BLEAK: Automatically Debugging Memory Leaks in Web Applications

John Vilk
University of Massachusetts Amherst, USA
jvilk@cs.umass.edu

Emery D. Berger
University of Massachusetts Amherst, USA
emery@cs.umass.edu

Abstract

Despite the presence of garbage collection in managed languages like JavaScript, memory leaks remain a serious problem. In the context of web applications, these leaks are especially pervasive and difficult to debug. Web application memory leaks can take many forms, including failing to dispose of unneeded event listeners, repeatedly injecting iframes and CSS files, and failing to call cleanup routines in third-party libraries. Leaks degrade responsiveness by increasing GC frequency and overhead, and can even lead to browser tab crashes by exhausting available memory. Because previous leak detection approaches designed for conventional C, C++ or Java applications are ineffective in the browser environment, tracking down leaks currently requires intensive manual effort by web developers.

This paper introduces BLEAK (**B**rowser **Leak** debugger), the first system for automatically debugging memory leaks in web applications. BLEAK's algorithms leverage the observation that in modern web applications, users often repeatedly return to the same (approximate) visual state (e.g., the inbox view in Gmail). Sustained growth between round trips is a strong indicator of a memory leak. To use BLEAK, a developer writes a short script (17–73 LOC on our benchmarks) to drive a web application in round trips to the same visual state. BLEAK then automatically generates a list of leaks found along with their root causes, ranked by return on investment. Guided by BLEAK, we identify and fix over 50 memory leaks in popular libraries and apps including Airbnb, AngularJS, Google Analytics, Google Maps SDK, and jQuery. BLEAK's median precision is 100%; fixing the leaks it identifies reduces heap growth by an average of 94%, saving from 0.5 MB to 8 MB per round trip. We believe BLEAK's approach to be broadly applicable beyond web applications, including to GUI applications on desktop and mobile platforms.

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06...$15.00
https://doi.org/10.1145/3192366.3192376

CCS Concepts • **Software and its engineering → Software testing and debugging**;

Keywords Memory leaks, debugging, leak detection, web development, JavaScript

ACM Reference Format:
John Vilk and Emery D. Berger. 2018. BLEAK: Automatically Debugging Memory Leaks in Web Applications. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3192366.3192376

1 Introduction

Browsers are one of the most popular applications on both smartphones and desktop platforms [3, 53]. They also have an established reputation for consuming significant amounts of memory [26, 38, 43]. To address this problem, browser vendors have spent considerable effort on shrinking their browsers' memory footprints [13, 23, 39, 47, 75] and building diagnostic tools that track the memory consumption of specific browser components [21, 42].

Memory leaks in web applications only exacerbate the situation by further increasing browser memory footprints. These leaks happen when the application references unneeded state, preventing the garbage collector from collecting it. Web application memory leaks can take many forms, including failing to dispose of unneeded event listeners, repeatedly injecting iframes and CSS files, and failing to call cleanup routines in third-party libraries. Leaks are a serious concern for developers since they lead to higher garbage collection frequency and overhead. They reduce application responsiveness and can even trigger browser tab crashes by exhausting available memory [6, 24, 31, 40, 49].

Despite the fact that memory leaks in web applications are a well-known and pervasive problem, there are no effective automated tools that can find them. The reason is that existing memory leak detection techniques are ineffective in the browser: *leaks in web applications are fundamentally different from leaks in traditional C, C++, and Java programs.* Staleness-based techniques assume leaked memory is rarely touched [8, 25, 48, 52, 74], but web applications regularly interact with leaked state (e.g., via event listeners). Growth-based techniques assume that leaked objects are uniquely owned or that leaked objects form strongly connected components in the heap graph [41, 74]. In web applications, leaked objects frequently have multiple owners, and the entire heap graph is often strongly connected due to widespread references to the global scope (`window`). Finally, techniques that depend on static type information [28] do not work for web applications because JavaScript is dynamically typed.

Faced with this lack of automated tool support, developers are currently forced to manually inspect heap snapshots to locate objects that the application incorrectly retains [6, 31, 40, 49]. Unfortunately, these snapshots do not necessarily provide actionable information (see §2.2). They simultaneously provide too much information (every single object on the heap) and not enough information to actually debug these leaks (no connection to the code responsible for leaks). Since JavaScript is dynamically typed, most objects in snapshots are simply labeled as Objects or Arrays, which provides little assistance in locating the source of leaks. The result is that even expert developers are unable to find leaks: for example, a Google developer closed a Google Maps SDK memory leak (with 99 stars and 51 comments) because it was "infeasible" to fix as they were "not really sure in how many places [it's] leaking" [16].

We address these challenges with BLeak (**B**rowser **Leak** debugger), the first system for automatically debugging memory leaks in web applications. BLeak leverages the following fact: over a single session, users repeatedly return to the same visual state in modern web sites, such as Facebook, Airbnb, and Gmail. For example, Facebook users repeatedly return to the news feed, Airbnb users repeatedly return to the page listing all properties in a given area, and Gmail users repeatedly return to the inbox view.

We observe that *these round trips can be viewed as an oracle to identify leaks*. Because visits to the same (approximate) visual state should consume roughly the same amount of memory, sustained memory growth between visits is a strong indicator of a memory leak. BLeak builds directly on this observation to find memory leaks in web applications, which (as §6 shows) are both widespread and severe.

To use BLeak, a developer provides a short script (17–73 LOC on our benchmarks) to drive a web application in a loop that takes round trips through a specific visual state. BLeak then proceeds automatically, identifying memory leaks, ranking them, and locating their root cause in the source code. BLeak first uses heap differencing to locate locations in the heap with sustained growth between each round trip, which it identifies as leak roots. To directly identify the root causes of growth, BLeak employs JavaScript rewriting to target leak roots and collect stack traces when they grow. Finally, when presenting the results to the developer, BLeak ranks leak roots by return on investment using a novel metric called LeakShare that prioritizes memory leaks that free the most memory with the least effort by dividing the "credit" for retaining a shared leaked object equally among the leak roots that retain them. This ranking focuses developer effort on the most important memory leaks first.

```
1  class Preview extends PureComponent {
2    // Runs when Preview is added to GUI
3    componentDidMount() {
4      const { codeMirror } = this.props.editor;
5      const wrapper = codeMirror.getWrapperElement();
6      codeMirror.on("scroll", this.onScroll);
7      wrapper.addEventListener("mouseover", this._mover);
8      wrapper.addEventListener("mouseup", this._mup);
9      wrapper.addEventListener("mousedown", this._mdown);
10   }
11 }
```

Figure 1. This code from Firefox's debugger (truncated for readability) leaks 0.5MB every time a developer opens a source file (§2). Leak detectors that rely on staleness metrics would fail to find these leaks because the leaked objects (event listeners) are frequently touched. BLeak finds all these leaks automatically.

Guided by BLeak, we identify and fix over 50 memory leaks in popular JavaScript libraries and applications including Airbnb, AngularJS, jQuery, Google Analytics, and the Google Maps SDK. BLeak has a median precision of 100% (97% on average). Its precise identification of root causes of leaks makes it relatively straightforward for us to fix nearly all of the leaks we identify (all but one). Fixing these leaks reduces heap growth by 94% on average, saving from 0.5 MB to 8 MB per return trip to the same visual state. We have submitted patches for all of these leaks to the application developers; at the time of writing, 16 have already been accepted and 4 are in the process of code review.

Contributions

This paper makes the following contributions:

- It introduces novel techniques for automatically locating, diagnosing, and ranking memory leaks in web applications (§3), and presents algorithms for each (§4).
- It presents BLeak, an implementation of these techniques. BLeak's analyses drive websites using Chrome and a proxy that transparently rewrites JavaScript code to diagnose leaks, letting it operate on unmodified websites (including over HTTPS) (§5).
- Using BLeak, we identify and fix numerous memory leaks in widely used web applications and JavaScript libraries (§6).

2 Background

Before presenting BLeak and its algorithms, we first describe a representative memory leak we discovered using BLeak (see Figure 1), and discuss why prior techniques and existing tools fall short when debugging leaks in web applications.

This memory leak is in Firefox's debugger, which is a pure HTML5 application that runs as a normal web application in all browsers. Lines 6-9 register four event listeners on

Summary ▼ Class filter Objects allocated between Snapshot 1 and Snap

Constructor	Distance	Objects Count		Shallow Size		Retained Size	
▶ Array	4	3 143	0 %	100 576	0 %	31 099 584	26 %
▶ (array)	4	6 190	0 %	24 387 568	20 %	24 497 176	21 %
▶ BranchChunk	5	592	0 %	33 152	0 %	7 496 720	6 %
▶ LeafChunk	5	2 382	0 %	114 336	0 %	7 385 168	6 %
▶ Line	4	59 549	4 %	4 287 528	4 %	6 717 288	6 %
▶ (string)	5	3 823	0 %	4 761 800	4 %	4 761 800	4 %
▶ (sliced string)	5	55 931	4 %	2 237 240	2 %	2 237 240	2 %
▶ Doc	3	1	0 %	200	0 %	1 965 840	2 %
▶ Preview	6	1	0 %	200	0 %	489 016	0 %

(a) A truncated heap snapshot of the Firefox debugger, filtered using the three snapshot technique. The only relevant item is `Preview`, which appears low on the list underneath non-leaking objects.

Constructor	Distance	Objects Count		Shallow Size		Retained Size	
▼ Preview	6	1	0 %	200	0 %	489 016	0 %
▶ Preview @40(	6			200	0 %	489 016	0 %

Retainers

Object	Distance	Shallow Size		Retained Size	
▼ bound_this in *native_bind()* @4001	5	48	0 %	48	0 %
▶ [0] in Array @700847	4	32	0 %	64	0 %
▶ 0 in (object elements)[] @285972	5	32	0 %	32	0 %
▶ onScroll in Preview @400119	6	200	0 %	489 016	0 %
▼ this in system / Context @397635	5	56	0 %	56	0 %
▼ context in *()* @387667	4	72	0 %	160	0 %
▼ native in HTMLDivElement @362	3	40	0 %	400	0 %
▼ [97] in Document DOM tree /	2	0	0 %	0	0 %
1 in (Document DOM trees)	1	0	0 %	0	0 %

(b) The retaining paths for `Preview`, the primary leaking object in the Firefox debugger. Finding the code responsible for leaking this object involves searching the entire production code base for identifiers in the retaining paths, which are commonly managed by third-party libraries and obfuscated via minification.

