JSWhiz

Static Analysis for JavaScript Memory Leaks

Jacques A. Pienaar
Purdue University
jpienaar@purdue.edu

Robert Hundt
Google, Inc.
rhundt@google.com

Abstract

JavaScript is the dominant language for implementing dynamic web pages in browsers. Even though it is standardized, many browsers implement language and browser bindings in different and incompatible ways. As a result, a plethora of web development frameworks were developed to hide cross-browser issues and to ease development of large web applications. An unwelcome side-effect of these frameworks is that they can introduce memory leaks, despite the fact that JavaScript is garbage collected. Memory bloat is a major issue for web applications, as it affects user perceived latency and may even prevent large web applications from running on devices with limited resources.

In this paper we present JSWhiz, an extension to the open-source Closure JavaScript compiler. Based on experiences analyzing memory leaks in Gmail, JSWhiz detects five identified common problem patterns. JSWhiz found a total of 89 memory leaks across Google's Gmail, Docs, Spreadsheets, Books, and Closure itself. It contributed significantly in a recent effort to reduce Gmail memory footprint, which resulted in bloat reduction of 75% at the 99th percentile, and by roughly 50% at the median.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers; D.3.4 [*Processors*]: Optimization

General Terms Performance, Static Analysis

Keywords Optimization, Performance, Static Analysis, Memory Leak, JavaScript

1. Introduction

JavaScript is the dominant language for implementing dynamic web pages in browsers. It is the "Lingua Franca" of the web. It has been standardized [14] and is supported by all major web browsers, such as Microsoft's Internet Explorer, Mozilla's Firefox, Apple's Safari, Google's Chrome, and many others. With JavaScript's execution speed accelerating, efforts such as `node.js` [1] promote using JavaScript in servers as well.

Despite the fact that JavaScript and its semantics have been standardized, the various JavaScript engines and their embedding browsers implement important parts of JavaScript in incompatible ways, a problem that has burdened the web development community for years. As a result, a whole range of JavaScript development environments were developed to ease development of larger JavaScript web applications and to hide cross-browser differences. Prominent libraries such as JQuery [10] and Dojo [24], as well as frameworks such as Google Web Toolkit [12, 15] or Google's Closure [8] are widely used and applications written with these toolsets are being used by hundreds of millions of users every day. Google Closure stands out as it contains a JavaScript compiler as well as library code. The (open-source) Closure compiler is based on Mozilla's Rhino [21]. It accepts legal JavaScript as input and produces JavaScript as output. It contains a JavaScript comment-based type system to aid large-scale developments. The type system is also helpful in the context of this paper. The compiler offers many optimizations targeting performance, minification and obfuscation, which are all important for web applications.

An interesting and unwelcome side-effect of these frameworks' abstractions and wrappers is that they can introduce memory leaks, despite the fact that JavaScript is garbage collected. As with any garbage collected language, if references to otherwise dead objects exist, the objects are reachable and cannot be reclaimed. In browsers, a simple example could be that a group of DOM nodes is managed as a "view", and the application maintains a (badly managed) cache of these views. Since there are JavaScript objects maintaining references to DOM nodes, those can never be freed, unless the references from JavaScript are set to null as well.

A specifically common problem in browsers is being introduced by the abstractions to handle the incompatible native browser event systems. Typically, the frameworks provide their own abstractions and wrap the native event systems to hide the browser differences. Often, global reg-

CGO'13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

istries are being used where references to events and event targets are being maintained and have to be explicitly released to avoid leaks. This is similar to, e.g., the C++ `new` and `delete` memory allocation API requiring matching calls. This situation is further complicated by modern, dynamically compiled, browser-embedded execution engines. Compiled JavaScript living in an internal code cache can hang on to DOM nodes which are allocated in the rendering parts of the browser. Such problems are typically very hard to find, specifically because of insufficient tool support in browsers. Forgetting to manually "unlisten" to events can lead to leaking DOM nodes and unlimited memory growth.

Memory bloat for web applications is big problem, it impacts, as we shall show, user experienced latency, and can prevent larger, long running web applications from even running on smaller devices. In the context of Gmail, we studied the impact of memory bloat itself, as well its impact on user experienced latency.

The Gmail team started an effort to reduce memory bloat and identified and fixed several dozen problems in a labor intensive, manual process. Since Gmail utilizes the Closure compiler, the question was whether it would be possible to automate finding of these problems with help of static analysis in the compiler.

The results presented in this paper are specific to the use of the Closure library and mostly to the event system abstractions made in this framework. However, we want to emphasize that the problem itself is general—any framework that uses auxiliary data structures to maintain certain objects' lifetimes is susceptible to introducing memory leaks from mis-matched API calls. For example, the jQuery framework has several leak patterns [3], and both the Dojo framework and the YUI framework suffer from leak patterns in the event system [2, 4]. We focus on the Closure specific leak patterns, but believe that many of these other patterns can be found with static analysis similar to the one presented here. We make the following main contributions:

- We show the correlation of memory bloat and latency degradations for web applications;
- We identify seven patterns that introduce memory leaks in JavaScript; these examples are Closure specific, but, again, the conceptual problems are general.
- We developed compiler passes to identify five of them reliably, without any false positives. This implementation will be released open-source with Google Closure.

Using the new compiler passes, we found 89 memory leaks across several large Google applications, such as Gmail, Docs, Spreadsheets, Presentations, Books, and the Closure libraries. Together with a range of improvements in Chrome's garbage collection itself, fixing the issues contributed to the majority of the memory reduction for Gmail, which were more than 75% at the 99th percentile, and approximately 50% at the 50th percentile.

