DOI:10.1145/3422598

BLeak: Automatically Debugging Memory Leaks in Web Applications

By John Vilk and Emery D. Berger

Abstract

Memory leaks in web applications are pervasive and difficult to debug. Leaks degrade responsiveness by increasing garbage collection costs and can even lead to browser tab crashes. Previous leak detection approaches designed for conventional applications are ineffective in the browser environment. Tracking down leaks currently requires intensive manual effort by web developers, which is often unsuccessful.

This paper introduces BLeak (Browser 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 0.5MB to 8MB per round trip.

1. INTRODUCTION

Browsers are one of the most popular applications on both smartphones and desktop platforms. They also have an established reputation for consuming significant amounts of memory. To address this problem, browser vendors have spent considerable effort on shrinking their browsers' memory footprints[5, 11] and building tools that track the memory consumption of specific browser components.[4, 10]

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.

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,[2, 6, 12, 14, 16] but web applications regularly interact with leaked state (e.g., via event listeners). Growth-based techniques assume that leaked objects are uniquely owned or form strongly connected components in the heap graph.[9, 16] 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`).

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.[1, 8] Unfortunately, these snapshots do not necessarily provide actionable information (see Section 2.1). They simultaneously provide too much information (every 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 labeled as objects or arrays, which provides little assistance in locating leak sources. The result is that even expert developers are unable to find leaks: for example, a Google developer closed a Google Maps SDK leak (with 117 stars and 62 comments) because it was "infeasible" to fix as they were "not really sure in how many places [it's] leaking".[1]

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*

[1] https://issuetracker.google.com/issues/35821412.

The original version of this paper appeared in the *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA, June 18–22, 2018), 15–29.

oracle to identify leaks. Each time a web application returns to the same visual state, it should consume approximately the same amount of memory. Sustained memory growth across round trips is thus a clear indicator of a memory leak. BLEAK builds directly on this observation to find memory leaks in web applications, which (as Section 6 shows) are 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 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 leaks first.

Guided by BLEAK, we identify and fix over 50 memory leaks in popular JavaScript libraries and applications including Airbnb, AngularJS, jQuery, Google Analytics, and 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 0.5MB to 8MB 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.

This paper makes the following contributions:

- It introduces novel techniques for automatically locating, diagnosing, and ranking memory leaks in web applications (Section 3) and presents algorithms for each (Section 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) (Section 5).
- Using BLEAK, we identify and fix numerous leaks in widely used web applications and JavaScript libraries (Section 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 tools fall short when debugging leaks in web applications.

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

Figure 1. This code from Firefox's debugger (truncated for readability) leaks 0.5MB every time a developer opens a source file (Section 2). BLEAK finds all four leaks automatically.

```
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 }
```

2.1. Leak debugging via heap snapshots

There are currently no automated techniques for identifying memory leaks in web applications. 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.

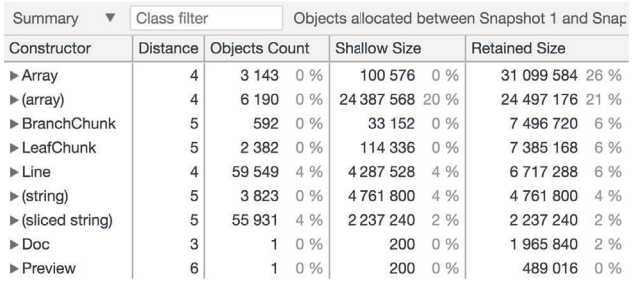
The most popular way to manually debug memory leaks is via the *three heap snapshot technique* introduced by the Gmail team.[8] Developers repeat a task twice on a webpage and examine still-live objects created from the first run of the task. The assumption is that each run will clear out most of the 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 reopening 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, as 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 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

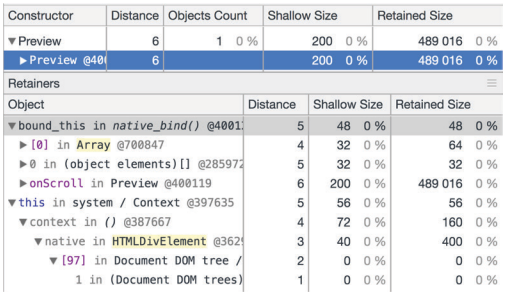
Figure 2. The manual memory leak debugging process: Currently, developers debug leaks by first examining heap snapshots to find leaking objects (a). Then, they try to use retaining paths to locate the code responsible (b). Unfortunately, these paths have no connection to code, so developers must search their codebase for identifiers referenced in the paths (see Section 2.1). 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.