Figure 2. The manual memory leak debugging process: Currently, developers debug leaks by first examining heap snapshots to find leaking objects (Figure 2a). Then, they try to use retaining paths to locate the code responsible (Figure 2b). Unfortunately, these paths have no connection to code, so developers must search their codebase for identifiers referenced in the paths (see §2.2). This process can be time consuming and ultimately fruitless. BLeak saves considerable developer effort by automatically detecting and locating the code responsible for memory leaks.

the debugger's text editor (`codeMirror`) and its GUI object (`wrapper`) every time the user views a source file. The leak occurs because the code fails to remove the listeners when the view is closed. Each event listener leaks `this`, which points to an instance of `Preview`.

2.1 Prior Automated Techniques

There currently are no effective automated techniques for finding memory leaks in web applications. Previous effective automated techniques for finding memory leaks operate in the context of conventional applications written in C, C++, and Java. These techniques predominantly use a staleness metric to discover [8, 25, 48] or rank [74] memory leaks, but the four memory leaks in the Firefox debugger would not be considered stale. These four listeners continue to execute and touch leaked state every time the user uses the mouse on the editor, marking that state as "fresh". In web applications, many leaks are connected to browser events: 77% of the memory leaks found by BLeak would not be found by a staleness-based approach (§6.5). For this reason, BLeak focuses on object growth rather than staleness.

Prior growth-based techniques assume that leaked objects are uniquely owned (*dominated*) by a single object or that they form strongly connected components in the heap [41, 74]. These assumptions do not hold for the leaked objects in the Firefox debugger because 1) they are owned by four separate leak locations that are only dominated by the global scope, and 2) they reference the global scope (`window`) and are thus strongly connected with nearly the entire heap. Because these properties are common in web applications, BLeak does not use ownership information to identify leaks.

2.2 Manual Leak Debugging via Heap Snapshots

Since there are currently no automated techniques for identifying memory leaks in web applications, developers are forced to use manual approaches. The current state of the art is manual processing of heap snapshots. As we show, this approach does not effectively identify leaking objects or provide useful diagnostic information, and it thus does little to help developers locate and fix memory leaks.

The most popular way to manually debug memory leaks is via the *three heap snapshot technique* introduced by the GMail team [31]. Developers repeat a task twice on a webpage and examine still-live objects created from the first run of the task. The assumption is that each run will clear out most of the objects created from the previous run and leave behind only leaking objects; in practice, it does not.

To apply this technique to Firefox's debugger, the developer takes a heap snapshot after loading the debugger, a second snapshot after opening a source file, and a third snapshot after closing and re-opening a source file. Then, the developer filters the third heap snapshot to focus only on objects allocated between the first and second.

This filtered view, shown in Figure 2a, does not clearly identify a memory leak. Most of these objects are simply reused from the previous execution of the task and are not actually leaks, but developers must manually inspect these objects before they can come to that conclusion. The top item, `Array`, conflates all arrays in the application under one

```
exports.loop = [
// Loop that repeatedly opens and closes a source document.
// First, open a source document in the text editor.
{
  check: function() {
    const nodes =  $('.node');
    // No documents are open
    return $('.source-tab').length === 0 &&
      // Target document appears in doc list
      nodes.length > 1 && nodes[1].innerText === "main.js";
  },
  next: function() { $('.node')[1].click(); }
},
// Next, close the document after it loads.
{
  check: function() {
    // Contents of main.js are in editor
    return $('.CodeMirror-line').length > 2 &&
      // Editor displays a tab for main.js
      $('.source-tab').length === 1 &&
      // Tab contains a close button
      $('.close-btn').length === 1;
  },
  next: function() { $('.close-btn').click(); }
}];
```

(a) This script runs the Firefox debugger in a loop, and is the only input BLeak requires to automatically locate memory leaks. For brevity, we modify the script to use jQuery syntax.

```
# Leak Root 1 [LeakShare: 811920]

## Leak Paths

* Event listeners for 'mouseover' on window.cm.display.wrapper

## Stack Traces Responsible

1. Preview.componentDidMount
   http://localhost:8000/assets/build/debugger.js:109352:22
2. http://localhost:8000/assets/build/debugger.js:81721:24
3. measureLifeCyclePerf
   http://localhost:8000/assets/build/debugger.js:81531:11
4. http://localhost:8000/assets/build/debugger.js:81720:31
5. CallbackQueue.notifyAll
   http://localhost:8000/assets/build/debugger.js:61800:21
6. ReactReconcileTransaction.close
   http://localhost:8000/assets/build/debugger.js:83305:25
7. ReactReconcileTransaction.closeAll
   http://localhost:8000/assets/build/debugger.js:42268:24
```

(b) A snippet from BLeak's memory leak report for the Firefox debugger. BLeak points directly to the code in Figure 1 responsible for the memory leak.

Figure 3. Automatic memory leak debugging with BLeak: The only input developers need to provide to BLeak is a simple script that drives the target web application in a loop (Figure 3a). BLeak then runs automatically, producing a ranked list of memory leaks with stack traces pointing to the code responsible for the leaks (Figure 3b).

heading because JavaScript is dynamically typed. Confusingly, the entry `(array)` just below it refers to internal V8 arrays, which are not under the application's direct control. Developers would be unlikely to suspect the `Preview` object, the primary leak, because it both appears low on the list and has a small retained size.

Even if a developer identifies a leaking object in a snapshot, it remains challenging to diagnose and fix because the snapshot contains no relation to code. The snapshot only provides retaining paths in the heap, which are often controlled by a third party library or the browser itself. As Figure 2b shows, the retaining paths for a leaking `Preview` object stem from an array and an unidentified DOM object. Locating the code responsible for a leak using these retaining paths involves grepping through the code for instances of the identifiers along the path. This task is often further complicated by two factors: (1) the presence of third-party libraries, which must be manually inspected; and (2) the common use of minification, which effectively obfuscates code and heap paths by reducing most variable names and some object properties to single letters.

Summary: To debug memory leaks, developers currently must manually sift through large heap snapshots and all JavaScript code on a page. Snapshots conflate many JavaScript objects as either arrays or Objects, and existing ranking techniques can incorrectly place severe leaks below non-leaks. Since snapshots only contain retaining paths, programmers must manually determine program points by searching code for promising identifiers in the retained paths, but these are commonly obfuscated by minifiers. These challenges combine to make manual leak debugging a daunting task.

3 BLeak Overview

This section presents an overview of the techniques BLeak uses to automatically detect, rank, and diagnose memory leaks. We illustrate these by showing how to use BLeak to debug the Firefox memory leak presented in Section 2.

Input script: Developers provide BLeak with a simple script that drives a web application in a loop through specific visual states. A *visual state* is the resting state of the GUI after the user takes an action, such as clicking on a link or submitting a form. The developer specifies the loop as an array of objects, where each object represents a specific visual state, comprising (1) a `check` function that checks the preconditions for being in that state, and (2) a transition function `next` that interacts with the page to navigate to the next visual state in the loop. The final visual state in the loop array transitions back to the first, forming a loop.

Figure 3a presents a loop for the Firefox debugger that opens and closes a source file in the debugger's text editor. The first visual state occurs when there are no tabs open in the editor (line 8), and the application has loaded the list of documents in the application it is debugging (line 10); this is the default state of the debugger when it first loads. Once the application is in that first visual state, the loop transitions the application to the second visual state by clicking on `main.js` in the list of documents to open it in the text editor (line 12). The application reaches the second visible state once the debugger displays the contents of `main.js` (line 18). The loop then closes the tab containing `main.js` (line 24), transitioning back to the first visual state.

Locating leaks: From this point, BLeak proceeds entirely automatically. BLeak uses the developer-provided script to drive the web application in a loop. Because object instances can change from snapshot to snapshot, BLeak tracks *paths* instead of objects, letting it spot leaks even when a variable or object property is regularly updated with a new and larger object. For example, `history = history.concat(newItems)` overwrites `history` with a new and larger array.

After each visit to the first visual state in the loop, BLeak takes a heap snapshot and tracks specific paths from GC roots that are continually growing. BLeak treats a path as growing if the object identified by that path gains more outgoing references (e.g., when an array expands or when properties are added to an object).

For the Firefox debugger, BLeak notices four heap paths that are growing each round trip: (1) an array within the `codeMirror` object that contains `scroll` event listeners, and internal browser event listener lists for (2) `mouseover`, (3) `mouseup`, and (4) `mousedown` events on the DOM element containing the text editor. Since these objects continue to grow over multiple loop iterations (the default setting is eight), BLeak marks these items as *leak roots* as they appear to be growing without bound.

Ranking leaks: BLeak uses the final heap snapshot and the list of leak roots to rank leaks by return on investment using a novel but intuitive metric we call *LeakShare* (§4.3) that prioritizes memory leaks that free the most memory with the least effort. LeakShare prunes objects in the graph reachable by non-leak roots, and then splits the credit for remaining objects equally among the leak roots that retain them. Unlike retained size (a standard metric used by all existing heap snapshot tools), which only considers objects *uniquely owned* by leak roots, LeakShare correctly distributes the credit for the leaked `Preview` objects among the four different leak roots since they *all* must be removed to eliminate the leak.

Diagnosing leaks: BLeak next reloads the application and uses its proxy to transparently rewrite all of the JavaScript on the page, exposing otherwise-hidden edges in the heap as object properties. BLeak uses JavaScript reflection to instrument identified leak roots to capture stack traces *when they grow* and *when they are overwritten* (not just where they were allocated). With this instrumentation in place, BLeak uses the developer-provided script to run one final iteration of the loop to collect stack traces. These stack traces directly zero in on the code responsible for leak growth.

Output: Finally, BLeak outputs its diagnostic report: a ranked list of leak roots (ordered by LeakShare), together with the heap paths that retain them and stack traces responsible for their growth. Figure 3b displays a snippet from BLeak's output for the Firefox debugger, which points directly to the code responsible for the memory leak from Figure 1.

Summary: Using BLeak, the only developer effort required is creating a short script to drive the web application in a loop. BLeak then locates memory leaks and provides detailed information pointing to the source code responsible. With this information in hand, we were able to discover four new memory leaks in the Firefox debugger, and quickly develop a fix that removes the event listeners when the user closes the document. This fix has been incorporated in the latest version of the debugger.

4 Algorithms

This section formally describes the operation of BLeak's core algorithms for detecting (§4.1), diagnosing (§4.2), and ranking leaks (§4.3).

4.1 Memory Leak Detection

The input to BLeak's memory leak detection algorithm is a set of heap snapshots collected during the same visual state, and the output is a set of *paths* from GC roots that are growing across all snapshots. We call these paths *leak roots*. BLeak considers a path to be *growing* if the object at that path has more outgoing references than it did in the previous snapshot. To make the algorithm tractable, BLeak only considers the shortest path to each specific heap item.

