GULFSTREAM: Staged Static Analysis for Streaming JavaScript Applications

Salvatore Guarnieri
University of Washington

Benjamin Livshits
Microsoft Research

Abstract

The advent of Web 2.0 has led to the proliferation of client-side code that is typically written in JavaScript. Recently, there has been an upsurge of interest in static analysis of client-side JavaScript for applications such as bug finding and optimization. However, most approaches in static analysis literature assume that the *entire program* is available to analysis. This, however, is in direct contradiction with the nature of Web 2.0 programs that are essentially being streamed at the user's browser. Users can see data being streamed to pages in the form of page updates, but the same thing can be done with code, essentially delaying the downloading of code until it is needed. In essence, the entire program is never completely available. Interacting with the application causes more code to be sent to the browser.

This paper explores *staged static analysis* as a way to analyze streaming JavaScript programs. We observe while there is variance in terms of the code that gets sent to the client, much of the code of a typical JavaScript application can be determined statically. As a result, we advocate the use of combined offline-online static analysis as a way to accomplish fast, browser-based client-side online analysis at the expense of a more thorough and costly server-based offline analysis on the static code. We find that in normal use, where updates to the code are small, we can update static analysis results quickly enough in the browser to be acceptable for everyday use. We demonstrate the staged analysis approach to be advantageous especially in mobile devices, by experimenting on popular applications such as Facebook.

1 Introduction

The advent of Web 2.0 has led to the proliferation of client-side code that is typically written in JavaScript. This code is often combined or *mashed-up* with other code and content from different third-party servers, making the application only fully available within the user's browser. Recently, there has been an upsurge of interest in static analysis of client-side JavaScript. However, most approaches in the static analysis literature assume that the entire program is available for analysis. This, however, is in direct contradiction with the nature of Web 2.0 programs that are essentially being *streamed* to the user's browser. In essence, the JavaScript application is never available in its entirety: as the user interacts with the application, more code is sent to the browser.

A pattern that emerged in our experiments with static analysis to enforce security properties [14], is that while most of the application can be analyzed offline, some parts of it will need to be analyzed on-demand, in the browser. In one of our experiments, while 157 KB (71%) of Facebook JavaScript code is downloaded right away, an additional 62 KB of code is downloaded when visiting event pages, etc. Similarly, Bing Maps downloads most of the code right away; however, requesting traffic requires additional code downloads. Moreover, often the parts of the application that are downloaded later are composed on the client by referencing a third-party library at a fixed CDN URL; common libraries are jQuery and `prototype.js`. Since these libraries change relatively frequently, analyzing this code ahead of time may be inefficient or even impossible.

The dynamic nature of JavaScript, combined with the incremental nature of code downloading in the browser leads to some unique challenges. For instance, consider the piece of HTML in Figure 1. Suppose we want to statically determine what code may be called from the `onclick` handler to ensure that none of the invoked functions may block. If we only consider the first `SCRIPT` block, we will conclude that the `onclick` handler may only call function `foo`. Including the second `SCRIPT` block adds function `bar` as a possible function that may be called. Furthermore, if the browser proceeds to download more code, either through more `SCRIPT` blocks or `XmlHttpRequests`, more code might need to be consid-

```
<HTML>
    <HEAD>
        <SCRIPT>
            function foo(){...}
            var f = foo;
        </SCRIPT>

        <SCRIPT>
            function bar(){...}
            if (...) f = bar;
        </SCRIPT>
    </HEAD>
    <BODY onclick="f();">
        ...
    </BODY>
</HTML>
```

Figure 1: Example of adding JavaScript code over time.

ered to find all possible targets of the `onclick` handler.

While it is somewhat of an artificial example, the code in Figure 1 demonstrates that JavaScript in the browser essentially has a *streaming programming* model: sites insert JavaScript into the HTML sent to the user, and the browser is happy to execute any code that comes its way.

GULFSTREAM advocates performing *staged static analysis* within a Web browser. We explore the trade-off between offline static analysis performed on the server and fast, staged analysis performed in the browser. We conclude that staged analysis is fast enough, especially on small incremental updates, to be made part of the overall browser infrastructure. While our focus is on analyzing large, modern AJAX applications that use JavaScript, we believe that a similar approach can be applied to other platforms such as Silverlight and Flash.

1.1 Contributions

This paper makes the following contributions:

- **Staged analysis.** With GULFSTREAM, we demonstrate how to build a staged version of a points-to analysis, which is a building block for implementing static analysis for a wide range of applications, including security and reliability checkers as well as optimizations. Our analysis is staged: the server first performs *offline* analysis on the statically available code, serializes the results, and sends them to a client which performs analysis on code *deltas* and updates the results from the offline analysis. To our knowledge, GULFSTREAM is the first static analysis to be staged across across multiple machines.
- **Trade-offs.** We use a wide range of JavaScript inputs of various sizes to estimate the overhead of staged computation. We propose strategies for choosing between staging analysis and full analysis for various network settings. We explore the trade-off between computation and network data transfer and suggest strategies for different use scenarios.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 provides background on both client-side Web applications and static analysis. Section 3 provides an overview of our approach. Section 4 gives a description of our implementation. Section 5 discusses our experimental results. Finally, Sections 6 and 7 describe related work and outline our conclusions.

2 Background

This section first provides a background on static analysis and its most common applications, and then talks about code loading in Web applications.

2.1 Static Analysis

Static analysis has long been recognized as an important building block for achieving reliability, security, and performance. Static analysis may be used to find violations of important reliability properties; in the context of JavaScript, tools such as JSLint [9] fulfill such a role. Soundness in the context of static analysis gives us a chance to provide guarantees on the analysis results, which is especially important in the context of checking security properties. In other words, lack of warnings of a static analyzer implies that no security violations are possible at runtime; several projects have explored this avenue of research for client-side JavaScript [8, 14]. Finally, static analysis may be used for optimization: statically-computed information can be used to optimize runtime execution. For instance, in the context of JavaScript, static knowledge of runtime types [18] may be used to improve the performance of runtime interpretation or tracing [13] within the JavaScript runtime.

