus University
Aarhus, Denmark
amoeller@cs.au.dk

ABSTRACT

Many bugs in JavaScript applications manifest themselves as objects that have incorrect property values when a failure occurs. For this type of error, stack traces and log files are often insufficient for diagnosing problems. In such cases, it is helpful for developers to know the control flow path from the creation of an object to a crashing statement. Such *crash paths* are useful for understanding where the object originated and whether any properties of the object were corrupted since its creation.

We present a feedback-directed instrumentation technique for computing crash paths that allows the instrumentation overhead to be distributed over a crowd of users and to reduce it for users who do not encounter the crash. We implemented our technique in a tool, Crowdie, and evaluated it on 10 real-world issues for which error messages and stack traces are insufficient to isolate the problem. Our results show that feedback-directed instrumentation requires 5% to 25% of the program to be instrumented, that the same crash must be observed 3 to 10 times to discover the crash path, and that feedback-directed instrumentation typically slows down execution by a factor 2x–9x compared to 8x–90x for an approach where applications are fully instrumented.

1. INTRODUCTION

Despite the best efforts of software engineers and testers, software shipped to end users still contains bugs, causing applications to crash or produce incorrect results. Failures that occur post-deployment are often reported via on-line error reporting facilities,[1] or in a bug reporting forum. Depending on the type of problem, additional information may be available along with failure reports. For example, log files may exist that contain a summary of an application's execution behavior, or a dump of an application's state at the time of a crash. However, such information is often of limited value, because the amount of information can be overwhelming (e.g., log files may span many megabytes, most of which is typically completely unrelated to the failure), or is woefully incomplete (e.g., a stack trace or memory dump usually provides little insight into how an application arrived in an erroneous state).

Many bugs that arise in JavaScript applications manifest themselves as objects that have incorrect property values when a failure occurs. This includes situations where an object is created with incorrect or missing property values, and where properties are corrupted after the object was created. In such cases, it is helpful for developers to know the control-flow path from the creation of the object of interest to the crashing statement. We will refer to such a path as a *crash path*. In the case studies reported on in this paper, we consider real-life bugs where the error message and stack trace provided with a bug report are insufficient to find and fix the underlying cause of a bug. In these case studies, information contained in the crash path provided crucial hints to developers in a debugging scenario. In principle, crash paths could be obtained by instrumenting applications so that control-flow is recorded as the program executes and exercising the application until the same failure is encountered. Unfortunately, such a "full instrumentation" approach tends to incur prohibitive runtime overhead and is indiscriminate in that much of the instrumentation occurs in regions of the code unrelated to the failure. Furthermore, in many cases, developers do not need an *entire* execution history as much of it tends to be unrelated to the bug being pursued.

In this paper, we present a feedback-directed technique for computing crash paths in scenarios where the same failure can be observed repeatedly. The technique relies on repeatedly deploying new versions of the application in which the amount of instrumentation is gradually increased to uncover successively longer suffixes of the crash path. This process continues until the complete crash path is found (i.e., the allocation site for the object of interest is found). We believe that our technique is particularly well-suited for a scenario where users collectively isolate crash paths associated with bugs in deployed software. In the scenario we envision, users

[*] The work of these authors was carried out during internships at Samsung Research America.

[1] E.g. Windows Error Reporting [22] or CrashReporter [5].

ICSE '16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884846

install applications from a central software repository (e.g., through an app store or a package manager). When a crash is encountered by a user, new versions of the application that contain additional instrumentation can be distributed to other users so that the effort of isolating crash paths is distributed over a user population. Furthermore, while our feedback-directed instrumentation approach occurs significant overhead, this tends to be up to an order of magnitude less than an approach where applications are fully instrumented. Similar crowdsourcing techniques for debugging were pioneered by Liblit et al. [18, 19] and by Orso et al. [10, 12], although they do not consider the specific problem of computing crash paths. We implemented the technique using Jalangi [28], and evaluated it on crashing executions of JavaScript applications available from GitHub. We report on instrumentation overhead and on the number of times that the same crash needs to be encountered to recover the complete crash path. In summary, the contributions are:

- We propose the notion of a *crash path* as a supplement to error messages and stack traces. A crash path records the path from the creation of an object of interest to a crashing statement.
- We present a technique for computing crash paths based on feedback-directed program instrumentation. This technique identifies increasingly longer suffixes of crash paths in successive executions of the application, by gradually increasing the amount of instrumentation.
- We implemented a prototype of our technique in a tool called Crowdie and used it to debug 10 real-world issues previously reported on GitHub. Our case study shows that error messages and stack traces are insufficient to debug these issues, and that crash paths provide useful hints to developers debugging these issues.
- Our experimental evaluation shows that only 5% to 25% of the program is instrumented, that the same crash must be observed between 3 to 10 times to recover complete crash paths, and that the feedback-directed instrumentation has a typical slowdown of 2x–9x compared to 8x–90x with full instrumentation.

Furthermore, we argue informally that our technique is well-suited to a "crowdsourcing" scenario, where the effort of identifying crash paths is distributed over a user population.

2. MOTIVATING EXAMPLES

In this section, we look at two real-world bugs in popular JavaScript libraries. What these bugs have in common is that traditional debugging information (e.g., the error message, the line on which the crash occurs, and the stack trace) is insufficient to locate the root cause of the defect. We will show how, in each case, the crash path provides information that is vital for understanding and fixing the bug.

Loki Issue #0046. Loki is a database written in JavaScript. In issue #0046, entitled "Clearing collection breaks index",[2] a user reports that "after clearing a collection I wasn't able to insert new objects without error." Such a report is typically not enough for the developer to reproduce the bug and locate its cause. A stack trace, which may be sent automatically to the server when a crash occurs, may provide some hints but is also often insufficient. To keep the example simple, imagine that we just try to execute the following four lines (`example1.js`) inspired by the text in the report, which happen to be enough to encounter the error:

```
1 var db = new loki('Example');
2 var col = db.addCollection(/* omitted */);
3 col.clear();
4 col.insert({ /* omitted */});
```

Running this code makes Loki throw an exception with the following stack trace:

```
TypeError: undefined is not a function
  at Collection.add (loki.js:1789)
  at Collection.insert (loki.js:1661)
  at Object.<anonymous> (example1.js:4)
```

This stack trace tells us that the crash occurs at line 1789 in `loki.js`, which is executed due to the call `col.insert` on line 4. Line 1789 in `loki.js` looks as follows:

```
1789   this.idIndex.push(obj.id);
```

As there is only one function call on this line, one can infer that `this.idIndex.push` is `undefined`. From this, a developer may conclude that the value of `this.idIndex` or its `push` property is incorrect. To find the root cause of the crash we need to know what is the value of `this.idIndex`, and why the `push` property is `undefined`. Was it never set, or was it perhaps mistakenly overwritten?

We cannot answer these questions easily by inspecting the stack trace and source code. We could use a tool such as `grep` to search for `this.idIndex`, but such an approach is very crude and may match statements unrelated to the crash. What we want to know is (a) where was the object referred to by `this.idIndex` created, and (b) how was the `push` property of this object modified in the time between its creation and the crash. Our Crowdie tool computes this information automatically. Applying Crowdie to this program reveals that the object referred to by `this.idIndex` originates from line 1670 in `loki.js`:

```
1670 this.idIndex = {};
```

At this point, the root cause becomes clear: this line assigns the `idIndex` property with an empty *object* instead of an empty *array*. Arrays have a `push` method, whereas ordinary objects do not, which causes the expression `this.idIndex.push` to evaluate to `undefined`, and calling the `push` method on `undefined` results in the exception being thrown. Crowdie also reveals that no writes to the `push` property of the object occur before the crash.

Upon realizing this problem, a project maintainer fixed the issue[3] by assigning an array instead of an object to `this.idIndex` on line 1670 of `loki.js`.

To find this problem using Crowdie, `this.idIndex` is designated as an object of interest. Then, Crowdie produces the following crash path (shown simplified here; details of the path format are described in Section 3.1, and details about the actual path for this bug appear in Table 1):

```
Start(loki.js:1670)      this.idIndex = {};
Return(example1.js:3)    col.clear();
Call(example1.js:4)      col.insert({ .. });
Call(loki.js:1661)       this.add(doc);
Crash(loki.js:1789)      this.idIndex.push(obj.id);
```

With this information, the developer would be able to quickly identify the buggy assignment on line 1670.

[2] https://github.com/techfort/LokiJS/issues/46

[3] Commit `5da46aeecda6046f738c6a612c2f181b21487108`

Immutable Issue #0381. Immutable is a JavaScript collection library created by Facebook. In issue #0381, entitled "subCursor (cursor.cursor('a')) returns improper type",[4] a user reports that the `cursor` method may return the wrong type of cursor. The following code fragment (`example2.js`), reported by the user, is a highly simplified version of the actual application code that triggers the bug:

```
1 var data = Immutable.fromJS({a: [1,2,3]});
2 var cursor = Cursor.from(data);
3 var deepCursor = cursor.cursor('a');
4 assert(deepCursor instanceof IndexedCursor);
```

In this example, the stack trace only tells us that the assertion on line 4 fails:

```
Error: AssertionError
  at Object.<anonymous> (example2.js:4)
```

We see that the value of `deepCursor` returned by the call to `cursor.cursor` has the wrong type, causing the assertion to fail. However, it is not obvious whether the call `Cursor.from(data)` on line 2 or the call `cursor.cursor('a')` on line 3 is at fault.