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 nonleaking 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.

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.

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.

During 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* (Section 4.3) that prioritizes memory leaks that free the most memory with the least effort. LeakShare prunes objects in the graph reachable by nonleak 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.

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 (a). BLeak then runs automatically, producing a ranked list of memory leaks with stack traces pointing to the code responsible for the leaks (b).

```
exports.loop = [// Repeatedly open and close a source document.
{ // 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(); }
}, { // 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.

906358432530

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. With this information in hand, we were able to quickly develop a fix that removes the event listeners when the user closes the document. This fix has been incorporated into the latest version of the debugger.

4. ALGORITHMS

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

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.

```
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))
```

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,* "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 ($G.root$).[2]

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

[2] For simplicity, we describe heap graphs as having one root.

initializes every node in the new graph to *not growing* to 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:

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*).

```
FindLeakPaths(G)
 1  Q = [], T_Gr = {}
 2  for each edge e = (n1, n2, l) ∈ G.E where n1 == G.root
 3       e.mark = TRUE
 4       Enqueue(Q, (NIL, e))
 5  while |Q| > 0
 6       t = Dequeue(Q)
 7       (t_p, (n1, n2, l)) = t
 8       if n2.growing == TRUE
 9            T_Gr = T_Gr ∪ {t}
10       for each edge e = (n1', n2', l') ∈ G.E
11            if n1' == n2 and e.mark == FALSE
12                 e.mark = TRUE
13                 Enqueue(Q, (t, e))
14  return T_Gr
```

- *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*. LeakShare 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.

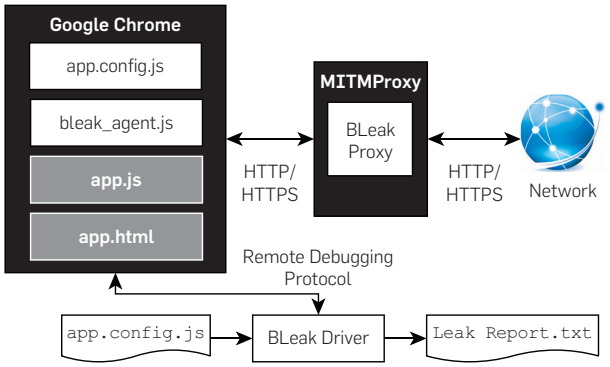
LeakShare first marks all of the items in the heap that are reachable from nonleaks via a breadth-first traversal that stops at leak roots. These nodes are ignored by subsequent traversals. Then, LeakShare performs a breadth-first traversal from each leak root that increments a counter on all reachable nodes. 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 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. Our PLDI paper presents the full algorithm for LeakShare.[15]

5. IMPLEMENTATION

BLEAK consists of three main components that work together to automatically debug memory leaks (see Figure 6): (1) a driver program orchestrates the leak debugging process; (2) a proxy transparently performs code rewriting on-the-fly on the target web application; and (3) an agent script embedded within the application exposes hidden state for leak detection and growth events for leak diagnosis. We briefly describe how these components work here; our PLDI paper provides further details.[15]

To initiate leak debugging, the BLEAK driver launches BLEAK's proxy and 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, navigates to the target web application, and uses the developer-provided configuration file to drive the application in a loop. During each repeat visit to the first visual state in the loop, the driver takes a heap snapshot via the remote debugging protocol

Figure 6. BLeak system overview. White items are BLeak components, gray items are rewritten by the proxy during leak diagnosis, and black items are unmodified.



