Gulfstream: Incremental Static Analysis for Streaming JavaScript Applications

Benjamin Livshits
Microsoft Research

Salvatore Guarnieri
University of Washington

January 20, 2010

MSR-TR-2010-4

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. 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, by interacting with the application, more and more code is sent over to the browser.

This paper explores incremental 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 incremental analysis at the expense of a more thorough and costly offline analysis on the static code. We find that in normal use, where updates to the code are small, we can incrementally update static analysis results quickly enough to be acceptable for everyday use. We demonstrate this hybrid approach to be advantageous in a wide variety of settings, especially in mobile devices.

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 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 to the user's browser. In essence, the entire program is never completely available, by interacting with the application, more and more code is sent over to the browser.

In the context of performing static analysis to enforce security properties, for instance, the pattern than becomes clear after analyzing several such large Web 2.0 applications 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. For example, 71% of Facebook code is downloaded right away, 62 more KB of code are downloaded when visiting event pages, etc. Similarly, Bing Maps downloads most of the code right away, however, requesting traffic requires additional code download. More-

over, often the parts of the application that are downloaded later are composed on the client by referencing a third-party library. This can be done to as a way to factor out browser-specific or even user-specific code. Analyzing this code ahead of time may be inefficient or even impossible.

The dynamic nature of JavaScript, combined with incremental code download construction 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 be considered to find targets of the `onclick` handler.

As the example in Figure 1 demonstrates, JavaScript in the browser essentially has a *streaming programming* model: sites insert JavaScript code into the information sent to the user, and the browser is happy to execute any code that comes its way.

GULFSTREAM advocates performing *incremental static analysis* within a web browser. We explore the trade-off between offline static analysis performed on the server and fast, incremental analysis performed in the browser. We conclude that incremental analysis is fast enough, especially on small incremental updates, to be made part of the overall browser infrastructure.

```
<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.

1.1 Contributions

This paper makes the following contributions:

- **Incremental analysis.** With GULFSTREAM, we demonstrate how to build an incremental version of a point-to analysis, which is a building block for implementing static checkers of various sorts. Our analysis is staged, meaning that portions of the analysis can be performed offline on the server, with analysis on code *deltas* being performed in the browser whenever needed.
- **Staged computation.** We demonstrate how to serialize pointer analysis information for staged computation which is done primarily on the server, and is incrementally updated in the browser. For completeness, we compare the results of a hand-coded and a BDD-based pointer analysis implementation and discover that GULFSTREAM represents data more compactly than BDDs.
- **Trade-off.** We use a wide range of JavaScript inputs of various sizes to estimate the overhead of incremental computation. We propose strategies for choosing between staging strategies for various network settings. We explore the tradeoff 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 provide conclusions.

2 Background

This section first provides a background on static analysis and why it is needed, and then talks about code loading in Web applications.

2.1 Static Analysis

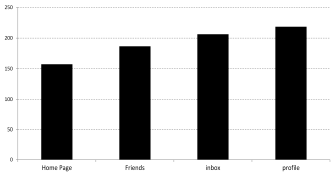
Static analysis has long been recognized as an important building block for achieving reliability, security, and performance. Static analysis may be used to find violations of important reliability properties; in the context of JavaScript, tools such as JSLint [8] fulfill such a role. Soundness in the context of static analysis gives us a

Page visited or action performed	Added JavaScript files	KB
FACEBOOK FRONT PAGE		
Home page	19	157
Friends	7	186
Inbox	1	206
Profile	1	219
FACEBOOK SETTINGS PAGE		
Settings—Network	13	136
Settings—Notifications	1	137
Settings—Mobile	3	151
Settings—Language	1	152
Settings—Payments	0	152
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

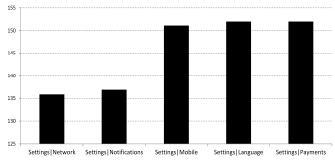
Figure 2: Incremental loading of Facebook and OWA JavaScript code.

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 [7, 15]. 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, knowledge of types may be used to improve the performance of runtime tracing [14] within the JavaScript JIT.

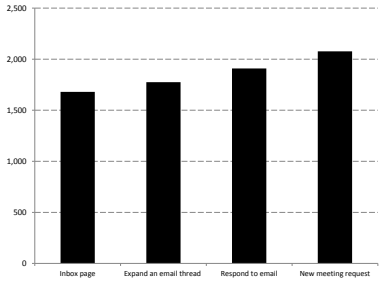
Several broad approaches exist in the space of static analysis. While some recent static analysis in type inference have been made for JavaScript [18], the focus of this paper is on *pointer analysis*. The goal of pointer analysis is to answer the question "given a variable, what heap objects may it 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 program below



(a) Facebook code growth over time (front page).



(b) Facebook code growth over time (settings page).



(c) Outlook Web Access code growth over time.

Figure 3: CDF of incremental code loading.