At this point, we ask Crowdie to find where the erroneous `deepCursor` object was allocated and by what path it reached the assertion. In response, Crowdie produces the following crash path:

```
Start(cursor.js:242)   ..new CursorClass(r,k,c,s);
Return(cursor.js:250)  return makeCursor(
Return(cursor.js:187)  ..subCursor(this, s);
Return(example2.js:3)  ..cursor.cursor('a');
Crash(example2.js:4)   assert(deepCursor ..);
```

This crash path shows that the object was created on line 242 in function `makeCursor` in `cursor.js`, which looks as follows:

```
236 function makeCursor(r, k, c, value) {
237   if (arguments.length < 4) {
238       value = r.getIn(k);
239   }
240   var s = value && value.size;
241   var CursorClass = Iterable.isIndexed(
        value) ? IndexedCursor : KeyedCursor;
242   return new CursorClass(r, k, c, s);
243 }
```

Looking at this code, we see that an object with the wrong type could be allocated for two reasons: (1) function `isIndexed` is buggy, or (2) `value` somehow has an incorrect value. We also note that `makeCursor` implements overloading by checking the number of arguments on line 237.

The crash path produced by Crowdie not only tells us that the object originated from line 242 but also provides information about the call stack when this object was created. This call stack can be obtained by observing that `Return` labels on the produced path must have had matching calls that occurred earlier. From this call stack, we learn that `makeCursor` was invoked by `subCursor`:

```
249 function subCursor(cursor, k, value) {
250     return makeCursor(
251         cursor._rootData,
252         newKeyPath(cursor._keyPath, k),
253         cursor._onChange,
254         value
255     ); }
```

Crucially, we observe that `subCursor` always passes four arguments to `makeCursor` on line 250 and that the fourth argument `value` is an argument to `subCursor` itself. Since it always passes four arguments, the condition on line 237 will always evaluate to false when called from `subCursor`. Furthermore, our tool tells us that `subCursor`, in turn, was called by `KeyedCursorPrototype.cursor`. The calling code of that function looks as follows:

```
187   ..subCursor(this, s);
```

Here, we can see that `subCursor` is passed two arguments even though it has three parameters, so the third parameter will take on a default value of `undefined`. The `subCursor` function in turn calls `makeCursor` with this parameter, which checks if it received four arguments instead of checking if the fourth argument has the default value `undefined`! The core issue is thus that `makeCursor` implements an overloading check in a way that `subCursor` did not anticipate.

The fix[5] for the issue is to change the overloading check in line 237 to `value === undefined`.

These examples demonstrate how Crowdie can provide information that is vital for understanding and fixing bugs, especially in cases where the error message and stack trace of a crash provides insufficient information. Note that Crowdie is not a fully automated debugging process: a nontrivial amount of human ingenuity may still be needed. However, in both cases, the crash path was sufficient to debug the issue. In particular, the (typically much longer) full execution path from the start of the application was not needed. Here, we have presented the crash paths as plain text, but we can imagine a scenario where IDEs naturally show the paths and allow the programmer to "jump" forwards and backwards through the statements on the path.

3. TECHNIQUE

A high-level architecture of the kind of system in which we envision our technique to be embedded is depicted in Figure 1. We assume a distribution model where users (shown on the left) install their applications through a central server (shown on the right). For example, in the case of client-side web applications the server is responsible for sending the JavaScript code to the users. As another example, in the case of a server-side application, the group of "users" may consist of a cluster of servers running the application. Finally, we envision that app stores for (mobile or non-mobile) devices could be extended with the techniques presented in this paper. To make this practical, an approach to feedback-directed instrumentation would be required that does not require pushing out new versions to users frequently. The latter could be accomplished by having selected parts of the instrumentation in an application be enabled or disabled based on an instrumentation policy that it periodically receives from the server.

The server in Figure 1 has several components, including an *instrumentor* component for instrumenting applications according to an *instrumentation policy* that specifies the functions that need to be traced. It also contains an *analyzer* component that decides which functions should be instrumented based on an analysis of traces of crashing executions that it receives from users and produces an instrumentation policy accordingly. In the scenario we envision, the instrumentation policy is updated repeatedly, each time the same failure is encountered.

[4] https://github.com/facebook/immutable-js/pull/381

[5] Commit `91afb4100eaed6375ceba239ad624a4453684429`

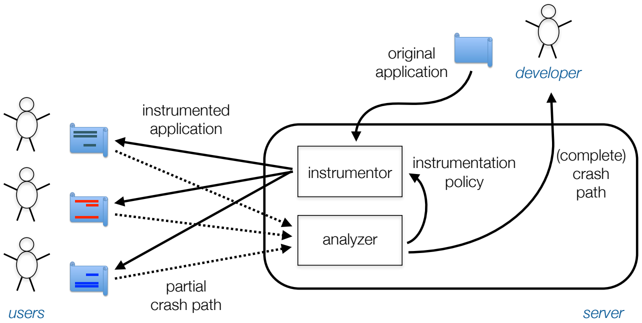

Figure 1: A central server deploys instrumented versions of the program to its users. Crashing users report information back to the server.

As users execute the application, some of them will inevitably encounter a crash (e.g., an uncaught exception, an assertion failure, a suspicious operation [11], or a security policy violation). When this happens, the added instrumentation ensures that a crash path is recorded that reflects the set of functions specified in the instrumentation policy. This crash path identifies at least in what function the crash occurred and what local and global variables may be relevant to the crash (see Section 4 for more details). The instrumentation ensures that this crash path is uploaded to the analyzer component of the server, which in response updates the instrumentation policy accordingly. Note that, in general, the crash paths that users upload to the server are partial crash paths, in the sense that they may not contain the full flow of control from the allocation of an object to the point where the crash occurred. However, when the analyzer component determines that a bug was localized because the crash path is complete, the complete crash path is passed to the developer and all instrumentation is disabled for the entire user population.

Various features can be added to the architecture discussed above. For example, the server can be configured to trace multiple bugs concurrently, and these efforts can be crowdsourced over different subsets of the user population. Furthermore, the server can compare crash paths for different bugs that it is tracking concurrently, try to determine when they are duplicates, and prefer the shortest candidate.

3.1 Execution Paths

The grammar shown in Figure 2 defines the structure of execution paths constructed by our technique. An execution path $p \in \mathit{Path}$ is a sequence of labels that are recorded during program execution. We capture intra-procedural control-flow using the three labels:

$$\mathit{Intra} := \texttt{Branch}(s,c) \mid \texttt{Start}(s) \mid \texttt{Crash}(s)$$

The $\texttt{Branch}(s,c)$ label records the value of a conditional c in an `if` or `while` statement s. The $\texttt{Start}(s)$ label records that the object of interest originates from statement s. The `Crash`(s) label records the statement s where the program crashed. We use heuristics to determine the object(s) of interest in an a crashing statement. For example, if the statement `o.f()` crashes because `o.f` is `undefined` then the object `o` is of interest. On the other hand, if the statement crashes because `o.f` is not a function then `o.f` is of interest. We define similar heuristics for other types of statements.

We say that an execution path that begins with a `Start` label and ends with a `Crash` label is a *crash path*. Intuitively, a crash path identifies a statement where an object was created and a sequence of control-flow decisions leading to a crash where that object was somehow involved. A path that ends with the `Crash` label, but does not begin with the `Start` label is a *partial* crash path.

$$\begin{aligned}
&\mathit{Intra} := \texttt{Branch}(s,c) \mid \texttt{Start}(s) \mid \texttt{Crash}(s,v)\\
&\mathit{Inter} := \texttt{Call}(s,f) \mid \texttt{Return}(s) \mid \texttt{Enter}(f) \mid \texttt{Exit}(f)\\
\ell \in{}& \mathit{Label} := \mathit{Intra} \mid \mathit{Inter}\\
p \in{}& \mathit{Path} := \mathit{Label}, \cdots, \mathit{Label}\\
f \in{}& \mathit{Fun} = \text{the set of functions in the program}\\
s \in{}& \mathit{Stm} = \text{the set of statements in the program}
\end{aligned}$$

Figure 2: Grammar for execution paths.

Inter-procedural control-flow is tracked using four labels:

$$\mathit{Inter} := \texttt{Call}(s,f) \mid \texttt{Return(s)} \mid \texttt{Enter}(f) \mid \texttt{Exit}(f)$$

The $\texttt{Call}(s,f)$ label records a function invocation from call site s to function f. The $\texttt{Return}(s)$ label records the completion of a call at statement s. The $\texttt{Enter}(f)$ label records that the control-flow has entered function f. The $\texttt{Exit}(f)$ label records that the control-flow has left function f.