The rest of this paper is organized as follows. We show a correlation between memory bloat and latency degradation in Section 2. We describe the current manual labor intensive process to find memory leaks in Section 3. In this section we also describe in detail the identified seven problem patterns and their resolution. We then discuss the static memory leak identification tool we have developed in Section 4. We discuss results in Section 5, before we present related work in Section 6. We conclude in Section 7.

2. Impact of Memory Bloat on Latency

The Gmail team regularly got complaints from users about massive size Gmail processes. 2GB were not uncommon, some users reported memory usage of over 10GB. For mobile devices, such as Chromebooks or mobile tablets, which typically have less physical memory available than desktop computers, Gmail would consume more memory than available and be terminated on a regular basis.

To further study the impact of memory bloat on latency, we ran a simple Gmail scenario (compose/discard) repeatedly for many hours. In Figure 1 the x-axis denotes time, 21,056 seconds for the whole experiment. The left y-axis shows the latency in Milli-seconds, from 0ms to over 2,000ms. The right y-axis shows the number of DOM nodes, which linearly increases in this experiment to up to over 2,000,000 nodes.

As the number of DOM nodes continues to grow into the millions because of Gmail memory leaks, we can see that the magnitude of the latency outliers also increases. An operation that took just about 100ms in the beginning can take more than 2 seconds at the end of the experiment. This can be explained by the effects of garbage collection. In order to detect detached DOM trees the browser has to traverse all nodes, an operation that scales linearly to the number of DOM nodes.

The effect of this can be observed for real individual user accounts as well. Figure 2 shows how memory grows for a single user over the course of 4 days. Starting at only a few hundred MB, memory grows to roughly 1.5GB. Figure 3 shows the distribution of latency for this user. The blue dots represent user requests that were done during the day, "empty" white space correspond to this user's inactive night time. Latency variation increases greatly with memory bloat. Latency outliers grow in size from roughly 2 seconds to over 12 seconds.

3. Memory leak patterns

Garbage collectors used by the modern browsers are able to reclaim memory in a great assortment of varied and complex situations, but unfortunately not in all. In this section we present instances where developers must explicitly manage objects in order to avoid leaks. The discussion is Closure specific. However, as mentioned earlier, the problem is general and can be found in any situation where auxiliary

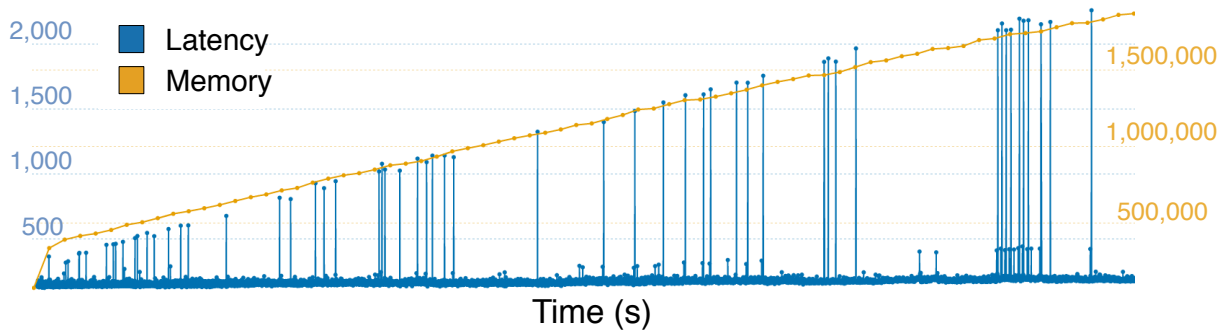

Figure 1. Relationship between memory bloat and latency outliers. The left y-axis is time measured in $[ms]$. The right y-axis counts the number of DOM nodes, which correlates directly to memory usage. The x-axis is the running time of the experiment, which was several hours.

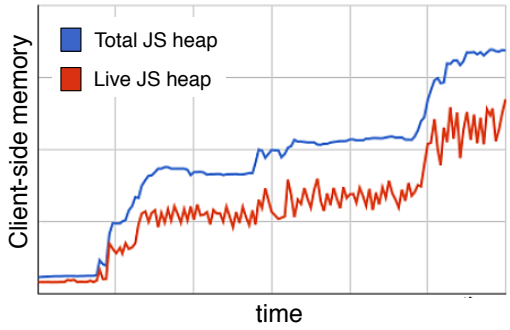

Figure 2. Memory growth for a single user over 4 days after starting Gmail. The y-axis shows memory growing up to 1.5GB over those 4 days

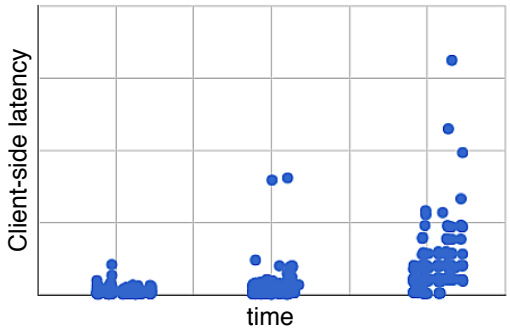

Figure 3. Increasing latency variability and magnitude caused by memory bloat during the same time span as in Figure 2. The y-axis denotes time per request in $[sec]$, which ranges from 0 up to 12 seconds.

data structures are being managed via matching API calls. This pattern is very common and we find similar problems in most of the major JavaScript development frameworks.

We refer to any memory that can not be reclaimed by the garbage collector, but that does not serve any further purpose in the business function of the application, as a memory leak.

3.1 Finding leaks

The memory leak patterns identified were the result of an intensive and laborious manual code review. Memory usage monitoring is enabled in the Gmail pre-release test suite, ran for many hours at a time and the outcome of these tests were used to direct the discovery of potential leaks. Once a sequence of actions were identified that could lead to a leak (e.g., clicking on Compose followed by clicking on a label) the Chrome developer tools were used to triage the leak.