Each heap snapshot contains a heap graph $G = (N, E)$ with a set of nodes N that represent items in the heap, and edges E where each edge $(n_1, n_2, l) \in E$ represents a reference from node n_1 to n_2 with label l. A label l is a tuple containing the type and name of the edge. Each edge's type is either a *closure variable* or an *object property*. An edge's name corresponds to the name of the closure variable or object property. For example, the object `O = { foo: 3 }` has an edge e from `O` to the number 3 with label $l = (property, \text{"foo"})$. A path P is simply a list of edges $(e_1, e_2, \ldots, e_n)$ where e_1 is an edge from the root node $(G.root)$.[1]

For the first heap snapshot, BLeak conservatively marks every node as *growing*. For subsequent snapshots, BLeak runs PropagateGrowth (Figure 4) to propagate the growth flags from the previous snapshot to the new snapshot, and discards the previous snapshot. On line 2, PropagateGrowth initializes every node in the new graph to *not growing* to

[1]For simplicity, we describe heap graphs as having a single root.

```
PropagateGrowth(G, G′)
1   Q = [(G.root, G′.root)], G′.root.mark = TRUE
2   for each node n ∈ G′.N
3       n.growing = FALSE
4   while |Q| > 0
5       (n, n′) = Dequeue(Q)
6       E_n = GetOutgoingEdges(G, n)
7       E′_n = GetOutgoingEdges(G′, n′)
8       n′.growing = n.growing ∧ |E_n| < |E′_n|
9       for each edge (n_1, n_2, l) ∈ E_n
10          for each edge (n′_1, n′_2, l′) ∈ E′_n
11              if l == l′ and n′_2.mark == FALSE
12                  n′_2.mark = TRUE
13                  Enqueue((n_2, n′_2))
```

Figure 4. PropagateGrowth propagates a node's growth status (*n.growing*) between heap snapshots. BLeak considers a path in the heap to be growing if the node at the path continually increases its number of outgoing edges.

```
FindLeakPaths(G)
Q = [], T_Gr = {}
for each edge e = (n_1, n_2, l) ∈ G.E where n_1 == G.root
    e.mark = TRUE
    Enqueue(Q, (NIL, e))
while |Q| > 0
    t = Dequeue(Q)
    (t_p, (n_1, n_2, l)) = t
    if n_2.growing == TRUE
        T_Gr = T_Gr ∪ {t}
    for each edge e = (n′_1, n′_2, l′) ∈ G.E
        if n′_1 == n_2 and e.mark == FALSE
            e.mark = TRUE
            Enqueue(Q, (t, e))
return T_Gr
```

Figure 5. FindLeakPaths, which returns paths through the heap to leaking nodes. The algorithm encodes each path as a list of edges formed by tuples (t).

prevent spuriously marking new growth as growing in the next run of the algorithm. Since the algorithm only considers paths that are the shortest path to a specific node, it is able to associate growth information with the terminal node which represents a specific path in the heap.

PropagateGrowth runs a breadth-first traversal across shared paths in the two graphs, starting from the root node that contains the global scope (`window`) and the DOM. The algorithm marks a node in the new graph as *growing* if the node at the same path in the previous graph is both growing and has fewer outgoing edges (line 8). As a result, the algorithm will only mark a heap path as a leak root if it consistently grows between every snapshot, and if it has been present since the first snapshot.

PropagateGrowth only visits paths shared between the two graphs (line 11). At a given path, the algorithm considers an outgoing edge e_n in the old graph and e'_n in the new graph as equivalent if they have the same label. In other words, the edges have to correspond to the same property name on the object at that path, or a closure variable with the same name captured by the function at that path.

After propagating growth flags to the final heap snapshot, BLeak runs FindLeakPaths (Figure 5) to record growing paths in the heap. This traversal visits *edges* in the graph to capture the shortest path to all unique edges that point to growing nodes. For example, if a growing object O is located at `window.O` and as variable p in the function `window.L.z`, FindLeakPaths will report both paths. This property is important for diagnosing leaks, as we discuss in Section 4.2.

BLeak takes the output of FindLeakPaths and groups it by the terminal node of each path. Each group corresponds to a specific leak root. This set of leak roots forms the input to the ranking algorithm.

4.2 Diagnosing Leaks

Given a list of leak roots and, for each root, a list of heap paths that point to the root, BLeak diagnoses leaks through hooks that run whenever the application performs any of the following actions:

- *Grows a leak root* with a new item. This growth occurs when the application adds a property to an object, an element to an array, an event listener to an event target, or a child node to a DOM node. BLeak captures a stack trace, and associates it with the new item.
- *Shrinks a leak root* by removing any of the previously-mentioned items. BLeak removes any stack traces associated with the removed items, as the items are no longer contributing to the leak root's growth.
- *Assigns a new value to a leak root*, which typically occurs when the application copies the state from an old version of the leaking object into a new version. BLeak removes all previously-collected stack traces for the leak root, collects a new stack trace, associates it with all of the items in the new value, and inserts the grow and shrink hooks into the new value.

BLeak runs one loop iteration of the application with all hooks installed. This process generates a list of stack traces responsible for growing each leak root.

4.3 Leak Root Ranking

BLeak uses a new metric to rank leak roots by return on investment that we call *LeakShare* (Figure 6). LeakShare prioritizes memory leaks that free the most memory with the

```
CalculateLeakShare(G, LR)
1  Q = [G.root], visitId = 0
2  for each node n ∈ G.N
3      n.mark = −1
4  while |Q| > 0
5      n = Dequeue(Q)
6      if n ∉ LR and n.mark ≠ visitId
7          n.mark = visitId
8          for each edge (n1, n2, l) ∈ G.E where n1 == n
9              Enqueue(Q, n2)
10 for each node n_root ∈ LR
11     visitId += 1
12     Q = [n_root]
13     while |Q| > 0
14         n = Dequeue(Q)
15         if n.mark ≠ 0 and n.mark ≠ visitId
16             n.mark = visitId
17             n.counter = n.counter + 1
18             for each (n1, n2, l) ∈ G.E where n1 == n
19                 Enqueue(Q, n2)
20 for each node n_root ∈ LR
21     visitId += 1
22     Q = [n_root]
23     while |Q| > 0
24         n = Dequeue(Q)
25         if n.counter ≠ 0 and n.mark ≠ visitId
26             n.mark = visitId
27             n_root.LS += n.size/n.counter
28             for each (n1, n2, l) ∈ G.E where n1 == n
29                 Enqueue(Q, n2)
```

Figure 6. CalculateLeakShare, which calculates the LeakShare metric ($n.LS$) for a set of leak roots LR.

least effort by dividing the "credit" for retaining a shared leaked object equally among the leak roots that retain them.

LeakShare first marks all of the items in the heap that are reachable from non-leaks via a breadth-first traversal that stops at leak roots (line 4). These nodes are ignored by subsequent traversals. Then, LeakShare performs a breadth-first traversal from each leak root that increments a counter on all reachable nodes (line 10). Once this process is complete, every node has a counter containing the number of leak roots that can reach it. Finally, the algorithm calculates the LeakShare of each leak root ($n.LS$) by adding up the size of each reachable node divided by its counter, which splits the "credit" for the node among all leak roots that can reach it (line 20).

5 Implementation

Applying BLeak's algorithms to web applications poses a number of significant engineering challenges:

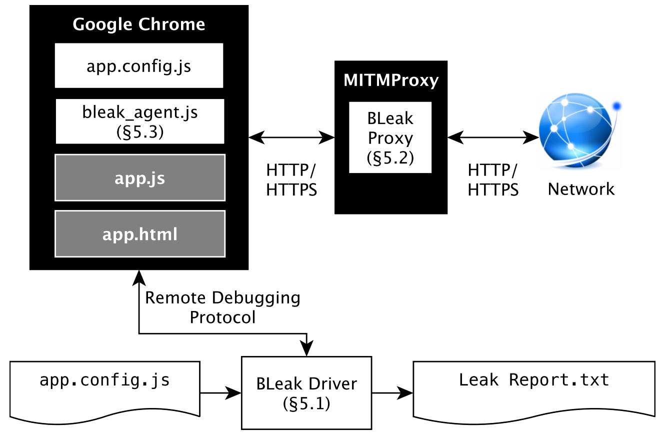

Figure 7. BLeak implementation overview. BLeak consists of a driver program that orchestrates the leak detection process (§5.1), a proxy that transparently rewrites the target web application's JavaScript during leak diagnosis (§5.2), and an agent script embedded in the application that hooks into relevant web APIs and leak roots (§5.3). Given a short developer-provided configuration script, BLeak automatically produces a leak report. White rectangles are BLeak components, gray items are automatically rewritten by the proxy during leak diagnosis, and black items are unmodified.

Leak identification and ranking: BLeak uses heap snapshots to identify and rank leaks, but many native methods (implemented in C++) do not expose their state to JavaScript heap snapshots. These native methods can harbor memory leaks and reduce the apparent severity of leaks that retain native state.

Leak diagnosis: BLeak's diagnostic strategy assumes that it can collect stack traces when relevant growth occurs, but the browser hides some state and state updates from JavaScript reflection. Native methods bypass JavaScript reflection and mutate state. JavaScript reflection cannot introspect into function closures, necessitating program transformations to expose this state. Transforming a web application is difficult because it can load code at any time from remote servers over HTTP or encrypted HTTPS. In addition, JavaScript contains dynamic features that are necessary but challenging to support with code transformations, including `eval` and `with` statements.

BLeak consists of three main components that work together to overcome these challenges (see Figure 7): (1) a driver program orchestrates the leak debugging process (§5.1); (2) a proxy transparently performs code rewriting on-the-fly on the target web application and `eval`-ed strings (§5.2); and (3) an agent script embedded within the application exposes hidden state for leak detection and growth events for leak diagnosis (§5.3).

5.1 BLeak Driver

The BLeak driver is responsible for orchestrating the leak debugging process. To initiate leak debugging, the driver launches BLeak's proxy and a standard version of the Google Chrome browser with an empty cache, a fresh user profile, and a configuration that uses the BLeak proxy. The driver connects to the browser via the standard Chrome DevTools Protocol [19], navigates to the target web application, and uses the developer-provided configuration file to drive the application in a loop. After each repeated visit to the first visual state in the loop, the driver takes a heap snapshot via the remote debugging protocol, and runs PropagateGrowth (Figure 4) to propagate growth information between heap snapshots. Prior to taking a heap snapshot, the driver calls a method in the BLeak agent embedded in the web application that prepares the DOM for snapshotting (§5.3.2).