Intuitively, the call/return labels represent a function invocation as seen from the *caller*, whereas the enter/exit labels represent a function invocation as seen from the *callee.* The reason for requiring both is that our technique does not instrument every function, and thus to observe whether or not calls are missing from a trace, we must record information inside both the caller and the callee. In particular, an execution path that has the labels

$$\texttt{Call}(s,f) \rightarrow \texttt{Enter}(f) \rightarrow \texttt{Exit}(f) \rightarrow \texttt{Return}(s)$$

is complete because it records the invocation of f from s, the execution inside f and the return of control to s. On the other hand, the execution path $\texttt{Call}(s,f) \rightarrow \texttt{Return}(s)$, which lacks the `Enter` and `Exit` labels, records a call to an *uninstrumented* function f (if f was instrumented then its execution would have generated the appropriate enter/exit labels). We will discuss in Section 3.3 how this information is used to determine whether a crash path is complete.

Example I. Consider the code fragment of Figure 3 and the execution path that is generated when `f` is invoked with the arguments `x = 7`, `y = 8` and `o = {}` (the empty object): For this example, execution crashes when line 5 is reached, because `o.missingMethod` has the value `undefined` at that point, and any attempt to call `undefined` as a function results in a crash. The full crash path[6] is:

$$\begin{aligned}
&\texttt{Start}(s_1) \rightarrow \texttt{Call}(s_7,\texttt{f}) \rightarrow \texttt{Enter(f)} \swarrow\\
&\rightarrow \texttt{Branch}(s_3,\texttt{true}) \rightarrow \texttt{Branch}(s_4,\texttt{true}) \rightarrow \texttt{Crash}(s_5)
\end{aligned}$$

Example II. In the example of Figure 3, the crash happened due to the arguments passed to `f` and a stack trace would contain all neccessary information to debug the issue. However, this is not always the case as illustrated by the program of Figure 4. For this example, the crash path is:

$$\begin{aligned}
&\texttt{Start}(s_9) \rightarrow \texttt{Branch}(s_{10},\texttt{true}) \rightarrow \texttt{Exit(f)} \swarrow\\
&\rightarrow \texttt{Return}(s_{13}) \rightarrow \texttt{Branch}(s_{14},\texttt{true}) \rightarrow \texttt{Crash}(s_{15})
\end{aligned}$$

[6] We will use the notation s_l where l is some line number in this example and subsequent examples to identify the program construct at line l.

```
1 var empty = {};
2 function f(x, y, o) {
3   if (x > 5)
4     if (y > 5)
5       o.missingMethod();
6 }
7 f(7, 8, empty);
```

Figure 3: Example I.

```
 8 function f(x) {
 9   var o = {};
10   if (x > 5)
11     return o;
12 }
13 var o = f(7);
14 if (?)
15   o.missingMethod();
```

Figure 4: Example II.

```
16 var x;
17 function f() { g(); }
18 function g() { h(); j(); k(); }
19 function h() { i(); }
20 function i() { }
21 function j() { x = {}; /* empty object */ }
22 function k() { l(); m(); }
23 function l() { }
24 function m() { x.missingMethod(); }
25 f();
```

Figure 5: A JavaScript program.

Note that the path begins inside function `f` when the object `o` is created on line 9 and that it includes the branch choices made until the crash at line 15. Anything that occurred prior to that time is not part of the crash path. In particular, the crash path does not record the initial `Call` and `Enter` labels for `f`, but it *does* record the `Exit` and `Return` labels since they occurred after the object was created.

3.2 Feedback-Directed Instrumentation

Figure 5 shows a small program that we will use to illustrate our feedback-directed instrumentation approach. In this example, the function `f` is called by top-level code and invokes `g`. This function, in turn, invokes `h`, `j`, and `k`. The call to `h` invokes `i`. Function `j` creates an empty object and assigns it to variable `x`. The call to `k` invokes `l` and `m`. Inside `m` a crash occurs due to a missing method on `x`.

Our goal is to discover the crash path beginning with the creation of the empty object inside `j` and ending with the crash inside `m`. We could discover such paths by instrumenting the entire program, but that would be costly. Instead, we propose to perform instrumentation of the program in a feedback-directed manner, where we gradually instrument more functions until the complete crash path is discovered.

The intuition behind our approach is that instrumentation is performed in a breadth-first manner starting backwards from the function in which the crash occurs. Figure 6 illustrates this concept. Initially, the crash is observed to occur inside `m`, which causes `m` to be instrumented. The next time the crash is seen, it is observed that `m` was called from `k`, so `k` is instrumented as well. Since `k` contains two call sites its execution will produce a path that includes two call labels, labeled $\texttt{k}_1$ and $\texttt{k}_2$ in Figure 6. In the third iteration, `k` and `m` are instrumented and calls from `g` to `k` and from `k` to `l` are observed. In the fourth iteration, `g`, `k`, `l`, and `m` are instrumented and calls to `h` and `j` are observed. In the fifth and final iteration, functions `f`, `g`, `h`, `j`, `k`, `l`, and `m` are instrumented. Now `j` is instrumented, and each object that is allocated inside a function is tagged with a special property recording where it was created. When the crash occurs, the instrumentation discovers that this property is set, and reports the crash path: $\texttt{Start}(\texttt{j}) \rightarrow \texttt{Return}(\texttt{g}_2) \rightarrow \cdots \rightarrow \texttt{Call}(\texttt{k}_2, m) \rightarrow \texttt{Crash}(s_{24})$ (this path is highlighted in Figure 6 using dashed red lines). Note that the path does not include any control-flow that happened prior to entering `j`. In the end, `f`, `g`, `h`, `j`, `k`, `l`, and `m` have been instrumented, whereas function `i` and the top-level-code are left uninstrumented.

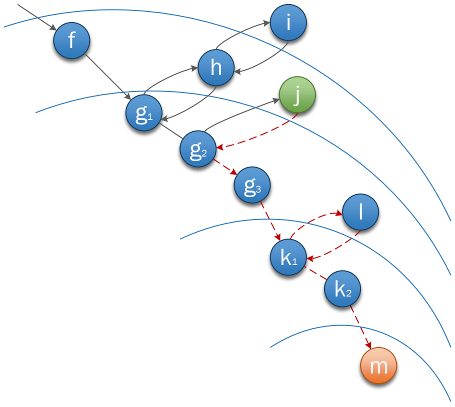

Figure 6: Control flow for the program in Figure 5. The object is created inside function `j`. The crash occurs inside function `m`. The crash path is shown in dashed red lines.

This feedback-directed process continues until the creation site of the object is discovered. In the worst case, the entire program may end up being instrumented, including functions whose execution was irrelevant to the observed crash (e.g., function `h` in the example). However, as we shall see in Section 5, it is often the case that only a small fraction of the code is instrumented in practice. A limitation of the technique is that it cannot locate the "source" of uninitialized variables and fields, since the *absence* of data flow cannot be attributed to any particular statement in the program.

3.3 Algorithm

We now present a high-level description of our algorithm for feedback-directed instrumentation. A key challenge in computing crash paths in a feedback-directed manner is dealing with situations where fragments of a path are missing because functions are called that have not been instrumented yet. For example, assume we have a situation where an instrumented function f invokes an uninstrumented function g. This gives rise to a path like:

$$\cdots \rightarrow \texttt{Call(s, g)} \rightarrow \texttt{Return(s)} \rightarrow \cdots$$

Note that, since g is not instrumented, no enter/exit labels are generated for it. Such a path is incomplete and indicates that g should be instrumented in the next iteration.

Figure 7 shows our algorithm as a function that takes one argument, s, the statement where the crash occurred. We maintain the last seen execution path in variable p and the set of instrumented functions in I. Initially, the path contains only the `Crash` label. We then repeatedly instrument and execute the program until a crash occurs. The outer

```
FEEDBACK-DIRECTED-INSTRUMENTATION($s \in Stm$)
var $p : Path = \texttt{Crash}(s)$ // the current path
var $I : \mathcal{P}(Fun) = \{s.fun\}$ // instrumented functions
repeat
    for each label $l$ on path $p$
        // instrument callee:
        if $l \equiv \texttt{Call}(s, f)$ do $I = I \cup \{f\}$
        // instrument caller:
        if $l \equiv \texttt{Enter}'(c, f)$ do $I = I \cup \{c\}$
    repeatedly execute the program until a crash occurs
        $p'$ = the path recorded in the last execution
        if $p'$ is a (partial) crashing path that matches $p$
            $p = p'$ // a longer path has been found
until $p$ begins with a Start label and contains
    a matching Enter label for each Call label
return $p$ // report the crashing path to the developer
```

Figure 7: Pseudo-code for the algorithm.

loop terminates once we have found a complete crash path by checking that p begins with a `Start` label and contains a matching `Enter` label for each `Call` label. Inside the loop, we perform two actions: (i) adding instrumentation for uninstrumented callers and callees based on the last observed partial crash path, and (ii) repeatedly execute the program until the same crash is encountered (i.e., deploy a new instrumentation policy and wait for a user to encounter the crash again) We say that two crashes are the same if their crash paths match. Formally, a crash p *matches* another crash path p' if the sequence of labels in p can be obtained by removing from p' any labels that do not occur in p.