The Chrome Developer Tools Team suggested a three snapshot approach to narrow down the source of memory leaks. This approach, summarized in Figure 4, consists of

- Taking a heap snapshot #1;
- Performing the suspected memory leak triggering actions x number of times;
- Taking another heap snapshot #2;
- Performing exactly the same actions x times again;
- Taking a final heap snapshot #3; and
- Filtering all objects allocated between snapshot 1 and 2 in snapshot 3. These objects are likely leaks. If objects appear in multiples of x, the likelihood becomes almost certainty.

As can be seen from the above description, finding leaks is not a straight forward process, especially for application without exhaustive set of tests exercising the UI. The process is labor intensive, there is a high chance that leaks are being missed, leaks can only be found at runtime, not as early as compile time, and it will only work on regions of code that were actually executed.

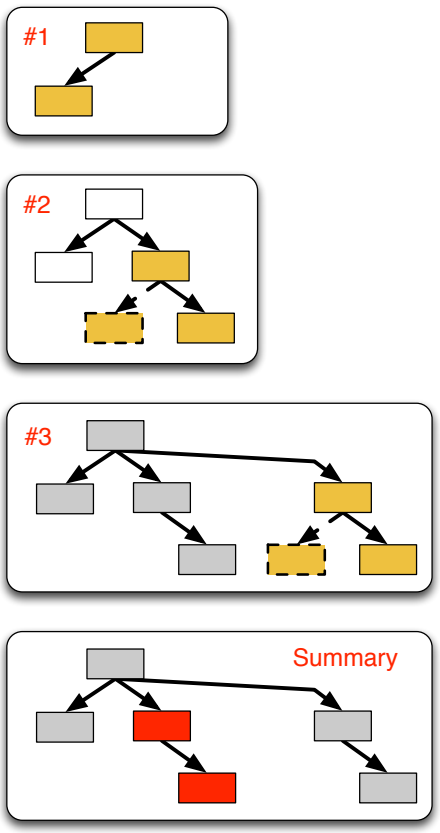

Figure 4. The three snapshot approach to triaging memory leaks.

In the following subsections we outline different typical problem patterns, explain why they are introducing leaks, and suggest solutions.

3.2 Create without dispose EventHandler

Closure objects of type `EventHandler` provide a simple mechanism to group all events and listeners associated with an object together. This allows for easy removal of listeners when the events being listened to can no longer fire. Unfortunately this is also one of the main sources of leaks.

Consider the following scenario. We wish to pop-up a new dialog when the user presses a button. The DOM tree representing the new dialog is created and an event listener is added to the dialog containing the button, associating the button click event with displaying the dialog. Here, a link (reference) is created between the DOM object being listened to, the DOM tree of the new dialog, the JavaScript object model of the containing dialog and the EventHandler.

If the button can no longer be clicked, say the dialog has been destroyed, then that event can no longer fire. But if the programmer did not dispose of the EventHandler (e.g., by calling the Closure `dispose()` function), then the DOM tree corresponding to the new dialog is still referenced and cannot be reclaimed.

```
X.prototype.maybeEnable = function() {
 if (this.user_.hasFeature(Y)) {
  var eventHandler = new goog.events.↩
      EventHandler();
  this.setEventHandler(eventHandler);
 }
};
```

Listing 1. Create without dispose EventHandler.

In Listing 1 we present an example of this pattern. In the actual code this example was derived from, class `X` was a UI element derived from type `goog.Disposable` which made fixing this leak very easy: the lifetime of the EventHandler was linked to the lifetime of the UI element by registering `eventHandler` as disposable.

This example does also introduce one of the difficulties with this analysis: What does the member function `setEventHandler` do with `eventHandler`? We shall return to this problem in Section 4.

This problem corresponds directly to the classic memory allocation pattern in C++, which requires matching `new` and `delete` calls. But, as JavaScript is garbage collected, programmers are not necessarily aware of the requirements for cleanup when disposing objects and simply forget, or overlook, the need to make those matching calls.

3.3 Setting member to null without removing event listeners

In JavaScript setting a variable to `null` is the idiomatic way of providing a hint to the garbage collector that the memory allocated by an object can be reclaimed. But the EventHandler and event listeners attached to the object prevents the garbage collector from disposing of the object and reclaiming memory.

In all instances where the developer wishes to free an EventHandler, he/she has to explicitly dispose of the object rather than just setting the variable to `null`. From the observed problem cases, we believe that programmers were not fully aware of the explicit requirement to dispose of objects for cleanup. Instead they relied on the garbage collector, which is not able to free all resources ad listeners, which led to memory leaks.

3.4 Undisposed member object has EventHandler as member

In this pattern an object is created which creates an EventHandler that is only disposed when the created object is.

```
/** @constructor */
X = function() {
  /** @type {!goog.events.EventHandler} */
  this.eventHandler = new goog.events.↩
      EventHandler;
  ...
}
X.prototype.dispose = function() {
```

```
 goog.dispose(this.eventHandler);
}
function() {
  ...
   var dragListGroup = new X();
  ...
}
```

Listing 2. Object created has EventHandler as member.

In Listing 2 `dragListGroup` is an instance of type `X` which is a class that has an EventHandler, `eventHandler`, as a member. This EventHandler object only gets disposed of when the instance of `X` that instantiated it does, but in this snippet the local variable `dragListGroup` never gets disposed of.

From the manual analysis of these cases, we believe programmers simply overlook or are not aware of the need to explicitly call `dispose()` on these objects.

3.5 Object graveyard

We refer to an object graveyard where objects with an EventHandler or event listeners attached, are added to an array/list and remain in this data structure long past their lifetime/use. In some of these cases the containing data structure is traversed and these objects disposed of, but just because it will get disposed of eventually, does not mean it won't result in memory bloat.

```
X.prototype.addNotification = function(t) {
  if (!goog.array.contains(onInvalidate_, t)) {
   this.onInvalidate_.push(t);
 }
};
```

Listing 3. Object graveyard.