At the end of a configurable number of loop iterations (the default is 8), the driver shifts into diagnostic mode. The driver runs FindLeakPaths to locate all of the paths to all of the leak roots (Figure 5), configures the proxy to perform code rewriting for diagnosis (§5.2), and reloads the page to pull in the transformed version of the web application. The driver runs the application in a single loop iteration before triggering the BLeak agent to insert diagnostic hooks that collect stack traces at all of the paths reported by FindLeakPaths (§5.3.1). Then, the driver runs the application in a final loop before retrieving the collected stack traces from the agent. Finally, the driver runs LeakShare (Figure 6) to rank the leak roots and generates a memory leak report.

5.2 BLeak Proxy

The BLeak proxy uses mitmproxy [11] to transparently intercept all HTTP and HTTPS traffic between the web application and the network. The proxy rewrites the web application's JavaScript during leak diagnosis to move closure variables into explicit scope objects, chains scope objects together to enable scope lookup at runtime, and exposes an HTTP endpoint for transforming `eval`-ed code. The proxy also injects the BLeak agent and developer-provided configuration file into the application, uses Babel [4] to translate emerging JavaScript features into code that BLeak can understand, and supports the JavaScript `with` statement. Due to space constraints we do not discuss these features further.

Exposing closure variables for diagnosis: During leak diagnosis, the BLeak proxy rewrites the JavaScript on the webpage, including JavaScript inlined into HTML, to make it possible for the BLeak agent to instrument closure variables. Since this process distorts the application's memory footprint, BLeak does not use this process during leak detection and ranking. This code transformation moves local variables into JavaScript "scope" objects (Imagen uses a similar procedure to implement JavaScript heap snapshots [33]). Scope objects are ordinary JavaScript objects where property `foo` refers to the local variable `foo`; the browser-provided `window` object functions as a global scope object, and works identically. BLeak adds a `__scope__` property to all JavaScript `Function` objects that refer to that function's defining scope, and rewrites all variable reads and writes to refer to properties in the scope object. With this transformation, the BLeak agent can capture variable updates in the transformed program in the same manner as object properties.

As an optimization, BLeak performs a conservative escape analysis to avoid transforming variables that are not captured by any function closures. However, if the program calls `eval` or uses the `with` statement, then BLeak assumes that all reachable variables escape.

The scope object transformation treats function arguments differently than local variables. A function's arguments are reflected in an implicit array-like object called `arguments`, and updates to an argument also update the corresponding element in `arguments`.[2] To preserve this behavior, BLeak rewrites updates to arguments so that it simultaneously updates the property in the scope object and the original argument variable.

Runtime scope lookup: The JavaScript transformation knows statically which scope objects contain which variables, but the BLeak agent needs this information at runtime to instrument the correct scope object for a given variable. One solution would be to reify scope information into runtime metadata objects that the agent can query, but this would add further runtime and memory overhead. Instead, the proxy uses a simpler design that uses JavaScript's built-in prototype inheritance to naturally encode scope chains. Each scope object inherits from its parent, and the outermost scope object inherits from the browser-provided `window` object. To perform scope lookup, the BLeak agent uses JavaScript reflection to find the first scope object in the chain that defines a property corresponding to the variable.

`eval` support: `eval` evaluates a string as code within the context of the call site, posing two key challenges: (1) the string may not be known statically, and (2) the string may refer to outer variables that the code transformation moved into scope objects. The proxy overcomes these challenges by cooperating with the BLeak agent. The proxy transforms all references to `eval` into references to a BLeak agent-provided function that sends the program text synchronously to the proxy for transformation via an HTTP POST. The proxy transforms `eval`-ed code so that references to variables not explicitly defined in the new code refer to a single scope object, and then returns the transformed code to the agent. The agent creates the single scope object as an ECMAScript 2015 Proxy object [46] that interposes on property reads and writes to relay them to the appropriate scope object using runtime scope lookup (Proxy objects are available in modern

[2] This behavior does not occur in "strict mode", but many prominent libraries do not opt into "strict mode".

versions of all major browsers). Finally, the agent calls `eval` on the transformed code. Since this code transformation is independent of calling context, the BLeak agent can cache and re-use transformed code strings.

5.3 BLeak Agent

The BLeak agent is a JavaScript file that BLeak automatically embeds in the web application; it exposes globally-accessible functions that the BLeak driver can invoke via the Chrome DevTools Protocol. The agent is responsible for installing diagnostic hooks that collect stack traces for growth events. The agent also exposes hidden state in the browser's native methods so that PropagateGrowth (Figure 4) can find leaks within or accessible through this state.

5.3.1 Diagnostic Hooks

To diagnose memory leaks as described in Section 4.2, the BLeak agent needs to interpose on leak root growth, shrinkage, and assignment events. Although all leak roots are JavaScript objects, some types of objects have native browser methods that implicitly grow, shrink, or assign to properties on the object, necessitating interface-specific hooks:

Object hooks: BLeak uses Proxy objects to detect when objects gain and lose properties. These Proxy objects wrap JavaScript objects and expose hooks for various object operations, including when the application adds, deletes, reads, or writes properties on the object.

Proxy objects do not automatically take the place of the object they wrap in the heap, so the BLeak agent must replace all references to the object with the proxy to completely capture all growth/shrinkage events. If the agent fails to replace a reference, then it will not capture any object updates that occur through that reference. BLeak can miss a reference if it does not appear in the heap snapshot used for FindLeakPaths (Figure 5). This could happen if the heap path to the reference is determined by some external factor, such as the clock, a random number, or the amount of time spent on the page. This behavior appears to be rare; in our evaluation, BLeak reports all but one of the relevant stack traces for all of the true leaks it finds.

Proxy objects are semantically equivalent to the original object except that programs can observe that Proxy(O) $\neq O$. Since BLeak cannot guarantee that it replaces all references to O with Proxy(O), a program could run incorrectly if it directly compared these two objects. To preserve correctness, the BLeak proxy also transforms the binary operations ==, ===, !=, !== into calls to an agent function that treats Proxy(O) as equal to O. The BLeak agent also reimplements the functions `Array.indexOf` and `Array.lastIndexOf`, which report the index of a particular item in an array, so that calls with Proxy objects function appropriately.

Array hooks: JavaScript arrays contain a number of built-in functions that mutate the array without invoking Proxy hooks. The agent wraps `Array`'s `push`, `pop`, `unshift`, `shift`, and `splice` functions to appropriately capture growth / shrinkage / assignment events.

DOM node hooks: Applications can add and remove nodes from the DOM tree via browser-provided interfaces; these operations are not captured via Proxy objects. In order to capture relevant events on DOM nodes, the agent must wrap a number of functions and special properties. On `Node` objects, it wraps `textContent`, `appendChild`, `insertBefore`, `normalize`, `removeChild`, and `replaceChild`. On `Element` objects, it wraps `innerHTML`, `outerHTML`, `insertAdjacentElement`, `insertAdjacentHTML`, `insertAdjacentText`, and `remove`.

Leak root assignment hooks: Given a path $P = (e_1, \ldots, e_n)$ to a leak root, the agent instruments all edges $e \in P$ to capture when the program overwrites any objects or variables in the path from the GC root to the leak root. For example, given the path `window.foo.bar`, the program can overwrite `bar` by assigning a new value to `foo` or `bar`. When a leak root gets overwritten with a new value, the agent also wraps that value in a Proxy object.

To interpose on these edges, the agent uses JavaScript reflection to replace object properties with getters and setters that interpose on its modification. Since the BLeak proxy rewrites closure variables into properties on scope objects (§5.2), this approach works for all edges in the heap graph.

5.3.2 Exposing Hidden State

Some of the browser's native methods hide state from heap snapshots, preventing BLeak from accurately identifying and ranking memory leaks involving this state. To overcome this limitation, the agent builds a mirror of hidden state using JavaScript objects. Using these mirrors, BLeak can locate and diagnose memory leaks that are in or accessible through DOM nodes, event listeners, and partially applied functions.

DOM nodes: The agent builds a mirror of the DOM tree as JavaScript objects before the BLeak driver takes a heap snapshot, and installs it at the global variable `$$$DOM$$$`. Each node in the tree contains the array `childNodes` that contains a JavaScript array of (mirror) nodes, and a property `root` that points to the original native DOM node.

Event listeners: The agent overwrites `addEventListener` and `removeEventListener` to eagerly maintain an object containing all of the installed listeners. Because this object is maintained eagerly, ordinary object and array hooks capture event listener list growth.

`Function.bind`: The `bind` function provides native support for partial application, and implicitly retains the arguments passed to it. The agent overwrites this function with a pure JavaScript version that retains the arguments as ordinary JavaScript closure variables.

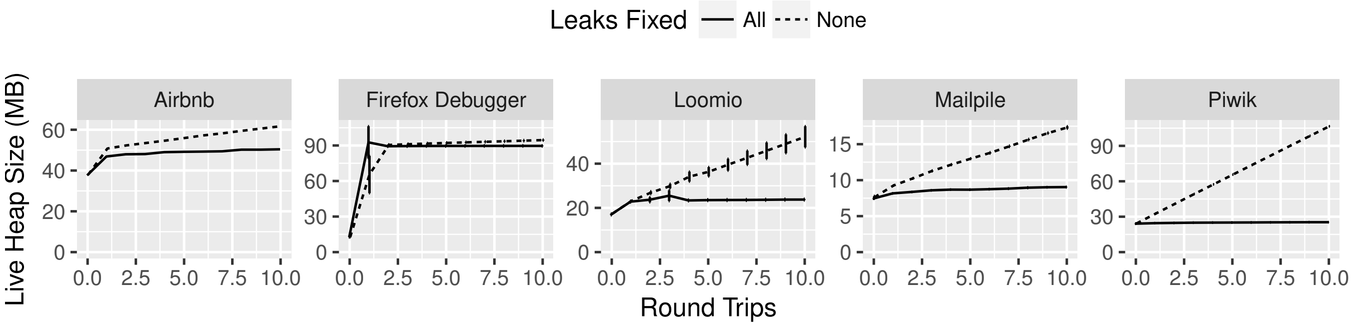

Figure 8. Impact of fixing memory leaks found with BLeak: Graphs display live heap size over round trips; error bars indicate the 95% confidence interval. Fixing the reported leaks eliminates an average of 93% of all heap growth.