Several broad approaches exist in the space of static analysis. While some recent static analysis in type inference have been made for JavaScript [16], the focus of this paper is on *pointer analysis*, long recognized as a key building block for a variety of static analysis tasks. Because function closures can be easily passed around in JavaScript, pointer analysis is even necessary for something as ostensibly simple as call graph construction.

The goal of pointer analysis is to answer the question "given a variable, what heap objects may it point to?" While a great variety of techniques exist in the pointer analysis space, resulting in widely divergent trade-offs between scalability and precision, a popular choice is to represent heap objects by their allocation site. For instance, for the following program

```
1. var v = null;
2. for (...) {
3.     var o1 = new Object();
4.     var o2 = new Object();
5.     if (...)
6.         v = o1;
7.     else
8.         v = o2;
9. }
```

variables `o1` and `o2` point to objects allocated on lines 3 and 4, respectively. Variable `v` may point to either object, depending on the outcome of the `if` on line 5. Note that all objects allocated on line 3 within the loop are represented by the same allocation site, potentially leading to imprecision. However, imprecision is inevitable in static analysis, as it needs to represent a potentially unbounded number of runtime objects with a constant number of static representations.

In this paper, we focus on the points-to analysis formulation proposed by the Gatekeeper project [14]. Gatekeeper implements a form of inclusion-based Andersen-style context-insensitive pointer analysis [2], which shows good scalability properties, potentially with a loss of precision due to context insensitivity. However, for many applications, such as computing the call graph for the program, context sensitivity has not been shown to be necessary [21].

Static analysis is generally used to answer questions about what the program might do at runtime. For instance, a typical query may ask if it is possible for the program to call function `alert`, which might be desirable to avoid code leading to annoying popup windows. Similarly, points-to information can be used to check heap isolation properties such as "there is no way to access the containing page without going through proper APIs" in the context of Facebook's FBJS [12]. Properties such as these can be formulated as statically resolved heap reachability queries.

2.2 Code Loading in Web Applications

As we described above, Web 2.0 programs are inherently streaming, which is to say that they are downloaded over time. Below we describe a small study we performed of two large-scale representative AJAX applications. Figure 2 summarizes the results of our experiments. We start by visiting the main page of each application and then attempt to use more application features, paying attention to how much extra JavaScript code is downloaded to the user's browser. Code download is cumulative: we take care not to change the browser location URL, which would invalidate the current JavaScript context.

As Figure 2 demonstrates, much of the code is downloaded initially. However, as the application is used, quite a bit of extra code, spanning multiple potentially independently changing files, is sent to the browser. In the case of Facebook, the JavaScript code size grows by about 30% (9 files) once we have used the application for a while. OWA, in contrast, is a little more monolithic, growing by about 23% (5 files) over the time of our use session. Moreover, the code that is downloaded on demand is highly workload-driven. Only some users will need certain features, leading much of the code to be used quite rarely. As such, analyzing the "initial" portion of the application on the server and analyzing the rest of the code on-the-fly is a good fit in this highly dynamic environment.

Page visited or action performed	Added JavaScript files	KB
FACEBOOK FRONT PAGE		
Home page	19	157
Friends	7	29
Inbox	1	20
Profile	1	13
FACEBOOK SETTINGS PAGE		
Settings: Network	13	136
Settings: Notifications	1	1
Settings: Mobile	3	14
Settings: Language	1	1
Settings: Payments	0	0
OUTLOOK WEB ACCESS (OWA)		
Inbox page	7	1,680
Expand an email thread	1	95
Respond to email	2	134
New meeting request	2	168

Figure 2: Incremental loading of Facebook and OWA JavaScript code.

3 Overview

In this paper, we consider two implementation strategies for points-to analysis. The first one is based on in-memory graph data structures that may optionally be serialized to be transmitted from the server to the client. The second one is Gatekeeper, a BDD-based implementation described by Guarnieri and Livshits in [14]. Somewhat surprisingly, for small code bases, we conclude that there is relatively little difference between the two implementations, both in terms of running time as well as in terms of the size of result representation they produce. In some cases, for small incremental updates, a graph-based representation is more efficient than the bddbddb-based one. The declarative approach is more scalable, however, as shown by our analysis of Facebook in Section 5.4. Figure 3 summarizes the GULFSTREAM approach and shows how it compares to the Gatekeeper strategy.

Staged analysis. As the user interacts with the Web site, updates to the JavaScript are sent to the user's browser

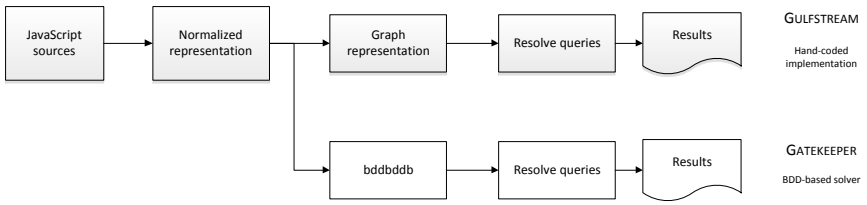


Figure 3: GULFSTREAM architecture and a comparison with the Gatekeeper project.

that in turn update the Web site. If the updates to the Web site's JavaScript are small, it would make sense that an staged analysis would perform better than a full program analysis. We looked at range of update sizes to identify when an staged analysis is faster than recomputing the full program analysis. Full program analysis might be faster because there is book keeping and graph transfer time in the staged analysis that is not present in the full program analysis. Section 5 talks about advantages of staged analysis in detail. In general, we find it to be advantageous in most settings, especially on slower mobile connections with slower mobile hardware.

Soundness. In this paper we do not explicitly focus on the issue of analysis soundness. Soundness would be especially important for a tool designed to look for security vulnerabilities, for instance, or applications of static analysis to runtime optimizations. Generally, sound static analysis of JavaScript only has been shown possible for *subsets* of the language. If the program under analysis belongs to a particular language subset, such as JavaScript$_{\text{SAFE}}$ advocated by Guarnieri et al. [14], the analysis results are sound. However, even if it does *not*, analysis results can still be used for bug finding, without necessarily guaranteeing that all the bugs will be found. In the remainder of the paper, we ignore the issues of soundness and subsetting, as we consider them to be orthogonal to staged analysis challenges.