We use a slightly extended version of the `Enter` label in Figure 7. Specifically, $\texttt{Enter}'(c, f)$ represents both function f being entered and its caller, c. This is necessary in cases where a callee is instrumented, but not its caller. Our implementation obtains a function's caller by runtime inspection of the stack via the special JavaScript `caller` property.

Discussion. Our algorithm assumes that the same crash path is encountered repeatedly by users suffiently many times to localize the error. For high-priority bugs that occur only once (or rarely), the instrumentation overhead associated with tracking such bugs should remain low because the crash path will remain short unless the same bug is encountered repeatedly. Thus, we expect our technique to have reasonable performance characteristics. However, one could easily imagine a scenario where the techniques presented in this paper are deployed only on high-priority bugs that have been seen a certain number of times, to avoid introducing overhead due to bugs that occur only rarely.

In each iteration of the algorithm at least one additional function is instrumented, so the algorithm is guaranteed to terminate. In the worst case, this may result in the entire program being instrumented.

3.4 Events

Until now, we have assumed that the control-flow in a program is determined by the usual intra-procedural constructs (if, while, for, ...) together with inter-procedural calls and returns. However, in the case of JavaScript, we must also consider asynchrony and events. In an event-based program, event listeners (also commonly referred to as *callbacks*), are registered and executed in response to certain events, e.g., the completion of a network request. For such programs, the call stack does not contain information about what happened prior to the execution of the current event listener.

```
26 var cache = {};
27 var net = require('net');
28 var server = net.createServer(function(c) {
29   if (cache.data.length != 0) {
30     c.write(cache.data);
31   }
32 });
33 server.listen(80, function() {
34   fs.readFile('a.txt', function(err, data) {
35     cache.data = data;
36   });
37 });
```

Figure 8: An event-driven Node.js program.

Figure 8 shows an example of an event-driven Node.js application. This program contains a subtle race condition that may cause an exception. In particular, there is no guarantee that the initialization of the `cache.data` property by the function declared on line 34 will take place before this property is accessed inside the function declared on line 28. If the access takes place before the initialization, an `undefined` de-reference error will occur on line 29.

Applying our technique to such an execution of this program would reveal the following crash path:

$$\texttt{Start}(s_{26}) \to \cdots \to \texttt{Enter}(f_{33}) \to \cdots \to \texttt{Exit}(f_{33}) \to \cdots \swarrow$$
$$\to \texttt{Enter}(f_{28}) \to \cdots \to \texttt{Crash}(s_{29})$$

which shows that the listener function on line 28 was executed, but the listener function on line 34 was not. From this, a developer can infer that the `cache.data` field was uninitialized on line 29 and that the code should be changed so that the initialization is guaranteed to happen.

Accurate tracking of event listeners presents a minor challenge for our technique since an event listener is invoked without any corresponding call site in the source code. Fortunately, event listeners are only executed when the call stack is empty and they maintain the stack discipline. To capture control-flow due to event listeners we instrument every Node.js function that may cause the registration of a listener. For example, in Figure 8 the Node.js functions `net.createServer`, `server.listen` and `fs.readFile` are instrumented to record that the functions declared on lines 28, 33 and 33 are event listeners. With this knowledge, we can detect which event listeners were (and were not) executed when a crash occurs. Once we know the relevant event listeners, we instrument them as well as the functions where they were registered. In the example, we discover that event listeners declared on lines 28 and 33 were executed. Thus, the registration sites `net.createServer` and `server.listen` are relevant, so their containing function will be instrumented. This allows the technique to trace control flow backwards through event listeners.

In general, many similar executions may exist in which event handlers are scheduled in different orders. In its current form, our technique does not attempt to identify and exploit similarities between executions. For example, consider a scenario where a failing execution involves a sequence of event handlers f, g, and another involving a sequence f, h, g, where event handler h is unrelated to the failure. In such cases, it may be preferable to focus only on the crash path for the former execution, because it omits information unrelated to the failure. In principle, our technique can be extended to give priority to executions in which the fewest number of event listeners is executed before a crash.

4. IMPLEMENTATION

We have implemented our technique using the Jalangi instrumentation framework [28] in a tool called CROWDIE.

Overall Structure. The implementation splits the technique into two phases. In the *detection phase*, the program is instrumented to assign every function object a unique identifier and a global error handler is installed. This handler catches any thrown exception that reaches the top-level and records in which function the exception was thrown. The detection phase also determines which event listeners (if any) were executed before the crash. The detection phase is run repeatedly until an exception is encountered. Its runtime overhead is negligible since only one instruction per function declaration is added. A separate *isolation phase* implements the feedback-directed technique of Section 3.3. Only the last 10,000 labels are tracked, to prevent the crash path from growing too large.

Instrumentation Details. Figure 9 presents some of the instrumentation rules used in our tool. Here, we aim to be informal and do not describe the finer details of the transformations used, and just sketch their overall structure. More details can be found in the original paper on the Jalangi instrumentation framework [28]. A conditional statement labeled s is instrumented to record the `Branch` label (lines 38–41). Lines 42–43 show how a function call s is instrumented to record `Call` and `Return` labels. Likewise, lines 44–47 show how a function declaration f is instrumented to record the `Enter` and `Exit` labels[7]. Finally, an object creation statement s is instrumented to record the length of the path at the moment of creation (lines 48–50). This means that every object (created inside a function selected for instrumentation) knows its offset inside the path. In addition to these instrumentation steps, we assume that the program contains a special `crash(o)` function call where o is the object of interest. This call is intercepted by the instrumentation and generates the `Crash(s)` label, but it also checks if `__path_offset__` is set to the object o. If so, the origin of o has been found, and the `Start` label is inserted at the offset. If not, we have not yet instrumented the function in which the object was created and no `Start` label is generated.

Other JavaScript Features. Our tool implements the technique of Section 3.3 as well as the mechanisms related to event handling discussed in Section 3.4. Features such as exceptions, getters and setters, and native functions require additional support, and can be added with modest effort. Support for these features is in progress, but has not been needed in any of the case studies discussed in Section 5 (and we did not exclude any candidate programs due to lack of feature support).

5. EVALUATION

The evaluation of our technique aims to answer the following research questions:

Q1: How useful are the crash paths computed by our tool for understanding and fixing real bugs?

```
38 s: if (e) { s1 } else { s2 }
39    => var c = e;
40       rec(Branch(s, c));
41       if (c) { s1 } else { s2 }
42 s: f()
43    => rec(Call(s, f)); f(); rec(Return(s));
44 f: function() { body }
45    => function() {
46          rec(Enter(f)); body; rec(Exit(f))
47       }
48 s: new Object();
49    => var tmp = new Object();
50       tmp.__path_offset__ = ...;
```

Figure 9: Instrumentation rules.

Q2: How many times must a crash be observed before the crash path has been found?

Q3: How much runtime overhead is incurred by users of our tool for computing crash paths?

The rationale behind Q1 is to determine whether the information computed by our tool is useful for debugging, particularly in cases where the information contained in error messages and stack traces is insufficient. The purpose of Q2 is to determine if the number of times that a bug needs to be encountered is reasonably low. This is relevant because if the same crash path needs to be encountered many times by different users, then the usefulness of the technique would be limited to bugs that occur very often. Lastly, Q3 aims to determine whether runtime overhead is acceptable.

5.1 Experimental Methodology

CROWDIE produces information that is intended to assist developers with debugging, but manual effort remains required in diagnosing the problem. Therefore, we opt for an evaluation based on case studies, in which we apply our tool to 10 real bugs in open source programs taken from GitHub. For these programs, we compute crash paths with our tool, manually inspect these crash paths to determine whether they are helpful (Q1), and measure various aspects of the tool's execution behavior to answer Q2 and Q3.

Selection Criteria. We chose subject programs by looking at bug reports for popular JavaScript projects on GitHub. These applications had to satisfy some limitations of our implementation: (a) the program did not make use of `eval`, and (b) the program was runnable on Node.js version 0.12. Furthermore, we required that the reported bug could be reproduced with modest effort, and that the reported issue had an identifiable place in the source code where the problem was observable (i.e., a crash caused by an exception being thrown, an assertion failure, or an incorrect returned value). Lastly, we excluded "easy bugs" where the bug was local to the same function in which the crash occurred, i.e., situations where the line number in an error message or in a stack trace would suffice to diagnose the problem quickly.