6 Evaluation

We evaluate BLeak by running it on production web applications. Our evaluation addresses the following questions:

- **Precision:** How precise is BLeak's memory leak detection? (§6.2)
- **Accuracy of diagnoses:** Does BLeak accurately locate the code responsible for memory leaks? (§6.2)
- **Overhead:** Does BLeak impose acceptable overhead? (§6.2)
- **Impact of discovered leaks:** How impactful are the memory leaks that BLeak finds? (§6.3)
- **Utility of ranking:** Is LeakShare an effective metric for ranking the severity of memory leaks? (§6.4)
- **Staleness vs. growth:** How does BLeak compare to a staleness-based leak detector? (§6.5)

Our evaluation finds **59 distinct memory leaks** across five web applications, *all of which were unknown to application developers*. Of these, 27 corresponded to known-but-unfixed memory leaks in JavaScript library dependencies, of which only 6 were independently diagnosed and had pending fixes. We reported all 32 new memory leaks to the relevant developers along with our fixes; 16 are now fixed, and 4 have fixes in code review. We find new leaks in popular applications and libraries including Airbnb, Angular JS (1.x), Google Maps SDK, Google Tag Manager, and Google Analytics. Appendix A lists each of these memory leaks, the application or library responsible, and links to bug reports with fixes.

We run BLeak on each web application for 8 round trips through specific visual states to produce a BLeak leak report, as in Figure 3b. We describe these loops using only 17–73 LOC; Appendix B contains the code for each loop. This process takes less than 15 minutes per application on our evaluation machine, a MacBook Pro with a 2.9 GHz Intel Core i5 and 16GB of RAM. For each application, we analyze the reported leaks, write a fix for each true leak, measure the impact of fixing the leaks, and compare LeakShare with alternative ranking metrics.

6.1 Applications

Because there is no existing corpus of benchmarks for web application memory leak detection, we created one. Our corpus consists of five popular web applications that both comprise large code bases and whose overall memory usage appeared to be growing over time. We primarily focus on open source web applications because it is easier to develop fixes for the original source code; this represents the normal use case for developers. We also include a single closed-source website, Airbnb, to demonstrate BLeak's ability to diagnose websites in production. We present each web application, highlight a selection of the libraries they use, and describe the loop of visual states we use in our evaluation:

Airbnb [2]: A website offering short-term rentals and other services, Airbnb uses React, Google Maps SDK, Google Analytics, the Criteo OneTag Loader, and Google Tag Manager. BLeak loops between the pages `/s/all`, which lists all services offered on Airbnb, and `/s/homes`, which lists only homes and rooms for rent.

Piwik 3.0.2 [51]: A widely-used open-source analytics platform; we run BLeak on its in-browser dashboard that displays analytics results. The dashboard primarily uses jQuery and AngularJS. BLeak repeatedly visits the main dashboard page, which displays a grid of widgets.

Loomio 1.8.66 [34]: An open-source collaborative platform for group decision-making. Loomio uses AngularJS, LokiJS, and Google Tag Manager. BLeak runs Loomio in a loop between a group page, which lists all of the threads in that group, and the first thread listed on that page.

Mailpile v1.0.0 [36]: An open-source mail client. Mailpile uses jQuery. BLeak runs Mailpile's demo [35] in a loop that visits the inbox and the first four emails in the inbox (revisiting the inbox in-between emails).

Firefox Debugger (commit 91f5c63) [15]: An open-source JavaScript debugger written in React that runs in any web browser. We run the debugger while it is attached to a Firefox instance running Mozilla's SensorWeb [45]. BLeak runs

Program	Loop LOC	Leak Roots	False Pos.	Distinct Leaks	Stale Leaks	Prec.	Growth Reduction	Runtime Total	Algs	Proxy	Snapshot
Airbnb	17	32	2	32	4	94%	1.04 MB (81.0%)				
Piwik	32	17	0	11	4	100%	8.14 MB (99.3%)	224 s	7.1%	4.7%	50.3%
Loomio	73	10	1	9	4	90%	2.83 MB (98.3%)	149 s	3.7%	6.2%	37.9%
Mailpile	37	4	0	3	1	100%	0.80 MB (91.8%)	388 s	0.4%	1.0%	4.0%
Firefox Debugger	17	4	0	4	0	100%	0.47 MB (98.2%)	214 s	2.3%	37.9%	33.6%
Total / mean:	35	67	3	59	13	96.8%	2.66 MB (93.7%)	243.8 s	3.4%	12.5%	31.4%

Figure 9. BLEAK precisely finds impactful memory leaks: On average, BLEAK finds these leaks with over 95% precision, and fixing them eliminates over 90% of all heap growth. 77% of these leaks would not be found with a staleness metric (§6.5).

the debugger in a loop that opens and closes SensorWeb's `main.js` in the debugger's text editor.

6.2 Precision, Accuracy, and Overhead

To determine BLEAK's leak detection precision and the accuracy of its diagnoses, we manually check each BLEAK-reported leak in the final report to confirm (1) that it is growing without bound and (2) that the stack traces correctly report the code responsible for the growth. To determine BLEAK's overhead, we log the runtime of the following specific operations during automatic leak debugging: BLEAK's three core algorithms from §4 (Algs), proxy transformations from §5.2 (Proxy), and receiving and parsing heap snapshots from Google Chrome (Snapshot). We were unable to gather overhead information for Airbnb, the only closed-source application, because the company fixed the leaks we reported prior to this experiment. Figure 9 summarizes our results.

BLEAK has an average precision of 96.8%, and a median precision of 100% on our evaluation applications. There are only three false positives. All point to an object that continuously grows until some threshold or timeout occurs; developers using BLEAK can avoid these false positives by increasing the number of round trips. Two of the three false positives are actually the same object located in the Google Tag Manager JavaScript library.

With one exception, BLEAK accurately identifies the code responsible for all of the true leaks. BLEAK reports stack traces that directly identifies the code responsible for each leak. In cases where multiple independent source locations grow the same leak root, BLEAK reports all relevant source locations. For one specific memory leak, BLEAK fails to record a stack trace. **Guided by BLEAK's leak reports, we were able to fix every memory leak.** Fixing each memory leak took approximately 15 minutes. Most fixes involve adding simple cleanup hooks to remove unneeded references or logic to avoid duplicating state every round trip.

BLEAK locates, ranks, and diagnoses memory leaks in less than 7 minutes on our open-source evaluation applications. BLEAK's core algorithms (PROPAGATEGROWTH, FINDLEAKPATHS, CALCULATELEAKSHARE) contribute less than 8% to that runtime. The primary contribution to overhead on all benchmarks, with one exception, is receiving and parsing Chrome's JSON-based heap snapshots. The Firefox Debugger spends more time in the proxy because it uses new JavaScript features that BLEAK supports by invoking the Babel compiler, which dominates proxy runtime for that application [4].

6.3 Leak Impact

To determine the impact of the memory leaks that BLEAK reports, we measure each application's live heap size over 10 loop iterations with and without our fixes. We use BLEAK's HTTP/HTTPS proxy to directly inject memory leak fixes into the application, which lets us test fixes on closed-source websites like Airbnb. We run each application except Airbnb 5 times in each configuration (we run Airbnb only once per configuration for reasons discussed in §6.4).

To calculate the leaks' combined impact on overall heap growth, we calculate the average live heap growth between loop iterations with and without the fixes in place, and take the difference (Growth Reduction). For this metric, we ignore the first five loop iterations because these are noisy due to application startup. Figure 8 and Figure 9 present the results.

On average, fixing the memory leaks that BLEAK reports eliminates over 93% of all heap growth on the evaluation applications (median: 98.2%). These results suggest that BLEAK does not miss any significantly impactful leaks.

6.4 LeakShare Effectiveness

We compare LeakShare against two alternative ranking metrics: retained size and transitive closure size. Retained size corresponds to the amount of memory the garbage collector would reclaim if the leak root were removed from the heap graph, and is the metric that standard heap snapshot viewers display to the developer [29, 40, 44, 49]. The transitive closure size of a leak root is the size of all objects reachable from the leak root; Xu *et al.* use this metric along with staleness to rank Java container memory leaks [74]. Since JavaScript heaps are highly connected and frequently contain references to the global scope, we expect this metric to report similar values for most leaks.

Growth Reduction for Top Leaks Fixed

Program	Metric	25%	50%	75%
Airbnb	LeakShare	0K	111K	**462K**
	Retained Size	0K	0K	105K
	Trans. Closure Size	0K	**196K**	393K
Loomio	LeakShare	0K	**1083K**	**2878K**
	Retained Size	**64K**	186K	**2898K**
	Trans. Closure Size	59K	67K	2398K
Mailpile	LeakShare	**613K**	**817K**	**820K**
	Retained Size	**613K**	**817K**	**820K**
	Trans. Closure Size	0K	0K	201K
Piwik	LeakShare	**8003K**	**8104K**	**8306K**
	Retained Size	2073K	7969K	**8235K**
	Trans. Closure Size	103K	110K	374K

Figure 10. Performance of ranking metrics: Growth reduction by metric after fixing quartiles of top ranked leaks. **Bold** indicates greatest reduction (±1%). We omit Firefox because it has only four leaks which must all be fixed (see §2). LeakShare generally outperforms or matches other metrics.

We measure the effectiveness of each ranking metric by calculating the growth reduction (as in §6.3) over the application with no fixes after fixing each memory leak in ranked order. We then calculate the quartiles of this data, indicating how much heap growth is eliminated after fixing the top 25%, 50%, and 75% of memory leaks reported ranked by a given metric. We sought to write patches for each evaluation application that fix a single leak root at a time, but this is not feasible in all cases. Specifically, one Airbnb patch fixes two leak roots; one Mailpile patch (a jQuery bug) fixes two leak roots; and one Piwik patch, which targeted a loop, fixes nine leak roots. In these cases, we apply the patch during a ranking for the first relevant leak root reported.

We run each application except Airbnb for ten loop iterations over five runs for each unique combination of metric and number of top-ranked leak roots to fix. We avoid running duplicate configurations when multiple metrics report the same ranking. Airbnb is challenging to evaluate because it has 30 leak roots, randomly performs A/B tests between runs, and periodically updates its minified codebase in ways that break our memory leak fixes. As a result, we were only able to gather one run of data for Airbnb for each unique configuration. Figure 10 displays the results.

In most cases, LeakShare outperforms or ties the other metrics. LeakShare initially is outperformed by other metrics on Airbnb and Loomio because it prioritizes leak roots that share significant state with other leak roots. Retained size always prioritizes leak roots that uniquely own the most state, which provide the most growth reduction in the short term. LeakShare eventually surpasses the other metrics on these two applications as it fixes the final leak roots holding on to shared state.

6.5 Leak Staleness

We manually analyzed the leaks BLeak finds to determine whether they would also be found using a staleness-based technique. We assume that, to avoid falsely reporting most event listeners as stale, a staleness-based technique would exercise each event listener on the page that could be triggered via normal user interaction. In this case, no memory leaks stemming from event listener lists would be found by a staleness-based tool. Leaks in internal application arrays and objects that emulate event listener lists for user-triggered events would also not be found. Finally, we assume that active DOM elements in the DOM tree would not be marked stale, since they are clearly in use by the webpage. Memory leaks stemming from node lists in the DOM would also not be found by a staleness-based technique. **Of the memory leaks BLeak finds, at least 77% would not be found with a staleness-based approach.** Figure 9 presents results per application (see Appendix A for individual leaks).