Client analyses as queries. In addition to the pointer analysis, we also show how GULFSTREAM can be used to resolve two typical queries that take advantage of points-to analysis results. The first query looks for calls to `alert`, which might be an undesirable annoyance to the user and, as such, need to be prevented in third-party code. The second looks for calls to `setInterval`[1] with non-function parameters.

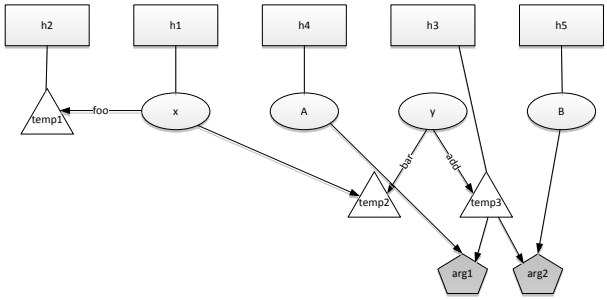
4 Techniques

The static analysis process in GULFSTREAM proceeds in stages, as is typical for a declarative style of program analysis. On a high level, the program is first represented as a database of facts. Next, a *solver* is used to derive new information about the program on the basis of initial facts and *inference rules*.

[1]Function `setInterval` is effectively a commonly overlooked form of dynamic code loading similar to `eval`.

```
1.  var A = new Object();
2.  var B = new Object();
3.  x     = new Object();
4.  x.foo = new Object();
5.  y     = new Object();
6.  y.bar = x;
7.  y.add = function(a, b) {}
8.  y.add(A, B)
```

(a) Input JavaScript program.



(b) Resulting graph.

Figure 4: Example of a program with a function call.

In GULFSTREAM, the first analysis stage is normalizing the program representation. Based on this normalized representation, we built two analyses. The first is the declarative, bddbddb-based points-to analysis described in Gatekeeper [14]. The second is a hand-coded implementation of points-to information using graphs as described in the rest of this section.

The graph-based representation also produces graphs that can efficiently compressed and transferred to the browser from the server. To our surprise, we find that at least for small programs, the graph-based representation performs at least as well as the bddbddb-based approach often advocated in the past; bddbddb-based analysis, however, performs faster on larger code bases, as discussed in Section 5.4.

Node type	Description	Node shape
Variable	The basic node is a simple variable node. It represents variables from the program or manufactured during normalization. Line 1 in Figure 4 has two variable nodes, `A` and `Object`.	Oval
Heap	These nodes represent memory locations and are sinks in the graph: they do not have any outgoing edges. Heap nodes are created when new memory is created like in line 1 in Figure 4 when a new `Object` is created.	Rectangle
Field	These nodes represent fields of objects. They are similar to variable nodes, except they know their object parent and they know the field name used to access them from their object parent. Conversely, variables that have fields contain a list of the field nodes for which they are the parent. Field nodes are represented by a triangular node connected to the object parent by a named edge. Line 4 shows the use of a field access. The name of the edge is the name of the field.	Triangle
Argument	The fourth type of node is a special node called an argument node. These nodes are created for functions and are used to link formals and actuals. The argument nodes contain edges to their respective argument variables in the function body and when a function is called, the parameter being passed in gets an edge to the respective argument node. In the graph, Argument nodes are represented by pentagons. Lines 7 and 8 from Figure 4 show a function node being created and used. Return values are also represented by this type of node.	Pentagon

Figure 5: Description of nodes types in the graph.

4.1 Normalization

The first analysis stage is normalizing the program representation and is borrowed from the Gatekeeper [14] project. The original program statements are broken down to their respective normalized versions, with temporaries introduced as necessary. Here is a normalization example that demonstrates variable introduction:

```
var x = new Date();     x = new Date();
var y = 17;             y = ⊥;
h.f   = h.g;            t = h.g; h.f = t;
```

Variable `t` has been introduced to hold the value of field `h.g`. Since we are not concerned with primitive values such as `17`, we see it represented as $\perp$.

4.2 Graph Representation

The points-to information is calculated from a graph representing the program stored in memory. The graph is generated from the normalized program. Assignments turn into edges, field accesses turn into named edges, constructor calls create new sinks that represent the heap, and so on. The graph fully captures the points-to information for the program. One important note is that this graph is not transitively closed. If the program states that `A` flows to `B` and `B` flows to `C`, the graph does not contain an edge from `A` to `C` even though `A` flows to `C`. The graph must be traversed to conclude that `A` points to `C`.

The full graph consists of several different types of nodes, as summarized in Figure 5. We use the program and corresponding graph in Figure 4 as an example for our program representation. In lines 1-5, the program is creating new objects which creates new heap nodes in the graph. In lines 4, 6, and 7, the program is accessing a field of an object which makes use of a field edge to connect the base object's node to the field's field node. The first use of a field creates this new edge and field node. Line 7 creates a new function, which is similar to creating a new object. It creates a new heap node, but the function automatically contains argument nodes for each of its arguments. These nodes act as a connection between actuals and formals. All actuals must flow through these argument nodes to reach the formal nodes inside the function body. Line 8 calls the function created in line 7. This line creates assignment edges from the actuals (`A` and `B`) to the argument nodes, which already have flow edges to the formal.

4.3 Serialized Graph Representation

The output of each stage of analysis is also the input to the next stage of analysis, so the size and transfer time of this data must be examined when looking at our staged analysis. We compare the sizes of two simple file formats that we implemented and a third that is the bddbddb graph output, which is a a serialized BDD.

The first format from our analysis is based on the graphviz DOT file format [11]. This format maintains variable names for each node as well as annotated edges. The second format from our analysis is efficient for directed graphs and removes all non-graph related data like names. This format is output in binary is as follows:

```
[nodeid];[field_id1],[field_id2],...;[arg_id1],...;
[forward_edge_node_id1],[forward_edge_node_id2],...;
[backward_edge_node_id1],[backward_edge_node_id2],...;
[nodeid]...
```

where `nodeid`, `field_id1`, etc. are uniquely chosen integer identifiers given to nodes within the graph. Finally, the third format is a serialized BDD-based representation of bddbddb.