Subject Programs and Issues. For triggering the bug of interest we use small bug triggering examples provided in the bug reports, as seen twice in Section 2. This creates a much simpler scenario than the one CROWDIE is intended to be used in, but no bug reports describe a complete scenario where the bug is encountered in a production environment. Each bug reporter has spent time manually creating these

[7] In reality this instrumentation is more complicated since the `body` may contain multiple exit points, e.g. explicit return statements or exceptional control-flow, which must be taken into account by the instrumentation.

Program and Issue				Instrumentation		Recorded Path		
Program	Issue	Lines	Func.	Iterations	Func. (%)	Length	Lines	Func. (%)
AlaSQL (v0.0.36)	#0092	12,561	752	5 (+ 1)	34 (5%)	74	38	8 (1%)
Bucket-JS (v1.81)	#0006	2,393	167	4 (+ 1)	10 (6%)	44	8	4 (3%)
Esprima (v2.1)	#0299	4,410	182	8 (+ 1)	45 (25%)	349	78	26 (14%)
Esprima (v1.2)	#0560	3,762	151	7 (+ 1)	37 (25%)	219	54	24 (16%)
Esprima (v2.2)	#1042	5,314	222	8 (+ 1)	49 (22%)	185	53	23 (10%)
Immutable (v3.6.2)	#0381	5,147	674	3 (+ 1)	10 (2%)	13	9	3 (1%)
Immutable (v3.7.3)	#0489	4,935	635	9 (+ 1)	52 (8%)	90	30	14 (2%)
Loki (v1.0)	#0042	2,599	132	5 (+ 1)	20 (15%)	46	19	4 (3%)
Loki (v1.0)	#0046	2,599	132	4 (+ 1)	8 (6%)	26	14	4 (3%)
Redis (v0.7.1)	#0067	1,529	87	3 (+ 1)	27 (31%)	17	9	4 (5%)

Table 1: Summary of main results from the case study.

		Performance benchmarks				Unit tests			
Program	Issue	Tests	Orig.	Partial instr.	Full instr.	Tests	Orig.	Partial instr.	Full instr.
Alasql (v0.0.36)	#0092	14	95 s	188 s (2x)	1929 s (20x)	415	15s	43 s (3x)	158 s (11x)
Esprima (v2.1)	#0299	7	40 s	315 s (8x)	321 s (8x)	1229	5 s	86 s (17x)	96 s (19x)
Esprima (v1.2)	#0560	7	40 s	297 s (7x)	325 s (8x)	1229	6 s	70 s (12x)	82 s (14x)
Esprima (v2.2)	#1042	7	40 s	332 s (8x)	332 s (8x)	1229	6 s	91 s (15x)	102 s (17x)
Immutable (v3.6.2)	#0381	6	6 s	27 s (5x)	726 s (121x)	258	5 s	13 s (3x)	422 s (84x)
Immutable (v3.7.3)	#0489	6	6 s	360 s (60x)	718 s (120x)	258	4 s	116 s (29x)	414 s (104x)
Loki (v1.0)	#0042	9	13 s	112 s (9x)	1164 s (90x)	50	5 s	19 s (4x)	25 s (5x)
Loki (v1.0)	#0046	9	13 s	15 s (1x)	1172 s (90x)	50	5 s	13 s (3x)	25 s (5x)

Table 2: Summary of performance results.

small examples before the reporting the bug, Crowdie could potentially alleviate the need for this manual work.

The leftmost four columns of Table 1 identify the selected subject programs and their associated bug reports (all taken from GitHub). Each row in the table corresponds to one bug report and one debugging scenario. The Program column shows the name of the application/library, the Issue column shows the number assigned to the bug report. The Lines and Functions columns show the number of lines in the source code (including whitespace and comments) and the total number of function declarations, respectively.

AlaSQL is a JavaScript SQL database. Buckets is a data structure library. Esprima is a JavaScript parser. Immutable is a collection library. Loki is a JavaScript database. Redis is a Redis client for JavaScript. The bugs include incorrectly returned objects, field values that are inadvertently corrupted, incorrect control-flow due to argument passing, and various type-related errors.

Process. In the case studies, we used the following 5-step process for diagnosing a bug report using our tool: (1) run the program and observe that a crash occurs; (2) manually identify the function and line which caused the crash; (3) manually select a subset of relevant local and global variables related to the function and line of interest (frequently only a single variable was selected). In the fully automated scenario the variables would be selected by heuristics, as discussed earlier; (4) run our tool with the given line and variables as input to compute a crash path; (5) manually understand and debug the issue using the computed crash path. In some cases, the crash was caused by an exception that was thrown from a generic error handling function. In such cases, we followed the above steps, except that we focused on the previous function on the call stack.

5.2 Experiments

We now discuss the results obtained by running our tool.

Quantitative Results: Instrumentation and Paths. The columns under the header Instrumentation in Table 1 show some key characteristics of our feedback-directed instrumentation method. The column Iterations shows the number of times the same bug had to occur for the technique to find the complete crash path. Moreover, the column Functions shows the number of functions instrumented by the technique in its final iteration, both as an absolute number and as a percentage of the total number of functions. For example, for Esprima issue #0560, 7 executions were required before the complete crash path was found plus one execution to detect the crash in the first place.

The columns under the header Recorded Path report characteristics of the complete crash path found by the technique. Here, the column Length reports the length of the crash path (i.e., the total number of labels), and the column Func. counts the number of functions on the crash path. Regarding the last metric, recall that a function might be instrumented although it is not on the crash path, so it is interesting to see whether the technique instruments many functions unnecessarily. Returning to Esprima issue #0560, during the last iteration 37 functions where instrumented corresponding to 25% of the total number functions in the program. Finally, the crash path contained 219 labels distributed over 54 source code lines in 24 functions corresponding to only 16% of the total number of functions. Looking at 219 labels might seem like a daunting task for a programmer, but multiple labels can be present on each of the 54 source code lines, e.g., in the case of loops, and some labels are implied by others, e.g. `Call`/`Enter`. Thus, the programmer has to look at significantly fewer statements/labels.

Quantitative Results: Performance. Column Func. under the heading Instrumentation in Table 1 shows the number (and percentage) of functions instrumented in the final iteration by the feedback-directed technique. Only between 2% and 31% of functions are ultimately instrumented, suggesting good potential for performance improvement compared to full instrumentation.

To measure actual impact on runtime performance we exercised the programs using benchmarks and unit tests available in the source repositories. Table 2 shows the results.

Starting with the performance benchmarks, the row for AlaSQL issue #0092 shows that the benchmark program contained 14 different executions and the original program took 95 seconds to execute them all. Using our feedback-directed approach, 34 functions are instrumented (see Table 1), resulting in a total running time of 188 seconds, i.e., a slowdown by a factor 2x. In contrast, running the benchmark with full instrumentation took 1929 seconds, i.e., a factor 20x overhead. Continuing, AlaSQL's unit test suite comprises 415 individual tests which took 15 seconds to run originally; this was 3x slower with feedback-directed instrumentation, and 11x slower with full instrumentation. For the three Esprima issues there is negligible difference in performance between feedback-directed and full instrumentation. For the remaining 5 programs there are significant differences ranging from 1.1x to 121x.

In general, the performance benchmarks show a larger performance difference than the unit tests when comparing partial and full instrumentation. We postulate this is because the unit tests cover a broad range of functionality and exercise the code more evenly (including the instrumented code), whereas the benchmark programs presumably exercise only performance critical components of the code that may avoid instrumentation with our technique.

We investigated why our technique worked so poorly for Esprima. We found that Esprima is divided into two components; a lexer and a parser. In each case, an unexpected token was generated by the lexer, and then caused a crash in the parser. The problem was not the actual token itself, but the control-flow that followed. Regardless, this architecture and the tight connectedness of the parser itself meant that large parts of the program were ultimately instrumented.

The Buckets and Redis programs were not included in Table 2 since they had no performance benchmark suites.

In summary, if we exclude the best and worst running times for both techniques, then feedback-directed instrumentation has a typical overhead of between 2x–9x compared to an overhead of 8x–90x for full instrumentation.

Usefulness. We previously discussed two of our case studies in Section 2 that illustrated how it can be useful to know: (1) where an object was allocated, (2) what the call stack looked like during that allocation and (3) what properties were not written since that allocation. Space limitations keep us from discussing the remaining 8 case studies in detail, but they cover similar, and more complex, issues. In each case, the crash path provides crucial information about object allocation and initialization, the call stack structure at points of interest, and the presence or absence of writes to a property of interest. In each case, some amount of human ingenuity remains necessary but we believe that the crash paths computed by CROWDIE would be helpful for developers. In particular, the size of the code base that developers need to consider is greatly reduced by allowing them to focus on only the source lines in the crash path; these reductions can be seen in Table 1. A detailed analysis of each of these case studies can be found in a technical report [20].

5.3 Summary of Results

We can now answer the research questions stated earlier.

Q1. The crash paths computed by CROWDIE can provide useful assistance to developers, but by themselves are not a panacea: some amount of human ingenuity remains necessary to complete the debugging task.