7 Related Work

Web application memory leak detectors: BLeak automatically debugs memory leaks in modern web applications; past work in this space is ineffective, out of date, or not sufficiently general. LeakSpot locates JavaScript allocation and reference sites that produce and retain increasing numbers of objects over time, and uses staleness as a heuristic to refine its output [52]. On real web applications, LeakSpot typically reports over 50 different allocation and reference sites that developers must manually inspect to identify and diagnose memory leaks. AjaxScope dynamically detects leaks due to a bug in web browsers that has now been fixed [30]. JSWhiz statically analyzes code written with Google Closure type annotations to detect specific leak patterns [50].

Web application memory debugging: Some tools help web developers debug memory usage and present diagnostic information that the developer must manually interpret to locate leaks (Section 1 describes Google Chrome's Development Tools). MemInsight summarizes and displays information about the JavaScript heap, including per-object-type staleness information, the allocation site of individual objects, and retaining paths in the heap [27]. Unlike BLeak, these tools do not directly identify memory as leaking or identify the code responsible for leaks.

Growth-based memory leak detection: LeakBot looks for patterns in the heap graphs of Java applications to find memory leaks [41]. LeakBot assumes that leak roots own all of their leaking objects, but leaked objects in web applications frequently have multiple owners. BLeak does not rely on specific patterns, and uses round trips to the same visual state to identify leaking objects. Cork uses static type information available in the JVM to locate types that appear to be the source of memory leaks. [28]. Cork is not applicable to dynamically typed languages like JavaScript.

Staleness-based memory leak detection: SWAT (C/C++), Sleigh (JVM), and Hound (C/C++) find leaking objects using a staleness metric derived from the last time an object was accessed, and identify the call site responsible for allocating them [8, 25, 48]. Leakpoint (C/C++) also identifies the last point in the execution that referenced a leaking memory location [9]. As we show (§6.5), staleness is ineffective for at least 77% of the memory leaks BLeak identifies .

Hybrid leak detection approaches: Xu *et al.* identify leaks stemming from Java collections using a hybrid approach that targets containers that grow in size over time and contain stale items. The vast majority of memory leaks found by BLeak would not be considered stale (§6.5).

Specification-based memory leak detection: LeakChaser and GC Assertions let developers manually annotate their programs with heap invariants that, when broken, indicate a memory leak [1, 73]. BLeak's growth-based approach identifies memory leaks automatically without taking liveness relationships into account.

8 Conclusion

This paper presents BLeak, the first effective system for debugging client-side memory leaks in web applications. We show that BLeak has high precision and finds numerous previously-unknown memory leaks in web applications and libraries. BLeak is open source [71], and is available for download at http://bleak-detector.org/.

We believe the insights we develop for BLeak are applicable to a broad class of GUI applications, including mobile applications. Many mobile applications are actually hybrid applications, which combine both native and browser components. However, even native GUI applications, like web applications, are commonly event-driven and repeatedly visit specific views. We plan in future work to explore the application of BLeak's techniques to find memory leaks in GUI applications.

Acknowledgments

John Vilk was supported by a Facebook PhD Fellowship. This material is based upon work supported by the National Science Foundation under Grant No. 1637536. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Edward Aftandilian and Samuel Z. Guyer. 2009. GC Assertions: Using the Garbage Collector to Check Heap Properties. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 235–244. https://doi.org/10.1145/1542476.1542503

[2] Airbnb, Inc. 2017. Vacation Rentals, Homes, Experiences, & Places - Airbnb. http://airbnb.com/. [Online; accessed 12-October-2017].

[3] Bill Allombert, Avery Pennarun, Bill Allombert, and Petter Reinholdtsen. 2017. Debian Popularity Contest. https://popcon.debian.org/. See entries for iceweasel and chromium-browser. [Online; accessed 4-November-2017].

[4] Babel. 2017. Babel - The compiler for writing next generation JavaScript. https://babeljs.io/. [Online; accessed 15-October-2017].

[5] Dmitry Baranovskiy. 2017. DmitryBaranovskiy/raphael: JavaScript Vector Library. https://github.com/DmitryBaranovskiy/raphael. [Online; accessed 6-November-2017].

[6] Kayce Basques. 2017. Fix Memory Problems. https://developers.google.com/web/tools/chrome-devtools/memory-problems/. [Online; accessed 2-November-2017].

[7] Jason Bedard. 2017. Deferred: fix memory leak of promise callbacks. https://github.com/jquery/jquery/pull/3657. [Online; accessed 8-November-2017].

[8] Michael D. Bond and Kathryn S. McKinley. 2006. Bell: bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. 61–72. https://doi.org/10.1145/1168857.1168866

[9] James A. Clause and Alessandro Orso. 2010. LEAKPOINT: pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. 515–524. https://doi.org/10.1145/1806799.1806874

[10] Jeff Collins. 2017. jeff-collins/ment.io: Mentions and Macros for Angular. https://github.com/jeff-collins/ment.io. [Online; accessed 6-November-2017].

[11] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. 2010–. mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/ [Online; accessed 15-October-2017].

[12] Criteo. 2017. Criteo OneTag Explained. https://support.criteo.com/hc/en-us/articles/202726972-Criteo-OneTag-explained. [Online; accessed 6-November-2017].

[13] Steve Dent. 2017. Firefox 'performance' tab will curb its RAM hunger. https://www.engadget.com/2017/04/13/firefox-performance-tab-will-curb-its-ram-hunger/. [Online; accessed 4-November-2017].

[14] Filament Group, Inc. 2017. filamentgroup/loadCSS: A function for loading CSS asynchronously. https://github.com/filamentgroup/loadCSS. [Online; accessed 6-November-2017].

[15] Firefox Developer Tools Team. 2017. debugger.html: The Firefox debugger that works anywhere. http://firefox-dev.tools/debugger.html/. [Online; accessed 12-October-2017].

[16] Google. 2011. Bug: Destroying Google Map Instance Never Frees Memory. https://issuetracker.google.com/issues/35821412. [Online; accessed 2-November-2017].

[17] Google. 2017. Adding analytics.js to Your Site. https://developers.google.com/analytics/devguides/collection/analyticsjs/. [Online; accessed 6-November-2017].

[18] Google. 2017. angular/angular.js: AngularJS - HTML enhanced for web apps! https://github.com/angular/angular.js. [Online; accessed 6-November-2017].

[19] Google. 2017. Chrome DevTools Protocol Viewer. https://chromedevtools.github.io/devtools-protocol/. [Online; accessed 7-November-2017].

[20] Google. 2017. Google Maps JavaScript API. https://developers.google.com/maps/documentation/javascript/. [Online; accessed 6-November-2017].

[21] Google. 2017. Speed up Google Chrome. https://support.google.com/chrome/answer/1385029. [Online; accessed 4-November-2017].

[22] Google. 2017. Tag Management Solutions for Web and Mobile. https://www.google.com/analytics/tag-manager/. [Online; accessed 6-November-2017].

[23] Kentaro Hara. 2013. Oilpan: GC for Blink. https://docs.google.com/presentation/d/1YtfurcyKFS0hxPOnC3U6JJroM8aRP49Yf0QWznZ9jrk. [Online; accessed 4-November-2017].

[24] Kentaro Hara. 2017. State of Blink's Speed. https://docs.google.com/presentation/d/1Az-F3CamBq6hZ5QqQt-ynQEMWEhHY1VTvlRwL7b_6TU. See slide 46. [Online; accessed 2-November-2017].

[25] Matthias Hauswirth and Trishul M. Chilimbi. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. 156–164. https://doi.org/10.1145/1024393.1024412

[26] Alan Henry. 2011. How Do I Stop My Browser from Slowing to a Crawl? https://lifehacker.com/5833074/how-do-i-stop-my-browser-from-slowing-to-a-crawl. [Online; accessed 4-November-2017].

[27] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. 2015. MemInsight: Platform-Independent Memory Debugging for JavaScript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 345–356. https://doi.org/10.1145/2786805.2786860

[28] Maria Jump and Kathryn S. McKinley. 2007. Cork: dynamic memory leak detection for garbage-collected languages. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 31–38. https://doi.org/10.1145/1190216.1190224

[29] Meggin Kearney. 2017. How to Record Heap Snapshots. https://developers.google.com/web/tools/chrome-devtools/memory-problems/heap-snapshots. [Online; accessed 11-November-2017].

[30] Emre Kiciman and Benjamin Livshits. 2007. AjaxScope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*. 17–30. https://doi.org/10.1145/1294261.1294264

[31] Loreena Lee and Robert Hundt. 2012. BloatBusters: Eliminating memory leaks in Gmail. https://docs.google.com/presentation/d/1wUVmf78gG-ra5aOxvTfYdiLkdGaR9OhXRnOlIcEmu2s. [Online; accessed 2-November-2017].

[32] Joe Lencioni. 2017. Possible memory leak when used with the same URL multiple times. https://github.com/filamentgroup/loadCSS/issues/236. [Online; accessed 6-November-2017].

[33] James Teng Kin Lo, Eric Wohlstadter, and Ali Mesbah. 2013. Imagen: runtime migration of browser sessions for JavaScript web applications. In *22nd International World Wide Web Conference*. 815–826. http://dl.acm.org/citation.cfm?id=2488459

[34] Loomio Cooperative Limited. 2017. Loomio - Better decisions together. https://www.loomio.org/. [Online; accessed 12-October-2017].

[35] Mailpile Team. 2017. Mailpile Demo's mailpile v1.0.0rc0. https://demo.mailpile.is/in/inbox/. [Online; accessed 8-November-2017].

[36] Mailpile Team. 2017. Mailpile: e-mail that protects your privacy. http://mailpile.is/. [Online; accessed 12-October-2017].

[37] Materialize. 2017. Dogfalo/materialize: Materialize, a CSS Framework based on Material Design. https://github.com/Dogfalo/materialize. [Online; accessed 6-November-2017].

[38] Kirk McElhearn. 2016. It's time for Safari to go on a memory diet. https://www.macworld.com/article/3148256/browsers/it-s-time-for-safari-to-go-on-a-memory-diet.html. [Online; accessed 4-November-2017].

[39] Kirk McElhearn. 2017. Apple's Safari Web Browser Now Uses Much Less Memory. https://www.kirkville.com/apples-safari-web-browser-now-uses-much-less-memory/. [Online; accessed 4-November-2017].