Overall, the sizes of the different formats of the staged-results graph vary widely. The DOT format is the largest, and this is to be expected since it is a simple text file describing how to draw the graph. The binary format and bddbddb output are closer in size, with the binary format being marginally smaller. A more detailed comparison of graph representation sizes is presented in Section 5.

```
pointsTo = ∅ ↦ ∅ // points-to map
reversePointsTo = ∅ ↦ ∅ // reverse version of points-to map
inc_insert(G, e) // incrementally update points-to map

 1: invalid = ∅
 2: if e.src ∈ G then
 3:     invalidate(e.src, invalid)
 4: end if
 5: if e.dst ∈ G then
 6:     invalidate(e.dst, invalid)
 7: end if
 8: G = ⟨G_N ∪ {e_src, e_dst}, G_E ∪ {e}⟩
 9: for all n ∈ invalid do
10:     ans =compute-points-to(n, ∅)
11:     pointsTo[n] = pointsTo[n] ∪ ans
12:     for all h ∈ ans do
13:         reversePointsTo[h] = reversePointsTo[h] ∪ n
14:     end for
15: end for

invalidate(n ∈ G_N, invalid) // recursively invalidate following flow edges

 1: if n ∈ invalid then
 2:     return
 3: end if
 4: invalid = invalid ∪{n}
 5: if n is FieldNode then
 6:     toVisit = compute-field-aliases(n.parent, n.fieldname)
 7: end if
 8: for all n' adjacent to n do
 9:     if n → n' is an assignment edge then
10:         toVisit = toVisit ∪ n'
11:     end if
12: end for
13: for all n' ∈ toVisit do
14:     invalidate(n')
15: end for
```

Figure 6: Routines inc_insert and invalidate.

Since our main focus was not to develop a new efficient graph storage format, we `gzip` all the graph output formats to see how their sizes compared under an industry-standard compression scheme. Since BDDs are highly optimized to minimize space usage, one would expect their zipped size to be similar to their unzipped size. As expected, the DOT format receives huge gains from being zipped, but it is still the largest file format. The difference between the three formats is minimal once they are all zipped. Since this data must be transferred from the server to the client to perform the staged analysis, these figures indicate that the graph output format does not make much of a difference on the staged analysis time on a fast link assuming `gzip` times do not vary much from one format to another. We leave more detailed measurements that take decompression time into account for future work.

4.4 Points-to Analysis Implementation

Our system normalizes JavaScript into a representation that we can easily output for analysis. This means it is straightforward for us to try several different analysis techniques. We have two outputs of our representation at the moment, an output to Datalog facts that is used by bddbddb and an output to a graph representing the program which is used by our implementation of a points-to analysis. The reader is referred to prior work for more information about bddbddb-based analyses [6, 14, 25].

```
compute-points-to(n, visitedNodes)
 1: if n ∈ visitedNodes then
 2:     return ∅
 3: else
 4:     visitedNodes = visitedNodes ∪ {n}
 5: end if
 6: toVisit = ∅
 7: ans = ∅
 8: if n is HeapNode then
 9:     return n
10: end if
11: if n is FieldNode then
12:     toVisit = toVisit ∪
            compute-field-aliases(n.parent, n.fieldname)
13: end if
14: for assignment-edge e leaving n do
15:     toVisit = toVisit ∪ {e.sink}
16: end for
17: for node n' ∈ toVisit do
18:     ans = ans ∪ compute-points-to(n', visitedNodes)
19: end for
20: return ans

compute-field-aliases(parent, fieldname)
 1: toVisit = ∅
 2: if parent is FieldNode then
 3:     toVisit = toVisit ∪
            compute-field-aliases(parent.parent, parent.fieldname)
 4: end if
 5: toVisit = toVisit∪ compute-aliases(parent)
 6: for n ∈ toVisit do
 7:     if n has field fieldname then
 8:         ans = ans ∪ {n.fieldname}
 9:     end if
10: end for
11: return ans

compute-aliases(n, visitedNodes)
 1: ans = n
 2: if n ∈ visitedNodes then
 3:     return ∅
 4: else
 5:     visitedNodes = visitedNodes ∪ {n}
 6: end if
 7: for edge e leaving n do
 8:     ans = ans ∪ compute-aliases(e.sink, visitedNodes)
 9: end for
10: return ans
```

Figure 7: Points-to computation algorithm.

GULFSTREAM maintains a graph representation that is updated as more of the program is processed. Figure 6 shows a pseudo-code version of the graph update algorithm that we use. In addition to maintaining a graph G, we also save two maps: $pointsTo$, mapping variables to heap locations and its reverse version for fast lookup, $reversePointsTo$. Function inc_insert processes every edge e inserted into the graph. If the edge is not adjacent to any of the existing edges, we update G with edge e. If it is, we add the set of nodes that are adjacent to the edge, together with a list of all nodes from which they flow to a worklist called $invalid$. Next, for all nodes in that worklist, we proceed to recompute their points-to values.

The points-to values are recomputed using a flow based algorithm. Figure 7 shows the pseudo-code ver-

sion of our points-to algorithm, including helper functions. For standard nodes and edges, it works by recursively following all reverse flow edges leaving a node until it reaches a heap node. If a cycle is detected, that recursion fork is killed as all nodes in that cycle will point to the same thing and that is being discovered by the other recursion forks. Since flows are created to argument nodes when functions are called, this flow analysis will pass through function boundaries.

Field nodes and argument nodes require special attention. Since these nodes can be indirectly aliased by accessing them through their parent object, they might not have direct flows to all their aliases. When a field node is reached in our algorithm, all the aliases of this field node are discovered, and all edges leaving them are added to our flow exploration. This is done by recording the name of the field for the current field node, finding all aliases of the parent to the field node, and getting their copy of a field node representing the field we are interested in. In essence, we are popping up one level in our flow graph, finding all aliases of this node, and descending these aliases to reach an alias of our field node. This process may be repeated recursively if the parent of a field node is itself a field node. The exact same procedure is done for argument nodes for the case when function aliases are made.