Q2. In our case studies, between 4 and 9 iterations (executions) were needed to discover the crash path. This is a relatively low number, suggesting that our technique may be generally useful for widely deployed software, except for bugs that are very rarely encountered.

Q3. The typical runtime overhead of the feedback-directed instrumentation in CROWDIE ranges from 2x–9x. While this is often dramatically better than the overhead of full instrumentation (8x–90x), work remains to be done on making the instrumentation more efficient. Furthermore, as the results for Esprima suggest, the architecture of certain applications may make them unsuitable candidates for the technique.

5.4 Threats to Validity

We selected our subject programs based on issues reported on GitHub. While these programs are widely used, they may not be representative of all programs and likewise the reported bugs that we investigated may not be representative either. Furthermore, we selected bug reports that involved crashes (thrown exceptions, assertions errors, ...) but not every bug necessarily manifests itself as a crash. When we debugged these issues we had no prior knowledge of the program. Thus, it is possible that a developer familiar with source code might have debugged the program differently. On the other hand, the debugging scenario we faced, with no prior knowledge of the codebase, is the hardest possible.

Another valid concern is that the experiments are all based on small snippets of code that trigger the bugs. These scenarios may differ from real-life deployment scenarios, for which the results could be very different from the ones we report here. These more complex scenarios might require longer crash paths and make CROWDIE more expensive to use, but extra complexity will also make the partial instrumentation of CROWDIE even more tractable compared to naive full instrumentation.

6. RELATED WORK

Numerous techniques have been developed to support debugging. A key property of our approach is that it helps obtaining critical information about failures that occur in deployed JavaScript programs. This setting is particularly well suited for crowdsourced analysis that collects data from user executions, since it is easy to deploy new instrumentation policies. Although we focus on errors that manifest as uncaught exceptions in the JavaScript programs, our technique works more generally, for example, also when debugging assertion failures, errors related to suspicious coercions and other bad coding practices [11], or DOM-related faults [26].

Record and replay techniques. The ability to record and replay failing executions is valuable for debugging, as demonstrated by, e.g., Narayanasamy et al. [23]. We argue that knowing the part of the execution history that we call the crash path is often sufficient for the developer to debug the crash, and is cheaper to produce in a JavaScript setting. The BugRedux framework by Jin and Orso [12] gathers partial execution traces from deployed programs and then uses symbolic execution to synthesize reproducible failures. SymCrash [8] instead uses dynamic symbolic execution and more selective instrumentation. We avoid the need for symbolic execution by the use of iterative instrumentation. Some techniques utilize static analysis to reduce the instrumentation overhead [24, 31], however, the dynamic language features in JavaScript are known to cause considerable challenges for static analysis [3].

One challenge to dynamic analysis of deployed JavaScript applications is that instrumenting JavaScript programs is known to incur a large runtime overhead. Techniques that attempt to reduce the overhead by only logging sources of nondeterminism, as e.g. Chronicler [6], are difficult to apply to JavaScript. Even though our implementation uses the state-of-the-art Jalangi infrastructure [28], we observe a substantial runtime overhead when full instrumentation is enabled, which necessitates the more selective instrumentation. Several other tools described in the literature are capable of recording live executions, which can subsequently be analyzed by the developers when debugging. Mugshot [21] is capable of capturing events using unmodified browsers. Similarly, WaRR [4] and, more recently, Timelapse and Dolos [7] can in principle provide full information about failing executions. Ripley [30] instruments JavaScript applications to enable replaying executions on the server with the purpose of ensuring computational integrity, and DoDOM [27] performs replaying to infer DOM invariants, not to aid debugging of crashes that users encounter. However, we have seen no concrete usage of these tools in real-world debugging. We believe that record/replay tools usually require that the record and replay environments, which include browser configuration, browser state, persistent data (e.g. cookies), network speed, processor speed, and operating system configuration, to be exactly same. Such a requirement is too strict and difficult to reproduce in real-world scenarios where an application can be run by any user under any possible environment. Crowdie does not suffer from such limitations because it does not aim to faithfully replay a buggy execution—it simply tries to collect a relevant portion of the control-flow path using light-weight and targeted instrumentation.

Fault localization techniques. Several automated fault localization techniques [13, 14, 18] rely on statistical data from user executions collected in order to assist developers with debugging. Such information may be useful for debugging, but it does not provide the crash paths that we argue are valuable when debugging. For future work, it will be interesting to perform a direct experimental comparison with such techniques. Liblit et al. [19] interestingly note that "the stack contains essentially no information about the bug's cause" in half of the bugs considered in their experiments, which aligns well with our observation that more information about the failing executions is often needed. The Holmes tool [9] localizes faults using path profiles, which are constructed by iteratively instrumenting and re-deploying programs similar to our technique. Unlike crash paths, path profiles only provide statistical information about intraprocedural and acyclic paths. The AutoFLox tool by Ocariza et al. [25] performs fault localization for JavaScript under the assumption that a complete failing execution is known, unlike our technique that aims to automatically find the crash path. Our algorithm for finding crash paths is also related to algorithmic program debugging [29] but does not require guidance by the programmer or by formal specifications.

Dynamic program slicing. Our notion of crash paths is related to the use of dynamic slicing for debugging [1, 2, 17, 32]. Such techniques typically compute slices backwards, which resembles our construction of crash paths, but usually assuming that a complete execution trace is already known. Although many variations of dynamic slicing have been proposed, they generally differ from our notion of crash path, which comprises a path from the creation of an object of interest to a crashing statement, without including all dependencies earlier in the execution. In principle, slicing may be applied subsequently to the crash path to filter away instructions that are likely irrelevant.

Other crowdsourced analysis techniques. Crowdsourced analysis has also been suggested for other debugging scenarios. For example, Kerschbaumer et al. [16] use crowdsourced analysis to test for information flow vulnerabilities by letting different users track different information flows so that a crowd of users can achieve high coverage of all information flows without imposing unacceptable performance overhead on any single user. Likewise, Kasikci et al. [15] test potential data races by distributing them over a set of users by giving each user an instrumented version of an application where the purpose of the added instrumentation code is to confirm whether the potential race happens in practice. To lower the overhead, instrumentation is enabled probabilistically. In contrast, we enable instrumentation only when a crash occurs, then increase instrumentation until a crash path is discovered, and finally disable the instrumentation.

7. CONCLUSIONS AND FUTURE WORK

Many bugs manifest themselves as objects that have incorrect property values when a failure occurs. For such bugs, error messages and stack traces often provide insufficient information for diagnosing the problem. We have proposed the notion of a crash path, which reflects the control flow from the allocation of a selected object of interest to the crashing statement, and argue that this often provides useful information for debugging such problems.

In principle, crash paths can be computed by executing a fully instrumented version of a program, but this incurs prohibitive runtime overhead. Therefore, we have developed a feedback-directed instrumentation technique for computing crash paths that we envision to be deployed in a crowdsourced scenario where the same failure can be observed repeatedly. We implemented the technique in a tool called Crowdie, and evaluated it in 10 case studies by using it to debug real-world issues reported on GitHub for which error messages and stack traces are insufficient to find and fix the bugs. In these case studies, the feedback-directed technique requires the same crash to be encountered 3 to 10 times, and the runtime overhead generally compares favorably to that of full instrumentation.

Directions for future work include achieving a deeper understanding in which situations the feedback-directed technique is useful, and reducing instrumentation overhead. We also plan to explore the use of more detailed instrumentation once a complete crashing path has been identified in order to prune irrelevant statements from the path.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Softw., Pract. Exper.*, 23(6):589–616, 1993.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[3] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *Proc. ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 17–31, 2014.

[4] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 403–410, 2011.

[5] Apple. Crash reporter. https://developer.apple.com/library/mac/technotes/tn2004/tn2123.html.

[6] J. Bell, N. Sarda, and G. E. Kaiser. Chronicler: lightweight recording to reproduce field failures. In *Proc. 35th International Conference on Software Engineering*, pages 362–371, 2013.

[7] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proc. 26th ACM Symposium on User Interface Software and Technology*, pages 473–484, 2013.

[8] Y. Cao, H. Zhang, and S. Ding. SymCrash: selective recording for reproducing crashes. In *Proc. ACM/IEEE International Conference on Automated Software Engineering*, pages 791–802, 2014.

[9] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani. Holmes: effective statistical debugging via efficient path profiling. In *Proc. 31st International Conference on Software Engineering*, pages 34–44, 2009.

[10] J. A. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *Proc. 29th International Conference on Software Engineering*, pages 261–270, 2007.

[11] L. Gong, M. Pradel, M. Sridharan, and K. Sen. DLint: dynamically checking bad coding practices in JavaScript. In *Proc. International Symposium on Software Testing and Analysis*, pages 94–105, 2015.

[12] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *Proc. 34th International Conference on Software Engineering*, pages 474–484, 2012.