This pattern is quite problematic, as it also corresponds to many valid usage patterns. In fact, without further semantic information as to the lifetime or use of the objects being stored into the array, one could not determine if code corresponding to the pattern was a leak or not.

From the manual analysis of the actual problem case we found that these cases happened in a complicated, convoluted, and large cache management system with an unclear object ownership model. Because of the source complexity, these problems were very hard to spot.

3.6 Local EventHandler

A local EventHandler instance that does not escape scope, e.g., a locally defined variable that is not assigned to a field member, added to an array, or captured in a closure, can not be disposed of later.

```
A.B = function(...) {
 ...
 // Listen for browser resize events.
 var handler = new goog.events.EventHandler(↩
     this);
 ...
}
```

Listing 4. Local EventHandler.

A common instance of this pattern is locally created EventHandler objects which listen to events using the member function `listenOnce`. The `listenOnce` call attaches an event listener to an object such that event listener is removed if the trigger action does occur. Conceptually this avoids memory leaks as the event is automatically unlistened to and the object can be reclaimed by the garbage collector. But this is only true if *the event is guaranteed to occur*. If the event never occurs then the event listener is never removed and the object is never reclaimed.

3.7 Overwriting EventHandler field in derived class

Closure implements inheritance using a prototype-based approach [8], which can break the expectations of programmers coming from a more standard OOP language background (such as C++ or Java). For EventHandlers this can cause memory leaks as the overwritten EventHandler cannot be freed [8].

In such instances the developer should either use an accessor function to access the member of the base class, or use a different member name for the variable. For EventHandlers the former is normally desirable.

This problem is very tricky and extremely hard to spot in code reviews because of the distributed location of base and derived classes, as well as JavaScript inheritance rules.

3.8 Unmatched listen/unlisten calls

The semantics of the listen and unlisten calls require all parameters to be the same. When the parameters do not match, the event listener is not removed and a reference to objects being listened remains, inhibiting the garbage disposal. Unlistening to an event using a (newly created) bound or anonymous function as callback function does not work as intended. For a listen/unlisten to match, all parameters in the calls must match, but two anonymous functions are different and the result of `goog.bind` (a bound function) is a new object. In both cases the calls won't form a matching listen/unlisten pair.

```
goog.events.listen(dialog, goog.ui.Dialog.↩
    EventType.SELECT, goog.bind(this.↩
    onContactSelectorClose_, this, dialog, ↩
    callback));
...

goog.events.unlisten(dialog, goog.ui.Dialog.↩
    EventType.SELECT, goog.bind(this.↩
    onContactSelectorClose_, this, dialog, ↩
    callback));
```

Listing 5. Unmatched listen/unlisten calls due to bound functions.

Each listen call need not have an unlisten call as Closure also provides the `goog.events.removeAll` function, which removes all events associated with an object (EventHandler objects have similarly a `removeAll` call). Furthermore, objects of type `goog.events.EventTarget` removes all event lis-

teners associated with them when disposed of. But an unlisten call using an anonymous/bound function as callback function does nothing.

This problem is also hard to spot in code reviews, because the arguments to the `listen` and `unlisten` calls appear to match. If one is not aware that the call to `bind` returns a new object on each invocation, it is easy to introduce these leaks.

3.9 Avoiding memory leaks

Modern garbage collectors are capable of handling numerous complicated scenarios. The above patterns serve as an indication of what to avoid. Fundamentally the garbage collector has to be conservative, it can never reclaim an object's memory that could be referenced later. In all of these patterns the programmer/library inhibited the reachability analysis of the garbage collector.

JavaScript application and library developers introduce additional abstractions for compatibility across different systems. But these abstractions, as well as legacy systems/browsers, can lead to the garbage collector being inhibited. This requires the developer to think about the lifetime of objects within his/her system, and where possible associate the lifetime of an EventHandler with that of an object which is disposed of when the events being listened to can no longer occur. For example, associating the lifetime of an EventHandler listening to button click to the lifetime of that button or to the dialog containing that button. The W3C DOM events model encourages a direct association between the object being listened to and the event listeners. This gives the garbage collector better visibility into the lifetime of these objects.

4. Statically detecting memory leaks

The memory leaks, discussed in the previous sections, were being discovered and triaged manually. This is a laborious process and more importantly requires measuring memory usage during actual execution. The earlier software defects can be discovered, the better, as the cost of of finding and fixing of software problems increases dramatically the later the problems are being found in the software release cycle. This led us to develop JSWhiz: a compiler pass to statically detect as many of these patterns as we could safely do, without flagging spurious errors.

JSWhiz's detection centers around the concept of eventful classes, which we shall describe next. A high-level overview of the approach is outlined in Algorithm 1. Using this concept we were able to track 5 of the 7 patterns described. We do not handle the patterns from Section 3.3 and Section 3.5, which require additional information to distinguish between valid and bloat inducing uses.

input : Type-annotated AST $G = (V, A, v_{\text{type}})$
output : List of potential leaks

```
// Seed eventful set
EC ← {goog.events.EventHandler};
// Initialize eventize DAG
EG ← (all types, ∅);
// Construct eventize DAG
foreach type t ∈ V do
    E_t ← {};
    if t is eventful locally then EC ← EC ∪ {t};
    foreach type r that eventize t do
        E_t ← (r, t);
    end
    // Add eventize arcs of t to EG
    A[EG] ← A[EG] ∪ E_t;
end
propagate eventful property using EG;
// Find potential leaks
foreach object o created in G do
    skip if o is filtered;
    if type[o] ∈ EC then
        if o is private member overwritten or
           o is not disposed
        then record error
    end
end
return errors reported
```

Algorithm 1: Overview of JSWhiz.