Note that the full analysis is a special, albeit more inefficient, case of the staged analysis where the $invalid$ worklist is set to be all nodes in the graph G_N. Figure 7 shows pseudo-code for computing points-to values for a particular graph node n.

4.5 Queries

The points-to information is essentially a mapping from variable to heap locations. Users can take advantage of this mapping to run queries against the program being loaded. In this paper, we explore two representative queries and show how they can be expressed and resolving using points-to results.

Not calling `alert`. It might be undesirable to bring up popup boxes, especially in library code designed to be integrated into large Web sites. This is typically accomplished with function `alert` in JavaScript. This query checks for the presence of `alert` calls.

Not calling `setInterval` **with a dynamic function parameter**. In JavaScript, `setInterval` is one of the dynamic code execution constructs that may be used to invoke arbitrary JavaScript code. This "cousin of `eval`" may be used as follows:

```
setInterval(
  new Function(
        "document.location='http://evil.com';"),
      500);
```

In this case, the first parameter is dynamically constructed function that will be passed to the JavaScript interpreter for execution. Alternatively, it may be a reference to a function statically defined in the code. In order to prevent arbitrary code injection and simplify analysis, it is desirable to limit the first parameter of `setInterval` to be a statically defined function, not a dynamically constructed function.

Figure 8 shows our formulation of the queries. The `detect-alert-calls` query looks for any calls to `alert`. It does this by first finding all the nodes that point to `alert`, then examining them to see if they are called (which is determined during normalization). The `detect-set-interval-calls` is somewhat more complicated. It cares if `setInterval` is called, but only if the first parameter comes from the return value of the `Function` constructor. So, all the source nodes from edges entering the first argument's node in `setInterval` must be examined to see if it has an edge to the return node of the `Function` constructor. In addition, all aliases of these nodes must also be examined to see if they have a flow edge to the return node of the `Function` constructor.

The results to these queries are updated when updates are made to the points-to information. This ensures that the results are kept current on the client machine. A policy is a set of queries and expected results to those queries. A simple policy would be to disallow any calls to `alert`, so it would expect `detect-alert-calls` from Figure 8 to return `false`. If `detect-alert-calls` ever returns true, the analysis engine could either notify the user or stop the offending page from executing.

4.6 Other Uses of Static Analysis

In addition to the two queries detailed above, there are many other uses that could be very useful. One other useful query would be to use the points-to results to identify when updates to pages modify important global variables. The points-to results can be traversed to identify when any aliases to global variables, like `Array` or common global library names, are modified which could lead to unexpected behavior.

Another use of the staged static analysis is helping to improve performance through optimization. Traditionally Just-In-Time (JIT) compilers have been used to improve the performance of dynamic languages like JavaScript [5]. These JITs have the benefit of actually seeing code paths during execution and optimizing them, but they must run on the client and thus have some amount of performance impact. Any complex analysis done by a JIT would negatively affect performance, which is what the JIT is trying to improve in the first place. Performing some amount of static analysis be-

```
detect-alert-calls()
 1: nodes = reversePointsTo[alert_Γ]
 2: for all n ∈ nodes do
 3:     if n.isCalled() then
 4:         return true
 5:     end if
 6: end for
 7: return false
```

```
detect-set-interval-calls()
 1: n = setInterval_Γ.arg1
 2: for all edge e entering n do
 3:     if e.src == Function_Γ.return then
 4:         return true
 5:     else
 6:         p = p ∪ find-all-aliases(e.src)
 7:     end if
 8: end for
 9: for all node n2 in p do
10:     for all edge e2 entering n2 do
11:         if e2.src == Function_Γ.return then
12:             return true
13:         end if
14:     end for
15: end for
```

```
find-all-aliases(node)
1: aliases = empty
2: heapNodes = pointsTo[node]
3: for all n ∈ heapNodes do
4:     aliases = aliases ∪ reversePointsTo[n]
5: end for
6: return aliases
```

Figure 8: Queries detect-alert-calls and detect-set-interval-calls.

fore running JavaScript through a JIT could empower the JIT to better optimize code [18]. GULFSTREAM would permit this without having to do an expensive analysis while the JIT is running. GULFSTREAM is especially well suited to this situation because the majority of the staged analysis is done offline and only updates to the code are analyzed on the client. Static analysis enables many analyses that can be semantically driven rather than syntactically driven and possibly fragile.

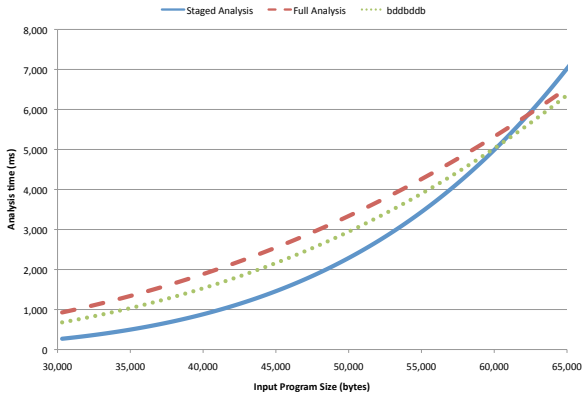


Figure 9: Trend lines for running times for full and staged analyses as well as the bddbddb-based implementation

5 Experimental Evaluation

This section is organized as follows. We first discuss analysis time and the space required to represent analysis results in Sections 5.1 and 5.2. Section 5.3 explores the tradeoff between computing results on the client and transferring them over the wire.

Measurements reported in this paper were performed on a MacBook Pro 2.4 GHz Dual Core machine running Windows 7. The JavaScript files we used during testing were a mix of hand crafted test files, procedurally generated files, and files obtained from Google code search, looking for JavaScript files. GULFSTREAM uses two bootstrap JavaScript files. The first introduces the native environment, where `Object`, `Array`, and other globals are defined. The second introduces a set of browser-provided globals such as `document` and `window`. Together these files are approximately 30 KB in size.