and runs PropagateGrowth (Figure 4) to propagate growth information between heap snapshots.

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, 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. Then, the driver runs the application in a final loop before retrieving stack traces from the agent. Finally, the driver runs LeakShare (Section 4.3) to rank leak roots and generate a memory leak report.

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? (Section 6.2)
- **Accuracy of diagnoses:** Does BLeak accurately locate the code responsible for memory leaks? (Section 6.2)
- **Impact of discovered leaks:** How impactful are the memory leaks that BLeak finds? (Section 6.3)
- **Utility of ranking:** Is LeakShare an effective metric for ranking the severity of memory leaks? (Section 6.4)

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.

We run BLeak on each web application for 8 round trips through specific visual states to produce a BLeak leak report, as shown in Figure 3b. We describe these loops using only 17–73 LOC. This process takes less than 15 min per application on our evaluation machine, a MacBook Pro with a 2.9GHz 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: 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: 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: 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: An open-source mail client. Mailpile uses jQuery. BLeak runs Mailpile's demo in a loop that visits the inbox and the first four emails in the inbox.

Firefox Debugger (commit 91f5c63): 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. BLeak runs the debugger in a loop that opens and closes SensorWeb's `main.js` in the debugger's text editor.

6.2. Precision and accuracy

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. Figure 8 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 identify 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.** Each memory leak took approximately 15 min to fix.

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 Section 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. Figures 7 and 8 present the results.

On average, fixing the memory leaks that BLeak reports eliminates over 93% of all heap growth on our benchmarks (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. The transitive closure size of a leak root is the size of all objects reachable from the leak root as used by Xu et al.[16] 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.

We measure the effectiveness of each ranking metric by calculating the growth reduction (as in Section 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; some leaks share the same root cause. 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

Figure 7. 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.

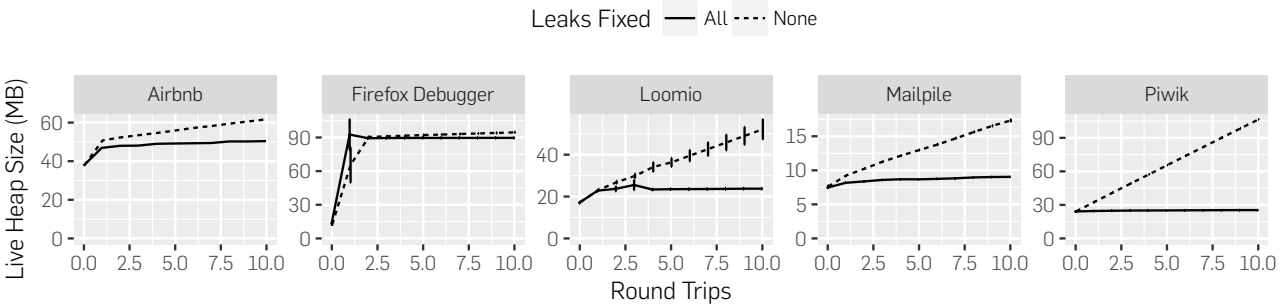


Figure 8. BLeak precisely finds impactful memory leaks: On average, BLeak finds these leaks with over 95% precision, and fixing them eliminates over 90% of all heap growth.

Program	Loop LOC	Leak Roots	False Positives	Distinct Leaks	Precision	Growth Reduction
Airbnb	17	32	2	32	94%	1.04 MB (81.0%)
Piwik	32	17	0	11	100%	8.14 MB (99.3%)
Loomio	73	10	1	9	90%	2.83 MB (98.3%)
Mailpile	37	4	0	3	100%	0.80 MB (91.8%)
Firefox Debugger	17	4	0	4	100%	0.47 MB (98.2%)
Total / mean:	35	67	3	59	96.8%	2.66 MB (93.7%)

Figure 9. Performance of ranking metrics: Growth reduction by metric after fixing quartiles of top-ranked leaks. Bold indicates greatest reduction (±1%). We omit Firefox; it has only four leaks, which must all be fixed (see Section 2). LeakShare generally outperforms or matches other metrics.

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

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 9 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.