4.1 Eventful classes

The above patterns we found could be encapsulated within the concept of, what we shall call, an *eventful class*. Intuitively an eventful class is one which has events associated with it. But such a definition is both too broad and too restrictive, though it does serve the intuition. Instead we shall define a class to be eventful either due to features local to the class, or due to the composition or derivation (as supported by Closure [8]) of the class as described next.

Denote the set of classes that are eventful by EC, then a class, A, is considered eventful if

Intrinsically eventful

- A has an unmatched listen call (this includes A listening to events using an anonymous function or returned from call to `goog.bind`);
- A removes event listeners/handlers during disposal;

Eventized

- A is of type `goog.events.EventHandler` (the seed member of EC);
- A extends a class $B \in$ EC; or
- A has a class member of type $B \in$ EC which is disposed of when A is disposed of.

The eventize relationship, describe above, forms a DAG between the different types in the JavaScript application. We shall refer to this DAG as the eventize graph, denoted EG in Algorithm 1. The eventfulness property is propagated

```
/*
 Assume classP extends goog.Disposable and
 is eventized by classPF.
*/
/**
 * @constructor
 * @extends {classP}
 */
classA = function() {
  goog.base(this);

  /**
   * @type {!classFa}
   */
  this.fieldA = new classFa();

  /**
   * @type {!classFb}
   */
  this.fieldB = new classFb();
  this.registerDisposable(this.fieldB);

  /**
   * @type {!classFc}
   */
  this.fieldC = new classFc();
}
goog.inherits(classA, classP);

/** @inheritDoc */
classA.prototype.disposeInternal=function() {
  goog.base(this, 'disposeInternal');
  goog.dispose(this.fieldA);
}
```

Listing 6. Eventize relationship.

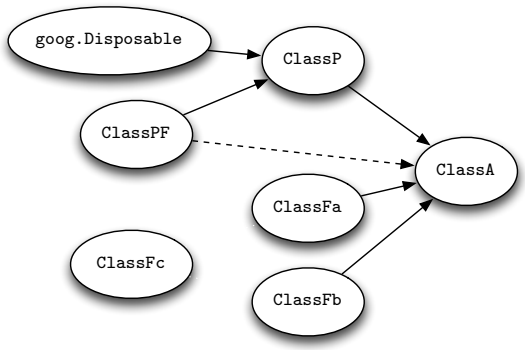

Figure 5. Eventize graph for the code example in Listing 6.

along the arcs by iterating across EG's vertices in topological order.

An example of an eventize graph is given in Figure 5 for the code snippet listed in Listing 6. Note that this snippet is annotated with Closure-style type annotations which allow the Closure compiler to make more precise statements about objects' possible types. This example shows the ways in which one class can eventize another due to composition or derivation. While `ClassPF` does, transitively, eventize `ClassA`, this is not explicitly captured in the eventize graph. Furthermore whether `ClassFc` is eventful or not does not affect `ClassA` as `fieldC` is not disposed of when an object of type `classA` is disposed.

In this example, if `classFc` was eventful, but `classA` was not intrinsically eventful and none of `classP`, `classFa` or `classFb` were, then JSWhiz would flag an error that `fieldC` was not being disposed of. Fixing this, i.e., disposing of `fieldC`, would then result in `classA` being eventful which might lead to modification of code using `classA`. Potentially fixing one memory leak identified could lead to uncovering a whole host of other leaks.

The eventize graph for Gmail is shown in Figure 6.

4.2 Find undisposed eventful objects

During the above AST traversal the set of eventful classes were determined using both intrinsic and extrinsic properties of the class. In the second pass JSWhiz identifies instances of eventful classes that are created but not disposed.

An object is considered *eventful* if it is an instance of an eventful class. JSWhiz was intended to be used frequently and it needed to execute quickly enough not to interfere with the developer's process. We therefore decided to detect undisposed eventful objects in a flow-insensitive manner by performing a single AST traversal.

Doing the analysis in a flow-insensitive manner is equal to assuming that an eventful object is disposed of if there exists a disposal instruction in the AST tree. This assumption is further necessitated by 1) the dynamic nature of JavaScript, 2) most disposals resulting from runtime events (user actions, that may or may not occur) and 3) the computational demands associated with full program/inter-procedural analysis.

4.3 Limitations of the current analysis

The set of eventful objects tracked is restricted to:

- Objects with fully qualified name. The primary target of this analysis is object properties, such as, `application.window.toolbar`, which have fully qualified names. Restricting the analysis to variables with fully qualified names excludes objects stored in arrays, lists or other data structures. Tracking such instances would require keeping track of where each element of the structure is defined and disposed. We considered multiple different approaches to statically track such cases but these attempts all resulted in false-positives without detecting additional leaks.

  Locally defined eventful objects were handled specially by tracking if they escape scope or get assigned to a globally visible property.