5.1 Analysis Running Time

Figure 9 shows full, staged and the bddbddb-based analyses on the same scale . For this experiment, we used our bootstrap file for the base in the staged analysis. We ran various sized JavaScript files through the full, staged, and bddbddb-based analyses. The full and bddbddb-based analyses processed the JavaScript file concatenated with the bootstrap file. The staged analysis processed the JavaScript file as an update to the already computed analysis on the bootstrap file.

We see that staged analysis is consistently faster than full analysis. In the cases of smaller code updates, the difference in running times can be as significant as a couple of orders of magnitude. We also see that for small updates, the staged analysis performs better than the bddbddb-based analysis. This is encouraging: it means that we can implement the staged analysis within the browser without the need for heavyweight BDD machinery, without sacrificing performance in the process. In the next section, we show that our space overhead is also generally less than that of BDDs.

5.2 Space Considerations

Figure 10 shows the sizes of three representations for points-to analysis results and how they compare to each other. The representations are DOT, the text-based graph format used by the Graphviz family of tools, bddbddb, a compact, BDD-based representation, as well as BIN, our graph representation described in Section 4.3. All numbers presented in the figure are after applying the industry-standard `gzip` compression.

We were not surprised to discover that the DOT version is most verbose. To our surprise, our simple bi-

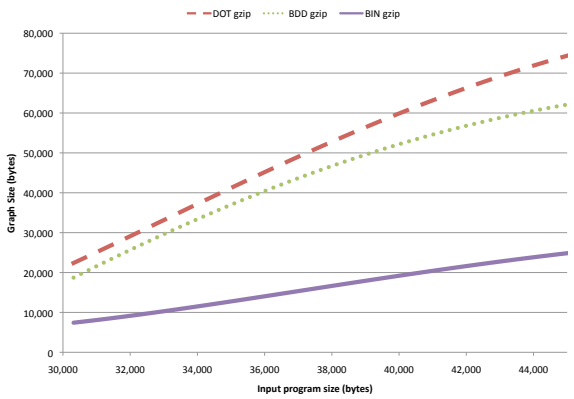


Figure 10: Trend lines for pointer analysis graph size as a function of the input JavaScript file size (`gzip`-ed).

nary format beats the compact bddbddb format in most cases, making us believe that a lightweight staged analysis implementation is a good candidate for being integrated within a Web browser.

5.3 Staged vs. Full Analysis Tradeoff

To fully explore the tradeoff between computing the full analysis on the client and computing part of the analysis on the server and transferring it over the wire to the client, we consider 10 device configurations. These configuration vary significantly in terms of the CPU speed as well as network connectivity parameters. We believe that these cover a wide range of devices available today, from the most underpowered: mobile phones connected over a slow EDGE network, to the fastest: desktops connected over a T1 link.

A summary of information about the 10 device configurations is shown in Figure 11. We based our estimates of CPU multipliers on a report comparing the performance of SunSpider benchmarks on a variety of mobile, laptop, and desktop devices [1]. While not necessarily representative of Web 2.0 application performance [23], we believe these benchmark numbers to be a reasonable proxy for the computing capacity of a particular device.

We compare between two options: 1) performing full analysis on the client and 2) transferring a partial result over the wire and performing the staged analysis on the client. The equation below summarizes this comparison. On the left is the overall time for the full analysis and on the right is the overall time for the staged analysis. B is the bandwidth, L is the latency, b is the main page size, Δ is the incremental JavaScript update size, size is the size of the points-to data needed to run the staged analysis, and F and I are the full and staged analysis times respectively. c is the CPU coefficient from Figure 11:

$$c \times F(b+\Delta) \; ? \; L + \frac{size}{B} + c \times I(\Delta)$$

ID	Configuration Name	CPU coef. c	Link type	Latency L in ms	Bandwidth B in kbps
1	G1	67.0	EDGE	500	2.5
2	Palm Pre	36.0	Slow 3G	500	3.75
3	iPhone 3G	36.0	Fast 3G	300	12.5
4	iPhone 3GS 3G	15.0	Slow 3G	500	3.75
5	iPhone 3GS WiFi	15.0	Fast WiFi	10	75.0
6	MacBook Pro 3G	1	Slow 3G	500	3.75
7	MacBook Pro WiFi	1	Slow WiFi	100	12.5
8	Netbook	2.0	Fast 3G	300	12.5
9	Desktop WiFi	0.8	Slow WiFi	100	12.5
10	Desktop T1	0.8	T1	5	1,250.0

Figure 11: Device settings used for experiments across CPU speeds and network parameters. Devices are roughly ordered in by computing and network capacity. Configurations 1–5 correspond to a mobile setting; configurations 6–10 describe a desktop setting.

Figure 12 summarizes the results of this comparison over our range of 10 configurations. The code analyzed at the server is the bootstrap code from before. Therefore, the points-to data sent to the client was always the same while the size of the incremental code update var-

Incremental size	Configuration (from Figure 11) 1	2	3	4	5	6	7	8	9	10
88	+	+	+	+	+	-	+	+	-	+
619	+	+	+	+	+	-	+	+	-	+
1,138	+	+	+	+	+	-	+	+	-	+
1,644	+	+	+	+	+	-	-	+	-	+
2,186	+	+	+	+	+	-	-	+	-	+
2,767	+	+	+	+	+	-	-	+	-	+
3,293	+	+	+	+	+	-	-	+	-	+
3,846	+	+	+	+	+	-	-	+	-	+
4,406	+	+	+	+	+	-	-	-	-	+
5,008	+	+	+	+	+	-	-	+	-	+
5,559	+	+	+	+	+	-	-	+	-	+
6,087	+	+	+	+	+	-	-	+	-	+
6,668	+	+	+	+	+	-	-	+	-	+
7,249	+	+	+	+	+	-	-	+	-	+
7,830	+	+	+	+	+	-	-	+	-	+
8,333	+	+	+	+	+	-	-	+	-	+
8,861	+	+	+	+	+	-	-	-	-	+
9,389	+	+	+	+	+	-	-	-	-	+
9,917	+	+	+	+	+	-	-	-	-	+
10,445	+	+	+	+	+	-	-	-	-	+
10,973	+	+	+	+	+	-	-	-	-	+
11,501	+	+	+	+	+	-	-	-	-	+
12,029	+	+	+	+	+	-	-	-	-	+
12,557	+	+	+	-	+	-	-	-	-	+
14,816	+	+	+	+	+	-	+	+	+	+
16,485	-	-	-	-	-	-	-	-	-	-
17,103	+	+	+	+	+	-	-	-	-	+
17,909	-	-	-	-	-	-	-	-	-	-
20,197	-	-	-	-	-	-	-	-	-	-
25,566	-	-	-	-	-	-	-	-	-	-
31,465	-	-	-	-	-	-	-	-	-	-
37,689	-	-	-	-	-	-	-	-	-	-
38,986	-	-	-	-	-	-	-	-	-	-
57,254	+	+	+	+	+	-	-	+	-	+
77,074	+	+	+	+	+	-	+	+	-	+
124,136	-	-	-	-	-	-	-	-	-	-
129,739	-	-	-	-	-	-	-	-	-	-