[40] Microsoft. 2017. Microsoft Edge F12 DevTools - Memory. https://docs.microsoft.com/en-us/microsoft-edge/f12-devtools-guide/memory. [Online; accessed 4-November-2017].

[41] Nick Mitchell and Gary Sevitsky. 2003. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *Proceedings of ECOOP 2003 - Object-Oriented Programming*. 351–377. https://doi.org/10.1007/978-3-540-45070-2_16

[42] Mozilla. 2017. about:memory. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/about:memory. [Online; accessed 4-November-2017].

[43] Mozilla. 2017. Firefox uses too much memory (RAM) - How to fix. https://support.mozilla.org/en-US/kb/firefox-uses-too-much-memory-ram. [Online; accessed 4-November-2017].

[44] Mozilla. 2017. Memory - Firefox Developer Tools. https://developer.mozilla.org/en-US/docs/Tools/Memory. [Online; accessed 11-November-2017].

[45] Mozilla. 2017. SensorWeb. http://aws-sensorweb-static-site.s3-website-us-west-2.amazonaws.com/. [Online; accessed 2-November-2017].

[46] Mozilla Development Network. 2017. Proxy - JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy. [Online; accessed 7-November-2017].

[47] Nick Nguyen. 2017. The Best Firefox Ever. https://blog.mozilla.org/blog/2017/06/13/faster-better-firefox/. [Online; accessed 4-November-2017].

[48] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2009. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 397–407. https://doi.org/10.1145/1542476.1542521

[49] Joseph Pecoraro. 2016. Memory Debugging with Web Inspector. https://webkit.org/blog/6425/memory-debugging-with-web-inspector/. [Online; accessed 4-November-2017].

[50] Jacques A. Pienaar and Robert Hundt. 2013. JSWhiz: Static analysis for JavaScript memory leaks. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*. 11:1–11:11. https://doi.org/10.1109/CGO.2013.6495007

[51] Piwik.org. 2017. #1 Free Web & Mobile Analytics Software. https://piwik.org/. [Online; accessed 12-October-2017].

[52] Masoomeh Rudafshani and Paul A. S. Ward. 2017. LeakSpot: Detection and diagnosis of memory leaks in JavaScript applications. *Software: Practice and Experience* 47, 1 (2017), 97–123. https://doi.org/10.1002/spe.2406

[53] Greg Sterling. 2015. No, Apps Aren't Winning. The Mobile Browser Is. http://marketingland.com/morgan-stanley-no-apps-arent-winning-themobile-browser-is-144303. [Online; accessed 2-November-2017].

[54] The jQuery Foundation. 2017. jquery/jquery: jQuery JavaScript Library. https://github.com/jquery/jquery/. [Online; accessed 6-November-2017].

[55] John Vilk. 2017. [Browser Client] Minor memory leak: ".mail_source" event resubscriptions. https://github.com/mailpile/Mailpile/issues/1911. [Online; accessed 8-November-2017].

[56] John Vilk. 2017. [Browser Client] Minor memory leak: Text nodes in notification area. https://github.com/mailpile/Mailpile/issues/1931. [Online; accessed 8-November-2017].

[57] John Vilk. 2017. Fix memory leak in Element.removeData(). https://github.com/DmitryBaranovskiy/raphael/pull/1077. [Online; accessed 8-November-2017].

[58] John Vilk. 2017. Fix memory leaks in data table / jqplot. https://github.com/piwik/piwik/pull/11354. [Online; accessed 8-November-2017].

[59] John Vilk. 2017. Fix multiple memory leaks in UserCountryMap. https://github.com/piwik/piwik/pull/11350. [Online; accessed 8-November-2017].

[60] John Vilk. 2017. Fix UIControl memory leak. https://github.com/piwik/piwik/pull/11362. [Online; accessed 8-November-2017].

[61] John Vilk. 2017. JavaScript Memory Leak: #columnPreview click handlers. https://github.com/piwik/piwik/issues/12058. [Online; accessed 8-November-2017].

[62] John Vilk. 2017. JavaScript Memory Leak: widgetContent $destroy handlers. https://github.com/piwik/piwik/issues/12059. [Online; accessed 8-November-2017].

[63] John Vilk. 2017. Memory Leak: gtm.js repeatedly appends conversion_async.js to head when pushing to dataLayer. https://goo.gl/WFPt4M. [Online; accessed 6-November-2017].

[64] John Vilk. 2017. Memory Leak in Preview Component. https://github.com/devtools-html/debugger.html/issues/3822. [Online; accessed 8-November-2017].

[65] John Vilk. 2017. Memory Leak: material_select never removes global click handlers. https://github.com/Dogfalo/materialize/issues/4266. [Online; accessed 8-November-2017].

[66] John Vilk. 2017. Minor frontend memory leaks due to unremoved LokiJS dynamic views. https://github.com/loomio/loomio/issues/4248. [Online; accessed 8-November-2017].

[67] John Vilk. 2017. Minor JavaScript Memory Leak: piwikApiService allRequests array. https://github.com/piwik/piwik/issues/12105. [Online; accessed 8-November-2017].

[68] John Vilk. 2017. Small Memory Leak and Correctness Bug in analytics.js. https://issuetracker.google.com/issues/66525724. [Online; accessed 6-November-2017].

[69] John Vilk. 2017. Small memory leak: Callbacks added to window._xdc_ are never cleared. https://issuetracker.google.com/issues/66529186. [Online; accessed 6-November-2017].

[70] John Vilk. 2017. Small Memory Leak in $rootScope.$on. https://github.com/angular/angular.js/issues/16135. [Online; accessed 8-November-2017].

[71] John Vilk and Emery D. Berger. 2018. BLeak repository. https://github.com/plasma-umass/bleak. [Online; accessed 20-March-2018].

[72] Brent Wheeldon. 2017. Unbind events to prevent memory leaks. https://github.com/jeff-collins/ment.io/pull/138. [Online; accessed 8-November-2017].

[73] Guoqing (Harry) Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. 2011. LeakChaser: Helping programmers narrow down causes of memory leaks. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 270–282. https://doi.org/10.1145/1993498.1993530

[74] Guoqing (Harry) Xu and Atanas Rountev. 2013. Precise memory leak detection for Java software using container profiling. *ACM Transactions on Software Engineering and Methodology* 22, 3 (2013), 17:1–17:28. https://doi.org/10.1145/2491509.2491511

[75] Limin Zhu. 2017. Improved JavaScript performance, WebAssembly, and Shared Memory in Microsoft Edge. https://blogs.windows.com/msedgedev/2017/04/20/improved-javascript-performance-webassembly-shared-memory. [Online; accessed 4-November-2017].

A Leaks Found By BLeak

In the next few pages, we document all 59 memory leaks found by BLeak in a separate table per evaluation application. Each memory leak corresponds to a specific source code location that causes unbounded growth; in some cases, multiple memory leaks grow the same leak root or a single memory leak grows multiple leak roots. For each bug, we report the leak root, the type of the leak root, the library responsible for the unbounded growth (Culprit), whether or not the memory leak was previously known (New), if the leaked objects would be considered stale under the assumptions discussed in Section 6.5 (Stale), a link to the bug report, and whether or not the bug has been fixed. A † in the "Fixed" column indicates that a fix is currently under code review, whereas ✓indicates that a fix has already been merged into the codebase. A † in the "New" column indicates that the memory leak was unknown to the application developers, whereas a ✓indicates that the memory leak was unknown to the developers of the culprit library/application.

#	Leak Root	Type	Culprit	New	Stale	Bug Report	Fixed
1	`document.body.childNodes`	DOM	loadCSS [14]	✓		[32]	†
2	'blur' listeners on `window`	EL	Google Maps SDK [20]	†		[16]	
3	'blur' listeners on `window`	EL	Airbnb	✓		Emailed to Airbnb	
4	'resize' listeners on `window`	EL	Google Maps SDK	†		[16]	
5	'click' listeners on `document`	EL	Google Maps SDK	†		[16]	
6	'scroll' listeners on `window`	EL	Google Maps SDK	†		[16]	
7	'scroll' listeners on `window`	EL	Airbnb	✓		Emailed to Airbnb	
8	'keydown' listeners on `document`	EL	Google Maps SDK	†		[16]	
9	'keypress' listeners on `document`	EL	Google Maps SDK	†		[16]	
10	`document.__e3_['keydown']`	Object	Google Maps SDK	†		[16]	
11	'keyup' listeners on `document`	EL	Google Maps SDK	†		[16]	
12	`__e3_['resize']`	Object	Google Maps SDK	†		[16]	
13	`document.__e3_['keyup']`	Object	Google Maps SDK	†		[16]	
14	`document.__e3_['click']`	Object	Google Maps SDK	†		[16]	
15	`__e3_['blur']`	Object	Google Maps SDK	†		[16]	
16	`document.__e3_['keypress']`	Object	Google Maps SDK	†		[16]	
17	'MSFullscreenChange' listeners on `document`	EL	Google Maps SDK	†		[16]	
18	'fullscreenchange' listeners on `document`	EL	Google Maps SDK	†		[16]	
19	'mozfullscreenchange' listeners on `document`	EL	Google Maps SDK	†		[16]	
20	'webkitfullscreenchange' listeners on `document`	EL	Google Maps SDK	†		[16]	
21	`document.__e3_ ['fullscreenchange']`	Object	Google Maps SDK	†		[16]	
22	`document.__e3_ ['mozfullscreenchange']`	Object	Google Maps SDK	†		[16]	
23	`document.__e3_ ['webkitfullscreenchange']`	Object	Google Maps SDK	†		[16]	
24	`document.__e3_ ['MSFullscreenChange']`	Object	Google Maps SDK	†		[16]	
25	`document.head.childNodes`	DOM	Google Tag Manager [22]	✓		[63]	
26	`_xdc_`	Object	Google Maps SDK	✓	✓	[69]	✓
27	'focus' listeners on `window`	EL	Airbnb	✓		Emailed to Airbnb	
28	`ga.h.t0.b.data.keys`	Array	Google Analytics [17]	✓	✓	[68]	
29	`document.body. childNodes[126].childNodes`	DOM	Criteo OneTag Loader [12]	✓		Emailed to Criteo	†
30	`e.extraData` in closure of `criteo_q.push`	Array	Criteo OneTag Loader	✓	✓	Emailed to Criteo	†
31	`A` in closure of `__inner__` property on the second 'popstate' listener of `window`	Array	Airbnb	✓	✓	Emailed to Airbnb	
32	`n['5v9T'].exports._events ['header:search']` within closure of `webpackJsonp`	Array	Airbnb	✓		Emailed to Airbnb	