[13] W. Jin and A. Orso. F3: fault localization for field failures. In *Proc. International Symposium on Software Testing and Analysis*, pages 213–223, 2013.

[14] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, 2005.

[15] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: crowdsourced data race detection. In *Proc. ACM SIGOPS 24th Symposium on Operating Systems Principles*, pages 406–422, 2013.

[16] C. Kerschbaumer, E. Hannigan, P. Larsen, S. Brunthaler, and M. Franz. CrowdFlow: Efficient information flow security. In *Proc. 16th Information Flow Security Conference*, 2013.

[17] B. Korel and J. W. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[18] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 141–154, 2003.

[19] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 15–26, 2005.

[20] M. Madsen, F. Tip, E. Andreasen, K. Sen, and A. Møller. Feedback-directed instrumentation for deployed JavaScript applications. Technical Report CS-2015-16, School of Computer Science, University of Waterloo, 2015. https://cs.uwaterloo.ca/sites/ca.computer-science/files/uploads/files/CS-2015-16.pdf.

[21] J. W. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for JavaScript applications. In *Proc. 7th USENIX Symposium on Networked Systems Design and Implementation*, pages 159–174, 2010.

[22] Microsoft. Windows error reporting. https://msdn.microsoft.com/en-us/library/windows/hardware/dn641144.aspx.

[23] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *Proc. 32st International Symposium on Computer Architecture*, pages 284–295, 2005.

[24] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[25] F. Ocariza, G. Li, K. Pattabiraman, and A. Mesbah. Automatic fault localization for client-side JavaScript. *Software Testing, Verification and Reliability*, 2015.

[26] F. S. Ocariza Jr., K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. In *Proc. ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 55–64, 2013.

[27] K. Pattabiraman and B. G. Zorn. DoDOM: Leveraging DOM invariants for web 2.0 application robustness testing. In *Proc. IEEE 21st International Symposium on Software Reliability Engineering*, pages 191–200, 2010.

[28] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proc. European Software Engineering Conference / ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 488–498, 2013.

[29] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, MA, USA, 1983.

[30] K. Vikram, A. Prateek, and V. B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proc. ACM Conference on Computer and Communications Security*, pages 173–186, 2009.

[31] S. H. Yong and S. Horwitz. Using static analysis to reduce dynamic analysis overhead. *Formal Methods in System Design*, 27(3):313–334, 2005.

[32] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proc. 25th International Conference on Software Engineering*, pages 319–329, 2003.

APPENDIX

A. CASE STUDY

Here we report on our experience debugging the bug reports not described in Section 2.

A.1 AlaSQL Issue #0092

In this issue entitled “Error: Something wrong with index on table” reported for AlaSQL (v0.0.36) the developer reports that an exception is thrown[8] after the creation of several tables, insertion of multiple rows and multiple queries:

```
Error: Something wrong with index on table
  at table.delete (alasql.js:8311:27)
  at statement (alasql:9262:41)
  at Function.alasql.drun (alasql:3369:43)
  at Function.alasql.dexec (alasql.js:3353:31)
  at Function.alasql.exec (alasql.js:3296:27)
  at alasql (alasql.js:106:31)
```

The error message hints that an index is corrupted. Inspecting the stack trace indicates that the error occurs when rows are deleted from a table. The (simplified) source code of the `table.delete` function is:

```
table.delete = function(i) {
  // ...
  var r = this.data[i];
  var pk = this.pk;
  var addr = pk.onrightfn(r);
  if (typeof this.indices[pk.hh][addr]
          == 'undefined') {
    throw new Error('Something␣wrong␣'
                 + 'with␣index␣on␣table');
  } else {
     this.indices[pk.hh][addr] = undefined;
  }
}
```

We see that the exception is thrown in response to `this.indices[pk.hh][addr]` being `undefined`. A closer look at the code reveals that there are three values that may be incorrect: `this.indices`, `pk.hh` and `addr` (which originates from `this.data`). Since the lookup `this.indices[pk.hh]` returned some object, i.e., it didn’t throw an `undefined` dereference exception, the value of `addr` seems most suspicious.

Looking further down the stack trace does not reveal any new information: We cannot determine the location of the fault from the error message and stack trace alone. Applying Crowdie to find the crash path for each of these values gives several paths for the developer to inspect. Looking at the crash path for `this.data` reveals something interesting:

```
... = function(db, params, alasql) {
  var res = selectfn(params);
  db.tables[tableid].data =
    db.tables[tableid].data.concat(res);
  return res.length;
}
```

which is a function used during a `SELECT` sub query, as revealed by the crash path. Note that the value of the `data` property is overwritten directly with the result of an array concatenation operation. This is problematic, since this only updates the array without updating the index accordingly. Instead, the `table.insert` function should have been used, which is responsible for keeping the index up-to-date. The fix is simply to use this function instead of directly mutating the `data` property.

In this case study the error message and stack trace were insufficient the debug the issue. Crowdie helped find the origin of several values of interest and revealed that the crash was related to sub-queries (which was not knowable from the stack trace). In the debugging scenario the developer still had to use some ingenuity: First, to determine the value(s) of interest. And, second, to know that the index is maintained by the `table.insert` function and that the `data` property should not be updated directly.

A.2 Buckets Issue #0005

In this issue, entitled “Various Dictionary bugs”, appearing in Buckets v1.82, the reporting user explains that a Dictionary implementation contains several bugs related to JavaScript semantics that the implementer did not take into account[9]. One of the bugs is exposed with the following code, extracted from the bug report:

```
1 var dict = new buckets.Dictionary();
2 var obj = {};
3 dict.set("hasOwnProperty", obj);
4 dict.keys();
```

The last call makes the library throw an exception: “TypeError: object is not a function”, with the stack trace:

```
TypeError: object is not a function
    at buckets.Dictionary.keys (buckets.js:796)
    at Object.<anonymous> (example.js:4)
```

The stack trace only reveals that a call at line 796 could not be performed. Looking at that line:

```
796 if (this.table.hasOwnProperty(name)) {
```

We see that the property `hasOwnProperty` was supposed to be a function, but according to the exception, it is not.

An experienced JavaScript programmer knows that `hasOwnProperty` is a built-in method in the Object prototype-chain, so the natural question is to find out what has overridden it. So we instruct Crowdie to find the origin of `this.table.hasOwnProperty`, Crowdie uses four iterations to discover the following path:

```
Start(ex.js:2)           var obj = {};
Call(ex.js:3)            dict.set("hasOwnProperty", obj);
Enter(buckets.js:751)    ...set = function(key, value) {
...
Exit(buckets.js:770)     return ret;
Return(ex.js:3))         dict.set("hasOwnProperty", obj);
Call(ex.js:4)            dict.keys();
Enter(buckets.js:793)    ...keys = function() {
Crash(buckets.js:796)    if (this.table.
                            hasOwnProperty(name)) {
```

From this path, it can be seen that the origin of the object is the `obj`-object on line 2 in the above example code. Crowdie also tells us that this object is used in `dict.set("hasOwnProperty", obj)` on the very next line, from which it is clear that the user-selected dictionary key, “hasOwnProperty”, breaks the dictionary implementation by creating a local property `hasOwnProperty` in the internal dictionary datastructure that shadows the `hasOwnProperty` property of a prototype object.

On line 796, Buckets relies on `hasOwnProperty` being the native implementation of `hasOwnProperty`, but the call on line 3 ends up shadowing that property with the object allocated at line 2. The fix for this issue is to remove the dependency in Buckets on `hasOwnProperty` being unmodi-

[8] https://github.com/agershun/alasql/issues/92

[9] https://github.com/mauriciosantos/Buckets-JS/issues/6

fied by user-selected values. The following code illustrates the principle behind the fix for line 796:

```
796 // if (this.table.hasOwnProperty(name)) {
797 var HOP = Object.prototype.hasOwnProperty;
798 if (HOP.call(this.table, name)) {
```

In this case study, we needed to know why a property had been overridden, and the stack trace could not tell us that. The crash path produced by Crowdie helped with finding the cause of the overriding problem. In particular, the crash path revealed that shortly after the object of interest was allocated, it was used inside Buckets to override the property of interest – this revealed that internal dependencies of Buckets needed to be rethought[10].

Although the bug report appears to originate from a code review, we believe the bug could just as well have occurred in the wild.

A.3 Esprima Issue #0560

In this issue entitled "LeftHandSideExpression in ForInStatement not parsing when it contains an 'in' operator", reported for Esprima (v1.2), the user states that Esprima crashes on the input "for (a[b in c] in d);" which he or she expected to parse[11], [12].