Figure 12: Analysis tradeoff in different environments. "+" means that staged incremental analysis is advantageous compared to full analysis on the client.

Page	Lines of code		Analysis time (seconds)			Full/
	Total	Inc.	Inc.	Full	bddbddb	bddbddb
home	965	339	591	599	4	148
friends	1,274	309	1,297	1,866	6	324
inbox	1,291	17	800	1,840	6	313
profile	1,308	17	851	4,180	6	716

Figure 13: Analysis times for an incrementally loading site (Facebook.com). Each page sends an update to the JavaScript that adds to the previous page's JavaScript.

ied. A + in the table indicates that staged analysis is faster. Overall, we see that for all configurations except 6, 7, and 9, staged analysis is generally the right strategy. "High-end" configurations 6, 7, and 9 have the distinction of having a relatively fast CPU and a slow network; clearly, in this case, computing analysis results from scratch is better than waiting for them to arrive over the wire. Unsurprisingly, the staged approach advocate by GULFSTREAM excels on mobile devices and underpowered laptops. Given the growing popularity of Web-connected mobile devices, we believe that the staged analysis approach advocated in this paper will become increasingly important in the future.

5.4 Facebook Analysis Experiment

Thus far, we have shown how GULFSTREAM performs when run on smaller JavaScript fragments, chosen to simulate incremental code updates of varying sizes. To see how GULFSTREAM handles a real incrementally loading JavaScript program, we captured an interactive session of Facebook usage. JavaScript code was incrementally loaded as the user interacted with the page. We fed each JavaScript update through GULFSTREAM to simulate a user navigating Facebook with GULFSTREAM updating analysis results, as more JavaScript is loaded into the browser.

The navigation session we recorded and used comprises a particular interaction with the main Facebook page. We were careful to use actions such as link clicks that do *not* take the user away from the current page (that would create a new, clean JavaScript engine context). The recorded interaction is clicking several links in a row that keeps the user at the same page. The user starts at the homepage where a large amount of JavaScript is downloaded. Then the user clicks on their friends link, which causes more JavaScript to be downloaded and the page to be updated. The same happens when the user then clicks on their inbox link, and their profile link. These four pages: the *homepage*, *friends*, *inbox*, and *profile* make up our Facebook staged analysis experiment.

In the experiment, each of the four pages was processed by the staged analysis, the full analysis, and the bddbddb analysis. For the staged analysis, the code from the initial page was considered an incremental update upon the bootstrap JavaScript that includes our native environment definition and our browser environment definition. Then in all subsequent pages, the code downloaded for that page was considered an incremental update upon the already computed results from the previous page. For the full analysis and the bddbddb analysis, each of the four pages was analyzed in isolation.

Figure 13 contains the sizes of the pages used in this experiment. The lines of code reported is the line count from the files as they are when downloaded. Note that in many cases, code includes long `eval` statements that expand to many hundreds to thousands of lines of code. For example, the incremental part of the first page expands to over 15,000 lines of code and the incremental part of the last page expands to over 1,400 lines of code.

The goal for the staged analysis is to perform updates to the analysis faster than it takes to recompute the entire analysis. This is highly desirable since the updates in this case are small and rerunning a full analysis to analyze just a few new lines of code is highly wasteful. Figure 13 confirms this intuition by comparing staged analysis against full analysis. Additionally Figure 14 shows the time savings in seconds for each of the device and network configurations from Figure 11. In every configuration the staged analysis fares better, leading to savings on the order of 5 minutes or more.

However, the bddbddb full analysis outperforms the staged analysis by several orders of magnitude, as shown in the last column in Figure 13. This is because the bddbddb analysis solver is highly optimized to scale well and because of our choice of an efficient variable order for BDD processing [25]. While this experiment shows that our staged analysis is better than our full analysis, it also shows that the highly optimized bddbddb-based technique is significantly better for analyzing the code quickly; this is in line with what has been previously observed for Java and C, when comparing declarative vs. hand-written implementations. It should also be noted that the JavaScript for these pages is more complex than in our hand-crafted and procedurally generated files used for other experiments, which produces more complex constraints and favors the more scalable bddbddb-based approach. However, running a highly optimized Datalog solver such as bddbddb within the browser might prove cumbersome for other reasons such as the size and complexity of the code added to the browser code base.

6 Related Work

In this section, we focus on static and runtime analysis approaches for JavaScript.

	Configuration (from Figure 11)									
Page	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
home	541	290	291	119	121	5	7	15	5	6
friends	38,083	20,460	20,469	8,516	8,530	554	564	1,133	450	454
inbox	69,685	37,439	37,451	15,589	15,606	1,022	1,035	2,075	827	832
profile	223,029	119,833	119,845	49,920	49,937	3,311	3,323	6,652	2,658	2,663

Figure 14: Time savings in ms from using staged analysis compared to full analysis on Facebook pages (device and network settings are from Figure 11).

6.1 Static Safety Checks

JavaScript is a highly dynamic language which makes it difficult to reason about programs written in it. However, with certain expressiveness restrictions, desirable security properties can be achieved. ADSafe and Facebook both implement a form of static checking to ensure a form of safety in JavaScript code. ADSafe [10] disallows dynamic content, such as `eval`, and performs static checking to ensure the JavaScript in question is safe. Facebook uses a JavaScript language variant called FBJS [12], that is like JavaScript in many ways, but DOM access is restricted and all variable names are prefixed with a unique identifier to prevent name clashes.