Figure 11. Memory leaks in Airbnb found by BLeak

#	Leak Root	Type	Culprit	New	Stale	Bug Report	Fixed
33	`jQuery223056319336220622061.events.resize`	Array	Piwik (Data Table)	✓		[58]	✓
34	`jQuery223056319336220622061.events.resize`	Array	Piwik (jqPlot Plugin)	✓		[58]	✓
35	`jQuery223056319336220622061.events.resize`	Array	Piwik (Visitor Map)	✓		[59]	✓
36	bb in closure of `Raphael`	Object	Raphael.js [5]	✓	✓	[57]	†
37	`body.jQuery223056319336220622061.events.mouseup` in closure of `$widgetContent.__proto__.mwheelIntent`	Array	Piwik	✓		[58]	✓
38	`document.body.childNodes`	DOM	Piwik	✓		[59]	✓
39	`allRequests` in closure of `a.piwikApi.withTokenInUrl` in closure of `Ea.jQuery223056319336220622062.$injector.invoke` in closure of `jQuery`	Piwik	Array	✓	✓	[67]	✓
40	`$widgetContent['0'].jQuery223056319336220622062.$scope.$$listeners.$destroy`	Array	Piwik	✓	✓	[62]	
41	`jQuery223056319336220622061.events.click`	Array	Materialize [37]	✓		[65]	✓
42	`piwik.UI.UIControl._controls`	Array	Piwik	✓	✓	[60]	✓
43	Property `jQuery223056319336220622061.events.click` on all `div` children of `#columnPreview`	Array	Piwik	✓		[61]	✓

Figure 12. Memory leaks in Piwik found by BLeak

#	Leak Root	Type	Culprit	New	Stale	Bug Report	Fixed
44	`angular.element.cache[3].events['resize']`	Array	Ment.io [10]	†		[72]	†
45	`angular.element.cache[2].events['click']`	Array	Ment.io	†		[72]	†
46	`angular.element.cache[2].events['paste']`	Array	Ment.io	†		[72]	†
47	`angular.element.cache[2].events['keypress']`	Array	Ment.io	†		[72]	†
48	`angular.element.cache[2].events['keydown']`	Array	Ment.io	†		[72]	†
49	`Loomio.records.discussions.collection.DynamicViews`	Array	Loomio	✓	✓	[66]	✓
50	`angular.element.cache[4].data.$scope.$parent.$$listeners.$translateChangeSuccess`	Array	AngularJS (1.x) [18]	✓	✓	[70]	✓
51	`Loomio.records.stanceChoices.collection.DynamicViews`	Array	Loomio	✓	✓	[66]	✓
52	`Loomio.records.versions.collection.DynamicViews`	Array	Loomio	✓	✓	[66]	✓

Figure 13. Memory leaks in Loomio found by BLeak

#	Leak Roots	Type	Culprit	New	Stale	Bug Report	Fixed
53	`list` in closure of `tuples[0][3].add` in closure of `$.ready.then`, and `list` in closure of `tuples[2][3].add` in closure of `$.ready.then`	Array	jQuery [54]	†	✓	[7]	†
54	`EventLog.eventbindings`	Array	Mailpile	✓		[55]	✓
55	`document.body.childNodes[3].childNodes[3].childNodes`	DOM	Mailpile	✓		[56]	✓

Figure 14. Memory leaks in Mailpile found by BLeak

#	Leak Roots	Type	Culprit	New	Stale	Bug Report	Fixed
56	'mouseover' listeners on `cm.display.wrapper`	EL	Firefox debugger	✓		[64]	✓
57	'mouseup' listeners on `cm.display.wrapper`	EL	Firefox debugger	✓		[64]	✓
58	'mousedown' listeners on `cm.display.wrapper`	EL	Firefox debugger	✓		[64]	✓
59	`cm._handlers.scroll`	Array	Firefox debugger	✓		[64]	✓

Figure 15. Memory leaks in the Firefox debugger found by BLEAK

B BLeak Evaluation Application Loops

This section lists the code for all of the loops used in the evaluation (§6). These scripts are the only input BLeak needs to automatically locate, rank, and debug the memory leaks from the evaluation. Note that the line counts reported in Figure 9 ignore comment lines.

```
exports.loop = [{
  check: function() {
    const buttons = document.getElementsByTagName('button');
    return document.getElementsByTagName('a')[4].getAttribute('aria-selected') === 'true';
  },
  next: function() {
    document.getElementsByTagName('a')[5].click();
  }
}, {
  check: function() {
    const buttons = document.getElementsByTagName('button');
    return document.getElementsByTagName('a')[5].getAttribute('aria-selected') === 'true' && buttons.length > 11 &&
     ↪  buttons[11].innerText.trim() === "Room type"
  },
  next: function() {
    document.getElementsByTagName('a')[4].click();
  }
}];
```

Figure 16. Airbnb's loop.

```
exports.login = [
  {
    check: function() {
      const input = document.getElementsByTagName('input');
      const username = input[0];
      const password = input[2];
      const submit = document.getElementsByClassName('submit')[0];
      return !!(username && password && submit);
    },
    next: function() {
      const input = document.getElementsByTagName('input');
      const username = input[0];
      const password = input[2];
      const submit = document.getElementsByClassName('submit')[0];
      username.value = "bleak";
      password.value = "bleakpldi";
      submit.click();
    }
  }
];
exports.loop = [
  {
    check: function() {
      const svg = document.getElementsByTagName('svg');
      const canvas = document.getElementsByTagName('canvas');
      return svg.length === 1 && canvas.length === 42;
    },
    next: function() {
      document.getElementsByClassName('item')[1].click();
    }
  }
];
```

Figure 17. Piwik's loop.

```
exports.login = [{
  check: function() {
    const emailField = document.getElementsByTagName('input')[1];
    if (emailField) {
      return emailField.getAttribute('name') === 'email';
    }
    return false;
  },
  next: function() {
    const emailField = document.getElementsByTagName('input')[1];
    emailField.value = 'default@loomio.org';
    // Notify Angular code of change.
    emailField.dispatchEvent(new Event("change"));
    const submitBtn = document.getElementsByTagName('button')[2];
    submitBtn.click();
  }
}, {
  check: function() {
    const pswdField = document.getElementsByTagName('input')[1];
    const modalHeader = document.getElementsByTagName('h2')[3];
    const submitBtn = document.getElementsByTagName('button')[3];
    return submitBtn && pswdField && pswdField.name === "password" && modalHeader && modalHeader.innerText === "Welcome back, default@loomio.org!" && submitBtn.innerText === "SIGN IN";
  },
  next: function() {
    const pswdField = document.getElementsByTagName('input')[1];
    pswdField.value = 'b0eb3a48';
    pswdField.dispatchEvent(new Event("change"));
    const submitBtn = document.getElementsByTagName('button')[3];
    submitBtn.click();
  }
}];
exports.setup = [{
  check: function() {
    const tp = document.getElementsByClassName('thread-preview');
    if (tp.length > 0) {
      const thread = tp[0];
      return thread.childNodes.length > 0 && thread.childNodes[0].tagName === "A" && thread.childNodes[0].getAttribute('href') === "/d/6jZ4c8FL/how-to-use-loomio";
    }
    return false;
  },
  next: function() {
    document.getElementsByTagName('md_icon_button')[0].click();
  }
}];
exports.loop = [{
  check: function() {
    const span = document.getElementsByTagName('span')[6];
    return !!span && span.innerText === "Fun Group 1";
  },
  next: function() {
    document.getElementsByTagName('span')[6].click();
  }
}, {
  check: function() {
    const tp = document.getElementsByClassName('thread-preview');
    if (tp.length > 0) {
      const thread = tp[0];
      return thread.childNodes.length > 0 && thread.childNodes[0].tagName === "A" && thread.childNodes[0].getAttribute('href') === "/d/6jZ4c8FL/how-to-use-loomio";
    }
    return false;
  },
  next: function() {
   document.getElementsByClassName('thread-preview')[0].childNodes[0].click();
  }
}, {
  check: function() {
    const paragraphs = document.getElementsByTagName('p');
    const h3 = document.getElementsByTagName('h3')[3];
    return paragraphs.length > 6 && h3 && h3.innerText.indexOf("Loomio Helper Bot started a proposal") === 0 && paragraphs[5].innerText === "Welcome to Loomio, an online place to make decisions together.";
  },
  next: function() {
    // Opens menu w/ logout.
    document.getElementsByTagName('md_icon_button')[0].click();
  }
}];
```

Figure 18. Loomio's loop.

```
const emailSubjects = [ "YO DAWG", "Icelandic Banana", "Demo ipsum",
  "CRYPTO-GRAM, January 15, 2014"];
function returnToInbox() {
  document.getElementById('sidebar-tag-b').children[0].click();
}
function itemSteps(i) {
  let lastCheck = 0;
  function inboxCheck() {
    return document.getElementsByClassName('message-subject').length === 0 &&
    ↪  document.getElementsByClassName('item-subject').length > 0;
  }
  function inboxClick() {
    document.getElementsByClassName('item-subject')[i].click()
  }
  return [{
    check: inboxCheck,
    next: inboxClick
  }, {
    check: function() {
      const ms = document.getElementsByClassName('message-subject');
      const rv = ms.length === 1 && ms[0].innerText === emailSubjects[i];
      const now = Date.now();
      // Mailpile's server fails a *lot*, requiring multiple clicks to get through.
      // This is a hack to get around what I consider to be a bug.
      if (!rv && inboxCheck() && (now - lastCheck) > 1000) {
        inboxClick();
      }
      lastCheck = now;
      return rv;
    },
    next: function() {
      lastCheck = 0;
      returnToInbox();
    }
  }];
}
exports.loop = [];
for (let i = 0; i < emailSubjects.length; i++) {
  exports.loop.push.apply(exports.loop, itemSteps(i));
}
```

Figure 19. Mailpile's loop.

```
exports.loop = [{
  check: function() {
    const nodes = document.getElementsByClassName('node');
    const sourceTabs = document.getElementsByClassName('source-tab');
    return sourceTabs.length === 0 && nodes.length > 1 && nodes[1].innerText === "main.js";
  },
  next: function() {
    document.getElementsByClassName('node')[1].click();
  }
}, {
  check: function() {
    // code mirror must be open, tab must be added, etc.
    return document.getElementsByClassName('CodeMirror-line').length > 2 && document.getElementsByClassName('source-tab').length === 1 && document.getElementsByClassName('close-btn').length === 1;
  },
  next: function() {
    document.getElementsByClassName('close-btn')[0].click();
  }
}];
```

Figure 20. Firefox Debugger's loop.