```
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 an 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 [15]. Gatekeeper implements a form of inclusion-based Andersen-style context-insensitive pointer analysis [2], which shows good scala-

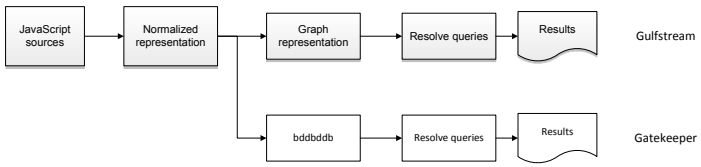


Figure 4: GULFSTREAM architecture and a comparison with the Gatekeeper project.

bility 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 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 this can be formulated as statically resolved heap reachability queries.

2.2 Code Loading in Web Applications

Web 2.0 programs are inherently streaming, which is to say that they are downloaded over time. We performed a small study 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, over time attempt to use more and more application features, paying attention to how much extra JavaScript code is downloaded to the user's browser. Note that we take care not to change the URL, which would invalidate the JavaScript context.

As Figure 2 and 3 demonstrate, much of the code is downloaded initially. However, as the application is getting used more and more, quite a bit of extra code is sent to the browser. For instance, the "account settings" functionality in the case of Facebook and the "new meeting request" in the case of OWA account for close to 200 KB of JavaScript code. In the case of Facebook, we end up with more than twice the amount of initial code once we have used the application for a while. OWA, in contrast, is a little more monolithic, growing by about 23% 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 main "initial" portion of the application on the server and analyzing the rest of the code on-the-fly is a reasonable alternative in this highly dynamic environment.

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 in Guarnieri et al. [15]. Somewhat surprisingly, 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. Figure 4 summarizes GULFSTREAM approach and shows how it compares to the Gatekeeper strategy.

Soundness. In this paper we do not focus on the issue of analysis soundness. Soundness would be especially important for a tool designed to look for security vulnerabilities, for instance. 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 $\text{JavaScript}_{\text{SAFE}}$ advocated in Guarnieri et al. [15], 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 find them to be orthogonal to incremental 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 undesirable annoyance to the user and, as such, need to be prevented in third-party code. The second looks for calls to `setInterval` with non-function parameters. This is effectively a commonly overlooked form of dynamic code loading similar to `eval`.

Incremental analysis. A website is often built from many sources. The very nature of JavaScript encourages scripts from various sources to be put together to create a website. We have noticed that when a website incrementally loads JavaScript, most of a webpage's JavaScript can be determined statically before it is sent to a browser. We noticed this on dynamic sites such as Facebook and OWA 2. As the user interacts with the website, updates to the JavaScript are sent to update the website. If the updates to the website's JavaScript are small, it would make sense that an incremental analysis would perform better than a full program analysis. We

$s ::=$	
ϵ	[EMPTY]
$s;\ s$	[SEQUENCE]
$v_1 = v_2$	[ASSIGNMENT]
$v = \bot$	[PRIMASSIGNMENT]
`return` v;	[RETURN]
$v =$ `new` $v_0(v_1, ..., v_n)$;	[CONSTRUCTOR]
$v = v_0(v_{this}, v_1, v_2, \ldots, v_n)$;	[CALL]
$v_1 = v_2.f$;	[LOAD]
$v_1.f = v_2$;	[STORE]
$v =$ `function`$(v_1, ..., v_n)\ \{s;\}$;	[FUNCTIONDECL]

Figure 5: JavaScript$_{\text{SAFE}}$ statement syntax in BNF.

looked at range of update sizes to identify when an incremental analysis is faster or when recomputing the full program analysis is faster. Full program analysis might be faster because there is book keeping and graph transfer time in the incremental analysis that is not present in the full program analysis.