A project by Chugh et al. focuses on staged analysis of JavaScript and finding information flow violations in client-side code [8]. Chugh et al. focus on information flow properties such as reading document cookies and changing the locations. A valuable feature of that work is its support for dynamically loaded and generated JavaScript in the context of what is generally thought of as whole-program analysis. Gatekeeper project [14] proposes a points-to analysis based on bddbddb together with a range of queries for security and reliability. GULFSTREAM is in many way a successor of the Gatekeeper project; while the formalism and analysis approaches are similar, GULFSTREAM's focus is on staged analysis.

Researchers have noticed that a more useful type system in JavaScript could prevent errors or safety violations. Since JavaScript does not have a rich type system to begin with, the work here is devising a correct type system for JavaScript and then building on the proposed type system. Soft typing [7] might be one of the more logical first steps in a type system for JavaScript. Much like dynamic rewriters insert code that must be executed to ensure safety, soft typing must insert runtime checks to ensure type safety.

Other work has been done to devise a static type system that describes the JavaScript language [3, 4, 24]. These works focus on a subset of JavaScript and provide sound type systems and semantics for their restricted subsets of JavaScript. As far as we can tell, none of these approaches have been applied to realistic bodies of code. GULFSTREAM uses a pointer analysis to reason about the JavaScript program in contrast to the type systems and analyses of these works. We feel that the ability to reason about pointers and the program call graph allows us to express more interesting security policies than we would be able otherwise.

This work presents staged analysis done on the client's machine to perform analysis on JavaScript that is loaded as the user interacts with the page. A similar problem is present in Java with dynamic code loading and reflection. Hirzel et al. solved this problem with a offline-online algorithm [15]. The analysis has two phases, an offline phase that is done on statically known content, and an online phase done when new code is introduced while the program is running. They utilize their pointer analysis results in the JIT. We use a similar offline-online analysis to compute information about statically known code, then perform an online analysis when more code is loaded. To our knowledge, GULFSTREAM is the first project to perform staged static analysis on multiple tiers.

6.2 Rewriting and Instrumentation

A practical alternative to static language restrictions is instrumentation. Caja [22] is one such attempt at limiting capabilities of JavaScript programs and enforcing this through the use of runtime checks. WebSandbox is another project with similar goals that also attempts to enforce reliability and resource restrictions in addition to security properties [20].

Yu et al. [26] traverse the JavaScript document and rewrite based on a security policy. Unlike Caja and WebSandbox, they prove the correctness of their rewriting with operational semantics for a subset of JavaScript called CoreScript. Instrumentation can be used for more than just enforcing security policies. AjaxScope [17] rewrites JavaScript to insert instrumentation that sends runtime information, such as error reporting and memory leak detection, back to the content provider. Static analysis may often be used to reduce the amount of instrumentation, both in the case of enforcement techniques such as ConScript [19] and regular code execution.

7 Conclusions

Static analysis is a useful technique for applications ranging from program optimization to bug finding. This paper explores staged static analysis as a way to analyze

streaming JavaScript programs. In particular, we advocate the use of combined offline-online static analysis as a way to accomplish fast, online analysis at the expense of a more thorough and costly offline analysis on the static code. The offline stage may be performed on a server ahead of time, whereas the online analysis would be integrated into the web browser. Through a wide range of experiments on both synthetic and real-life JavaScript code, we find that in normal use, where updates to the code are small, we can update static analysis results within the browser quickly enough to be acceptable for everyday use. We demonstrate this form of staged analysis approach to be advantageous in a wide variety of settings, especially in the context of mobile devices.

References

[1] Ajaxian. iPhone 3GS runs faster than claims, if you go by SunSpider. `http://bit.ly/RHHgO`, June 2009.

[2] L. O. Andersen. Program analysis and specialization for the C programming language. Technical report, University of Copenhagen, 1994.

[3] C. Anderson and P. Giannini. Type checking for JavaScript. In *In WOOD 04, volume WOOD of ENTCS. Elsevier, 2004. http://www.binarylord.com/work/js0wood.pdf*, 2004.

[4] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 429–452, July 2005.

[5] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A trace-based JIT compiler for CIL. Technical Report MSR-TR-2010-27, Microsoft Research, March 2010.

[6] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 103–114, 2003.

[7] R. Cartwright and M. Fagan. Soft typing. *ACM SIGPLAN Notices*, 39(4):412–428, 2004.

[8] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2009.

[9] D. Crockford. The JavaScript code quality tool. `http://www.jslint.com/`, 2002.

[10] D. Crockford. AdSafe: Making JavaScript safe for advertising. `http://www.adsafe.org`, 2009.

[11] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz - open source graph drawing tools. *Graph Drawing*, pages 483–484, 2001.

[12] Facebook, Inc. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`, 2007.

[13] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 465–478, 2009.

[14] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.

[15] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29(2):11, 2007.

[16] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proceedings of the International Static Analysis Symposium*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[17] E. Kıcıman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Proceedings of Symposium on Operating Systems Principles*, Oct. 2007.

[18] F. Logozzo and H. Venter. RATA: Rapid atomic type analysis by abstract interpretation- application to JavaScript optimization. In *Proceedings of the International Conference on Compiler Construction*, pages 66–83, 2010.

[19] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[20] Microsoft Live Labs. Live Labs Websandbox. `http://websandbox.org`, 2008.

[21] A. Milanova, A. Rountev, and B. G. Ryder. Precise call graphs for C programs with function pointers. *Automated Software Engineering*, 11(1):7–26, 2004.

[22] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. `http://google-caja.googlecode.com/files/caja-2007.pdf`, 2007.

[23] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Proceedings of the USENIX Conference on Web Application Development*, June 2010.

[24] P. Thiemann. Towards a type system for analyzing JavaScript programs. *European Symposium On Programming*, 2005.

[25] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog and binary decision diagrams for program analysis. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, Nov. 2005.

[26] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of Conference on Principles of Programming Languages*, Jan. 2007.