- Objects never returned. To track an eventful object that is returned JSWhiz would need to consider every call site and track the object across multiple functions. Tracking such instances is possible but results in the same ob-

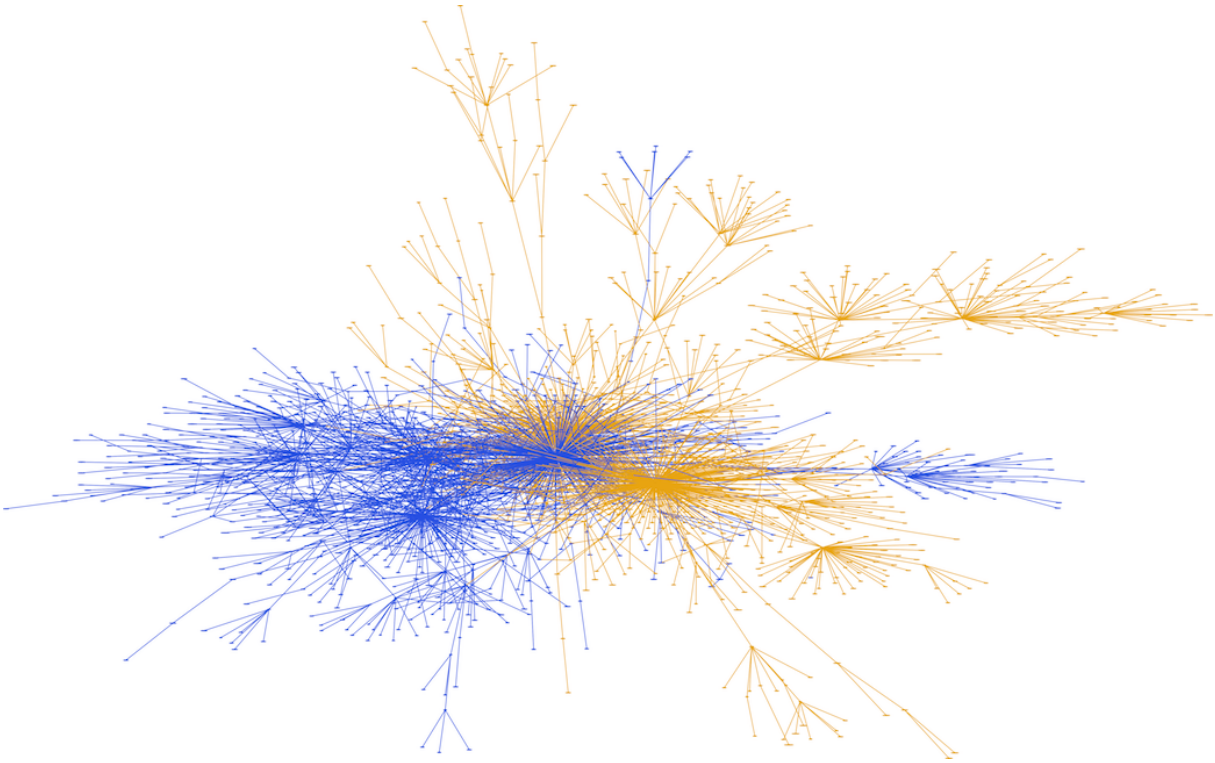

Figure 6. Eventize graph for Gmail. The graph consists of all the classes in Gmail with an arc between two classes denoting that the class at the tail eventizes the class as the head of the arc. Eventful classes, and arcs connecting eventful classes, were colored blue (primarily to the left) while the remainder was colored yellow.

ject corresponding being referred to by multiple different names.

- Objects not captured in closures. Events can be listened to using the `listenOnce` call, as well as custom created versions of `listenOnce`, where the eventful object is captured in the scope and disposed of after the event fires. This leads to the disposal of the object, but only if the listened to event ever fires.

Considering only this subset of eventful objects was not sufficient. A function could be called with an eventful object as argument, resulting in said eventful object being disposed of, or stored in an array. JSWhiz does not yet have a mechanism to automatically detect such functions—interprocedural analysis would be required to fully determine what happens to an eventful object passed into a function. But considering the basic, application independent, calls known to store eventful objects in arrays, lists, etc. (`push` , `enqueue`, `add`, *etc.*) was found to be sufficient in practice without incurring significant overhead.

Another limitation of our analysis is that we do not check for the potential of double-disposal. For example, programmers might be omitting calls to `dispose` out of fear (or potential) of introducing a double disposal. Since our analysis is neither path- nor context-sensitive, we cannot find such scenarios.

Henceforth we shall refer to the class of eventful objects defined above simply as *eventful*.

4.4 Miscellaneous checking

The majority of JSWhiz's execution is spent in checking for the disposal of eventful objects. The remaining time is checking for 1) eventful members being overwritten in the derived class and 2) unlistening with a bound/anonymous function.

Overwriting eventful members during inheritance is problematic as JavaScript (Closure) uses prototype-based inheritance [8] and therefore the base class's member is also overwritten. The base class's eventful object can then no longer be disposed of as there is no reference to it. This is actually well understood and was not among the class of leaks originally identified during code review, but JSWhiz did find 3 instances of this pattern across the 7 applications.

4.5 Asymptotic runtime

For the asymptotic runtime analysis of JSWhiz we shall only consider the sections specific to JSWhiz, excluding the runtime used by other parts of the Closure compiler (such as type inference and liveness analysis).

JSWhiz consists of two AST post-order traversals of G where the computation cost for each vertex (representing an AST node) is constant with regards to the number of vertices

Application	Leaks identified
Gmail	30
Books	14
Closure	3
Drive	10
Docs	4
Presentation	21
Spreadsheet	7

Table 1. Memory leaks identified by JSWhiz.

and arcs. The computation cost of a vertex does depend linearly on the inheritance depth [13] but that is independent and several orders of magnitude smaller than the number of vertices. Computing the set of eventful classes consists of a topological sort of EG post the first AST transversal.

But $|V[\text{EG}]| \leq |V[G]|$ and $|A[\text{EG}]| \leq |E[G]|$ hence the runtime of the AST traversal dominates JSWhiz's runtime asymptotically, therefore the runtime of JSWhiz is $O(|V[G]| + |E[G]|)$.

5. Results

JSWhiz was implemented as a command-line option of the internal version of the Closure compiler and activated within the build system of Gmail, Google Books, Drive, Docs, Presentation, Spreadsheet and Closure. JSWhiz identified 89 leaks across these applications (see Table 1 for a breakdown across application). The leaks identified by JSWhiz across these applications were manually evaluated and verified.