Section 5 talks about advantages of incremental analysis in detail. In general, we find it to be advantageous in most settings, especially on slower mobile connections.

4 Techniques

The first analysis stage is normalizing the program representation. Based on this normalized representation, we built two analyses. The first is the bddbddb-based points-to analysis described in Gatekeeper [15]. The other is a hand-coded representation of points-to information using graphs as described below. 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. The graph-based representation also produces graphs that can efficiently compressed and transferred to the browser from the server.

4.1 Normalization

The first analysis stage is normalizing the program representation and is borrowed from the Gatekeeper project and is shown in Figure 5. As can be seen from the figure, the original program statements are broken down to their simpler versions, with temporaries introduced as necessary. Here is a normalization example that

Node type	Description	Node shape	Edge
Variable	The basic node is a simple variable node. It represents variables from the program or manufactured during normalization. Line 1 in Figure 7 has two variable nodes, `A` and `Object`. Flow edges from variable nodes to other variable nodes are represented as solid black lines and represent assignments.	Oval	Solid edges
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 7 when a new `Object` is created.	Rectangle	Dashed edges
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	Solid edges marked with field name
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 and are connected to their function parent by dotted edges. Lines 7 and 8 from Figure 7 show a function node being created and used. Return values are also represented by this type of node.	Pentagon	Dotted edges

Figure 6: Description of nodes types in the graph.

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

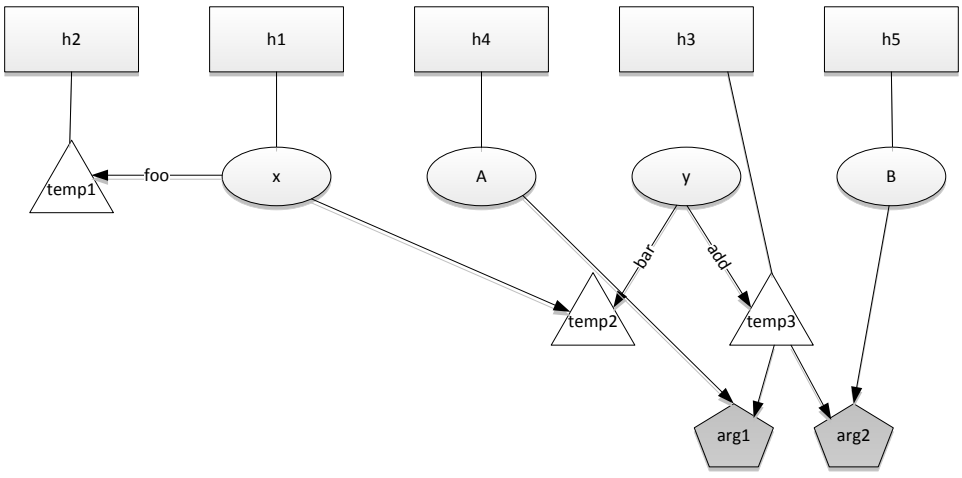
The points-to information is calculated from a graph representing the program stored in memory. The graph is generated from the normalized program. Assignments turn into edges, field accesses turn into named edges, constructor calls