Esprima crashes with the following error message and stack trace[13]:

```
Error: Line 1: Unexpected token in
    at throwError
    at throwUnexpected
    at expect
    at parseComputedMember
    at parseLeftHandSideExpressionAllowCall
    at parsePostfixExpression
    at parseUnaryExpression
    at parseBinaryExpression
    at parseConditionalExpression
    at parseAssignmentExpression
    at parseExpression
    at parseForStatement
    at parseStatement
    at parseSourceElement
    at parseSourceElements
    at parseProgram
    at Object.parse
    at Object.<anonymous>
```

The error message says that the token "in" was unexpected. This may lead the developer to believe that the token was somehow corrupted or that the token was accidently replaced by another token. The stack trace itself does not indicate whether or not this is the case. Using Crowdie we discover that the token was created inside scanIdentifier and was not changed along the crash path. While this information is useful, it does not, by itself, help us to locate the bug: The bug cannot be isolated with the error message, stack trace and Crowdie alone. Looking at the fix on GitHub we can see that fix involves intricate updates of a global variable state.allowIn inside the functions parseLeftHandSideExpressionAllowCall, parseLeftHandSideExpression and parseForStatement. Coming up with these changes seems to require a deep knowledge of the internals of Esprima.

Esprima issues #0299[14] and #1042[15] are similar to the above: We can use Crowdie to rule out certain scenarios, but such information by itself is not necessarily sufficient to locate the bug. Nevertheless, we believe that even in such scenarios, Crowdie provides developers with a useful new tool for debugging.

A.4 Immutable Issue #0489

In this issue, entitled "'deleteIn' fails for Set within Map" for Immutable v3.7.3 the user reports that a sequence of operations on a collection object cause an exception[16]. The user provides a code fragment:

```
new Map()
  .set("decks", new Set([1, 12, 2]))
  .deleteIn(["decks", 12])
```

which produces the error message and stack trace:

```
Error: invalid keyPath
  at invariant (immutable.js:932:31)
  at updateInDeepMap (immu.js:2509:9)
  at updateInDeepMap (immu.js:2515:27)
  at src_Map__Map.updateIn (immu.js:1816:28)
  at src_Map__Map.deleteIn (immu.js:1802:21)
```

This somewhat cryptic error message does not provide much information, at least for someone who is unfamiliar with the Immutable library. The stack trace tell us that the exception is thrown from:

```
function invariant(condition, error) {
  if (!condition) throw new Error(error);
}
```

which is a generic error construction function. This function is called from updateInDeepMap where the real error occurs[17]:

```
2495 function updateInDeepMap(existing, ...) {
2496   // ...
2497   invariant( /* ... */
2498             || (existing && existing.set),
2499             'invalid␣keyPath');
2500   // ...
2501 );
```

From the error message and stack trace alone it is not clear why the invariant is violated. Using Crowdie we discover that the invariant is violated when the existing variable points to an object without a set property [18]. As in previous case studies, it is not clear why this property is missing: Was it never set? Was it overwritten? Does existing point to the wrong object? Inspecting the crash path we discover that the object is created by the makeSet function:

```
function makeSet(map, ownerID) {
  var set = Object.create(SetPrototype);
  // ...
  return set;
}
```

The call to Object.create(SetPrototype) creates a new object whose internal prototype is the SetPrototype object. This is a global object which holds all functionality com-

[10] https://github.com/mauriciosantos/Buckets-JS/commit/f58f2f0a3edcb9041fb5d26afec7bc57b7f21de2

[11] https://code.google.com/p/esprima/issues/detail?id=560

[12] https://github.com/jquery/esprima/pull/267

[13] Here line numbers have been omitted for brevity.

[14] https://github.com/jquery/esprima/pull/299

[15] https://github.com/jquery/esprima/issues/1042

[16] https://github.com/facebook/immutable-js/issues/489

[17] Here the line number 2509 reported in the stack trace corresponds to line 2497 in the example, due to the elision of some irrelevant code.

[18] Here set should be understood as "assign" or "update", and not as the notion of a mathematical set.

monly available on sets. Manual inspection of the crash path shows that the `set` property is never set nor overwritten. Furthermore, the prototype object does not contain any `set` property.

Pondering this, we realize that `updateInDeepMap` implicitly assumes that it is only passed map objects (which have a `set` property) and not sets (which lack such a property). We can fix this issue by adding a `set` property to the prototype `SetProperty`.

In this issue CROWDIE was useful for finding the origin of the `existing` object and determining that the `set` property was not modified along the crash path, but was missing from the time the object was created (due to its type).

A.5 Loki Issue #0042

In this issue, entitled "when not found findOne returns Array[0] instead of null" for Loki (v1.0) the developer reports that the `findOne` function should return `null` instead of a zero-sized array when the result set is empty[19]. As stated earlier, Loki is a JavaScript database and the `findOne` function is used to retrieve the first row from a given query.

In the bug report, the developer does not provide any code fragment, presumably because it is very easy to reproduce. For the sake of exposition, we will assume that the developer encountered the problem while trying to run the code:

```
1 var db = new loki(/* ... */);
2 var users = db.addCollection(* ... */);
3 var result = users.findOne(* ... */);
4 assert(result == null);
```

which crashes with the assertion failure:

```
5 AssertionError: false == true
6     at main.js:4:11
```

Here (and in general) the error message and stack traces do not contain information about where the object came from, since the object was created inside the library and stack has since been collapsed. Using CROWDIE to discover the origin of `result` reveals that the `Resultset.prototype.find` function returns an empty array:

```
607 Resultset.prototype.find = function (...) {
608   if (this.collection.data.length === 0) {
609     // ..
610     return [];
611   }
612   // ...
613 }
```

This function, as revealed by the crash path, is called from:

```
266 function Resultset(...) {
267   // ...
268   if (typeof (queryObj) !== "undefined"
269       && queryObj !== null) {
270     return this.find(queryObj, firstOnly);
271   }
272   // ...
273 }
```

which according to the crash path is called from:

```
2469 ... findOne = function (...) {
2470   // ...
2471   return new Resultset(...);
2472 };
```

Note that although `Resultset` is invoked as a constructor (using `new` ...) on line 2471 its result is thrown away by the statement `return this.find(...)` on line 270 inside `Resultset`.

The issue is fixed by checking the size of the result set inside either `Resultset.prototype.find` or directly in the `Resultset` constructor. Debugging this issue with CROWDIE was easy since the tool readily identified from where the object came and the relevant functions on the crash path.

A.6 Redis Issue #0067

In this issue, entitled "hgetall returns when no results", reported for Loki v1.0, the user reports that "Testing for if(results) ends up with a false positive."[20]. The problem is that the library breaks the JavaScript idiom of returning a value that coerces to false in case of a query without any results. The returned value is an empty object instead of a value like `null` or `undefined`. In this case, the user did not include a code snippet with the bug report that triggers the problem. However, from the description of the issue, it was easy for us to reconstruct what such a code example would look like. In the discussion below, we will assume that the user encountered the reported problem in the context of the following example.

```
1 var client = redis.createClient();
2 client.hgetall("does␣not␣exist", function (
      err, results) {
3     if(results){
4         throw new Error("Redis-bug␣0067:␣
              non-null␣results");
5     }
6 });
```

The following stack trace is produced by running the code:

```
Error: Redis-bug 0067: non-null result
 at example.js:4
 at try_callback (redis.js:980)
 at RedisClient.return_reply (redis.js:1024)
 at RedisReplyParser.<anonymous> (redis.js:773)
 at RedisReplyParser.emit (events.js:107)
 at RedisReplyParser.send_reply (redis.js:502)
 at RedisReplyParser.execute (redis.js:387)
 at RedisClient.on_data (redis.js:936)
 at Socket.<anonymous> (redis.js:580)
 at Socket.emit (events.js:107)
 at readableAddChunk (_stream_readable.js:163)
 at Socket.Readable.push (_stream_readable.js:126)
 at TCP.onread (net.js:538)
```

The stack trace reveals that asynchronous JavaScript is being used because the root of the stack trace is not in the example code. This means that single-stepping the code with a debugger would be harder than usual.

Nevertheless, CROWDIE can find the origin of `results` in three iterations, at line 1021 of redis.js. In the allocating function it is seen that an empty `results` object is always allocated, and that each individual query result from the `externals` array is added to it later. The following code illustrates the principle:

```
var results = {};
for(var i = 0; i < externals.length; i++){
  var result = externals[i];
  results[result.name] = result.value;
}
return result;
```

The fix[21] for the issue is an early return of `null` if no

[19] https://github.com/techfort/LokiJS/issues/42

[20] https://github.com/NodeRedis/node_redis/issues/67

[21] https://github.com/NodeRedis/node_redis/commit/69092a3f267b67c8067c19f98ab08bad2f936aad

query results are found. This fix is applied where the empty `results` object was allocated, i.e., at the root of the crash path.

The following code illustrates the principle behind the fix:

```
var results = {};
if(externals.length === 0){
  return null;
}
for(var i = 0; i < externals.length; i++){
  var result = externals[i];
  results[result.name] = result.value;
}
return result;
```

In this case study, the source location of interest is actually *close* to one of the lines mentioned in the long stack trace, but Crowdie pinpoints exactly where the object was allocated – saving the bug fixer from having to look through a long stack trace for the right location to apply a fix.