Figure 7 shows the measured memory footprint for all Gmail users in the time from middle of February to middle of May 2012. This graph aggregates data for the web client for all users, all browsers, and all operating systems. Since we use a new browser interface to measure memory footprint (`console.memory`), the user distribution is biased towards users of Chrome. This explains why a Chrome garbage collection regression had the visible, large negative impact on the high percentiles. The leaks identified in Gmail were fixed during an week-long internal code review ("Hackathon"). This contributed to a reduction of memory usage of over 75% in the 99th percentile, and over 50% at the median. Again, these improvements are across all browsers, which allows to correlate the improvements with the fixing of the leaks themselves.

In regards to user perceived latency, during the time span shown in Figure 7 we could observe a $\pm 10\%$ improvement of client latency at the 95th percentile, and slightly higher at the 99th percentile. These numbers are usually noisy and improvements can, for example, be caused by a profitable change in the browser mix, or machine mix. While we cannot exactly attribute the percentage of the improvements that were caused by reducing memory bloat, we are certain it did contribute and had a positive impact.

JSWhiz is currently used during pre-production testing for Gmail within Google, several other Google teams are evaluating its utility as well. Newly introduced memory leaks that can be identified by JSWhiz are now found at compile time, even before testing. From the time of writing this paper to finally enabling the analysis in production, a time span of about three months, six additional leaks were introduced by developers and flagged by the tool.

The asymptotic runtime of JSWhiz was given in Section 4.5. The implementation was found to incur between 3% and 14% compilation runtime overhead over plain JavaScript compilation for these applications, some of which consists of many thousands of classes and more than a million lines of JavaScript code. The tests are only enabled in a handful of "Compile All" test cases and are therefore no burden to developers in their everyday work.

6. Related Work

Memory leaks and memory bloat have been researched extensively in the past. Besides publications, there are many open-source tools and commercial solutions. Much of this work targets C++, or Java and other garbage collected languages.

For C++ the prominent Purify [16] and Valgrind [22] use conservative garbage collection to find leaks. Novark et al. [23] presents *Hound*, which uses data sampling and a staleness tracking approach to find leaks and bloats precisely and efficiently for C++. SWAT [17] also uses a sampling based approach.

FastCheck [11] performs inter-procedural analysis on C programs to track allocation and deallocation points using a sparse representation consisting of a value flow graph containing guards to model program flow. Memory leak analysis thus becomes a reachability problem. They were able to detect 60 memory leaks in SPEC2000 alone, while keeping the false positive rate below 20%. Sparrow [19] uses a summary based, fix-point iteration approach to find leaks via abstract interpretation. They are as fast as FastCheck, yet claim to find more leaks, with a lower false positive rate (less than 13%). Finally, Saber [26] uses a *full-sparse* value-flow analysis for leak detection. They exploit field-, flow-, and context-sensitivity and also solve a graph reachability problem. On the benchmarks used by FastCheck and Sparrow, they are as accurate as Saber, but 14.2x faster. They also report 41% more leaks as FastCheck, with a slightly higher false positive rate and runtime overhead of 3.7x.

Saturn [5] is a sophisticated tool for static analysis of programs. It is summary based, analyzing each function in isolation and only utilizing summary information at call sites. Analyzes are formulated as constraints and a logic programming language is used to construct constraints and to access results. We believe that a system like this can perform analysis similar and even more powerful than the ones presented in this paper. Impressive results have been

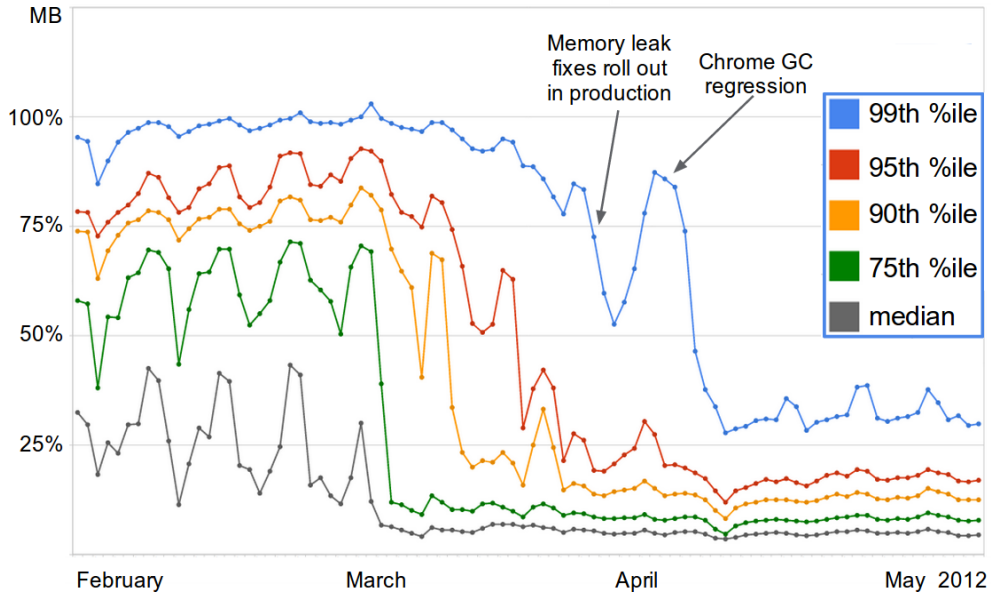

Figure 7. Improvements in Gmail memory footprint.

reported for a parallelized version running on the Linux kernel [28].

These tools cannot, however, find leaking objects that are reachable, or look beyond C++, e.g., into the JavaScript VM.