7. RELATED WORK

Web application memory leak detectors: BLEAK automatically debugs memory leaks in web applications; past work in this space is ineffective or not sufficiently general. LeakSpot locates allocation and reference sites that produce and retain increasing numbers of objects over time and uses staleness as a heuristic to refine its output.[14] On real applications, LeakSpot typically reports over 50 different allocation and reference sites that developers must manually inspect to identify and diagnose memory leaks. JSWhiz statically analyzes code written with Google Closure type annotations to detect specific leak patterns.[13]

Web application memory debugging: Some tools help web developers debug memory usage and present diagnostic information that developers must manually interpret to locate leaks (Section 2 describes Google Chrome's DevTools). MemInsight summarizes and displays information about the JavaScript heap, including per-object-type staleness information, the allocation site of objects, and retaining paths in the heap.[7] 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.[9] 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.

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.[6, 2, 12] Leakpoint (C/C++) also identifies the last point in the execution that referenced a leaking memory location.[3] 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. As we discuss in our PLDI paper, staleness is ineffective for at least 77% of the memory leaks BLEAK identifies.[15]

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 and is available for download at http://bleak-detector.org/.

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. Basques, K. Fix memory problems, 2017. https://developers.google.com/web/tools/chrome-devtools/memory-problems/.
2. Bond, M.D., McKinley, K.S. Bell: Bit-encoding online memory leak detection. In *ASPLOS*, ACM, San Jose, CA, 2006, 61–72.
3. Clause, J.A., Orso, A. LEAKPOINT: Pinpointing the causes of memory leaks. In *ICSE*, ACM, Cape Town, South Africa, 2010, 515–524.
4. Google. Speed up google chrome, 2017. https://support.google.com/chrome/answer/1385029.
5. Hara, K. Oilpan: GC for blink, 2013. https://docs.google.com/presentation/d/1YtfurcyKFS0hxPOnC3U6JJroM8aRP49Yf0QWznZ9jrk.
6. Hauswirth, M., Chilimbi, T.M. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, ACM, Boston, MA, 2004, 156–164.
7. Jensen, S.H., Sridharan, M., Sen, K., Chandra, S. MemInsight: Platform-independent memory debugging for JavaScript. In *FSE*, ACM, Bergamo, Italy, 2015, 345–356.
8. Lee, L., Hundt, R. BloatBusters: Eliminating memory leaks in Gmail, 2012. https://docs.google.com/presentation/d/1wUVmf78gG-ra5aOxvTfYdiLkdGaR9OhXRnOlIcEmu2s.
9. Mitchell, N., Sevitsky, G. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *ECOOP*, 2003, 351–377.
10. Mozilla. about: memory, 2017. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/about:memory.
11. Nguyen, N. The best firefox ever, 2017. https://blog.mozilla.org/blog/2017/06/13/faster-better-firefox/.
12. Novark, G., Berger, E.D., Zorn, B.G. Efficiently and precisely locating memory leaks and bloat. In *PLDI*, ACM, Dublin, Ireland, 2009, 397–407.
13. Pienaar, J.A., Hundt, R. JSWhiz: Static analysis for JavaScript memory leaks. In *CGO*, IEEE Computer Society, Shenzhen, China, 2013, 11:1–11:11.
14. Rudafshani, M., Ward, P.A.S. Leakspot: Detection and diagnosis of memory leaks in javascript applications. *Softw. Pract. Exp. 1*, 47 (2017), 97–123.
15. Vilk, J., Berger, E.D. BLEAK: Automatically debugging memory leaks in web applications. In *PLDI*, ACM, Philadelphia, PA, 2018, 15–29.
16. Xu, G.H., Rountev, A. Precise memory leak detection for Java software using container profiling. *TOSEM 3*, 22 (2013):17:1–17:28.

John Vilk and Emery D. Berger ([jvilk, emery]@cs.umass.edu), College of Information and Computer Sciences, University of Massachusetts Amherst.