For Java and other garbage collected languages, LeakBot [20] identifies Java data structures that may be potential leaks. Cork [18] uses points-to analysis and graph to infer sources of leaks. Bond and McKinley [9] introduces Bit-Encoding Leak Location, a statistical approach that encodes per-object sites to a single bit per object, utilizing large sample sizes and a brute-force decoding approach to recover leaking sites with high accuracy. This smart, compact encoding is part of Sleigh, an instrumentation based approach to finding leaks, which improves over similar previous work on SWAT for C/C++ [17]. We believe these tools are also limited to their language environments and cannot bridge, e.g., between a JavaScript VM and the browser's DOM bindings and event system to identify leaks.

FindBugs [6] is a static Java analysis tool which finds hundreds of coding problems mostly via simple pattern matching techniques. It also includes several sophisticated inter-procedural analyses. However, FindBugs doesn't appear to specifically target finding of Java memory leaks.

We are not aware of any systematic research to identify memory leak patterns in JavaScript statically. We believe that because of the complex browser embedding of JavaScript the types of memory leaks we were observing are not possible to catch with the approaches targeting either C++ or garbage collected languages. Rather, a general solution needs to take both environments into account. Static analysis, as presented in this paper, is a first step, but can only target know patterns and problems. This is subject to interesting future research.

The JavaScript developer community is aware of the potential for leaks in JavaScript. There are many web pages and blog posts describing specific problems, for example, [2–4, 7] or [27]. Several of the documented patterns describe issues in browsers itself. For example, the issue with Internet Explorer 6 described in [25] was actually the motivation for the current design of Closure's event system.

7. Conclusion

In this paper we have presented patterns that lead to memory leaks in JavaScript applications, approaches to finding and fixing them, and recommended practices for avoiding them in the first place. We introduced JSWhiz, a tool to statically analyze JavaScript applications and identify potential memory leaks. JSWhiz was tested across 7 major JavaScript applications and found 89 bugs in production code. Fixing of the bugs identified by JSWhiz contributed to more than 75% reduction in 99th percentile memory usage of Gmail, and more than 50% at the median. The latency impact at the higher percentiles is impossible to break out, but removing leaks certainly contributed to an observed 10% improvement at the 95th percentile. JSWhiz is currently being used in pre-release testing for Gmail and being evaluated by many other Google application teams.

Our discussion is mostly specific to Closure. However, the problem is not and the observed patterns are general in nature. Similar problems exist in many other JavaScript development frameworks. We hope our work raises awareness in the community and hope that open-sourcing our implementation will spawn interesting further research.

Acknowledgments

We thank Loreena Lee for providing the memory graphs and many helpful discussions. We like to thank John Lenz for his help with the Closure compiler and the many Google engineers helping us with the various application build and delivery frameworks. We thank the anonymous reviewers, specifically our shepherd Ben Zorn. All comments helped tremendously to improve the quality of this paper.

References

[1] node.js. URL `http://nodejs.org/`.

[2] Avoid memory leaks in Dojo, 2011. URL `http://stackoverflow.com/questions/7446188`.

[3] jQuery memory leak patterns and causes, 2011. URL `http://stackoverflow.com/questions/5046016`.

[4] How to do proper memory management using YUI to avoid leaks, 2011. URL `http://stackoverflow.com/questions/8529447`.

[5] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn project. PASTE '07, pages 43–48, New York, NY, USA, 2007. ACM.

[6] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Softw.*, 25 (5):22–29, Sept. 2008.

[7] A. Bhattacharya and K. S. Sund. Memory leak patterns in JavaScript. `http://www.ibm.com/developerworks/web/library/wa-memleak/`, 2007.

[8] M. Bolin. *Closure: The Definitive Guide*. Oreilly Series. O'Reilly Media, Incorporated, 2010.

[9] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. ASPLOS-XII, pages 61–72, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0.

[10] J. Chaffer and K. Swedberg. *Learning jQuery, Third Edition*. Packt Publishing, Limited, 2011.

[11] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. *SIGPLAN Not.*, 42(6):480–491, June 2007.

[12] R. Cooper and C. Collins. *GWT in Practice*. Manning Publications Co., Greenwich, CT, USA, 2008.

[13] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. An empirical study evaluating depth of inheritance on the maintainability of object-oriented software. *Empirical Software Engineering, An international Journal*, 1:109–132, 1996.

[14] ECMA. *ECMA-262: ECMAScript Language Specification*. 1999.

[15] R. Hanson and A. Tacy. *GWT in Action: Easy Ajax with the Google Web Toolkit*. Manning Publications Co., Greenwich, CT, USA, 2007.

[16] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[17] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. *SIGOPS Oper. Syst. Rev.*, 38(5):156–164, Oct. 2004.

[18] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. POPL '07, pages 31–38, New York, NY, USA, 2007. ACM.

[19] Y. Jung and K. Yi. Practical memory leak detector based on parameterized procedural summaries. ISMM '08, pages 131–140, New York, NY, USA, 2008. ACM.

[20] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *ECOOP 2003*, volume 2743 of *Lecture Notes in Computer Science*, pages 351–377. Springer, 2003.

[21] Mozilla. Rhino javascript compiler, 2012. URL `https://developer.mozilla.org/en-US/docs/Rhino`.

[22] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[23] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 397–407, New York, NY, USA, 2009. ACM.

[24] M. Russell. *Dojo: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, Incorporated, 2008.

[25] I. Schlueter. Memory leaks in Microsoft Internet Explorer, 2007. URL `http://foohack.com/2007/06/msie-memory-leaks/`.

[26] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. ISSTA 2012, pages 254–264, New York, NY, USA, 2012. ACM.

[27] J. K. Volcan Sarmal. JavaScript and memory leaks, 2012. URL `http://www.javascriptkit.com/javatutors/closuresleak/index.shtml/`.

[28] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. *SIGSOFT Softw. Eng. Notes*, 30(5):115–125, Sept. 2005.