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 6. We use the program and corresponding graph in Figure 7 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

```
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 7: Program with a function call.

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 incremental 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 [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 incremental 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.

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. There would be a cost to unzip each file and this must be added to the time needed to transfer the graph. 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, which means that with an efficient unzip routine, the graph output format does not make much of a difference on the incremental analysis time on a fast link. A more detailed comparison of graph representation sizes is presented in Section 5.

$pointsTo = \emptyset \mapsto \emptyset$ // *points-to map*
$reversePointsTo = \emptyset \mapsto \emptyset$ // *reverse version of points-to map*
inc_insert(G, e) // *incrementally update points-to map*

1: $invalid = \emptyset$
2: **if** $e.src \in G$ **then**
3: invalidate(e.src)
4: **end if**
5: **if** $e.dst \in G$ **then**
6: invalidate(e.dst)
7: **end if**
8: $G = \langle G_N \cup \{e_{src}, e_{dst}\}, G_E \cup \{e\} \rangle$
9: **for all** $n \in invalid$ **do**
10: ans =compute-points-to$(n, \emptyset)$
11: $pointsTo[n] = pointsTo[n] \cup ans$
12: **for all** $h \in ans$ **do**
13: $reversePointsTo[h] = reversePointsTo[h] \cup n$
14: **end for**
15: **end for**
16: return G

invalidate($n \in G_N$) // *recursively invalidate following flow edges*

1: **if** $n \in$ invalid **then**
2: return
3: **end if**
4: $invalid \leftarrow invalid \cup \{n\}$
5: **for all** n' adjacent to n **do**
6: **if** $n \rightarrow n'$ is an assignment edge **then**
7: invalidate(n')
8: **end if**
9: **end for**
10: return G

Figure 8: Routines inc_insert and invalidate.

4.4 Analysis

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, and 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 [5, 15, 25].

GULFSTREAM maintains a graph representation which is being updated as more of the program is processed. Figure 8 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 in graph G, 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 9 shows the pseudo-code version 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 incremental analysis where the $invalid$ worklist is set to be all nodes in the graph G_N.

```
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 noden′ ∈ 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 n
 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 9: Points-to computation algorithm.

Figure 9 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 10 shows our formulation of the queries. The $\texttt{detect} - \texttt{alert} - \texttt{calls}$ 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 $\texttt{detect} - \texttt{set} - \texttt{interval} - \texttt{calls}$ 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 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.

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

```
detect-set-interval-calls()
 1: n = setInterval.arg1
 2: for all edge e entering n do
 3:    if e.src == Function.return then
 4:       return true
 5:    else
 6:       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.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 10: Queries detect-alert-calls and detect-set-interval-calls.

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 $\texttt{detect} - \texttt{alert} - \texttt{calls}$ from Figure 10 to return `false`. If $\texttt{detect} - \texttt{alert} - \texttt{calls}$ ever returns true, the analysis engine could either notify the user or stop the offending page from executing.

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

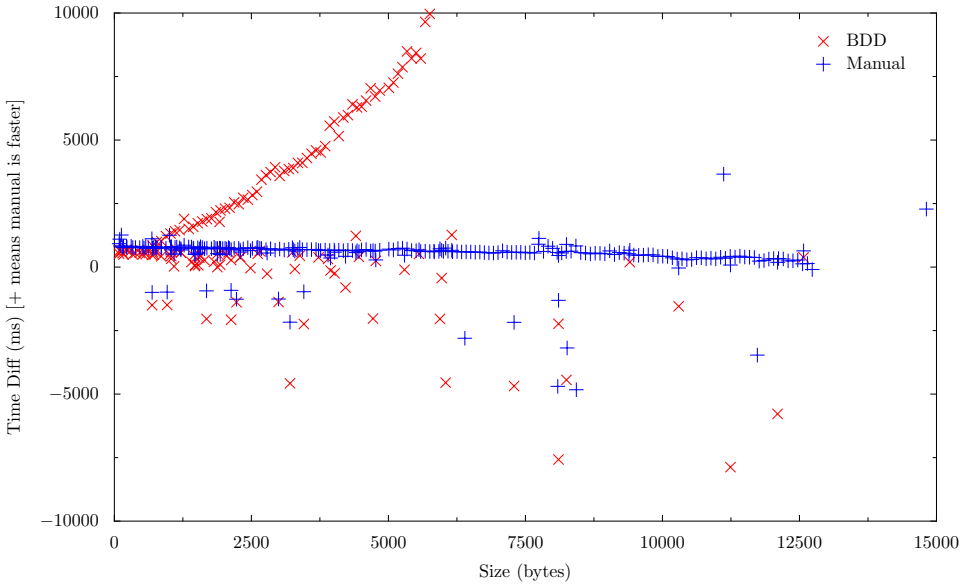


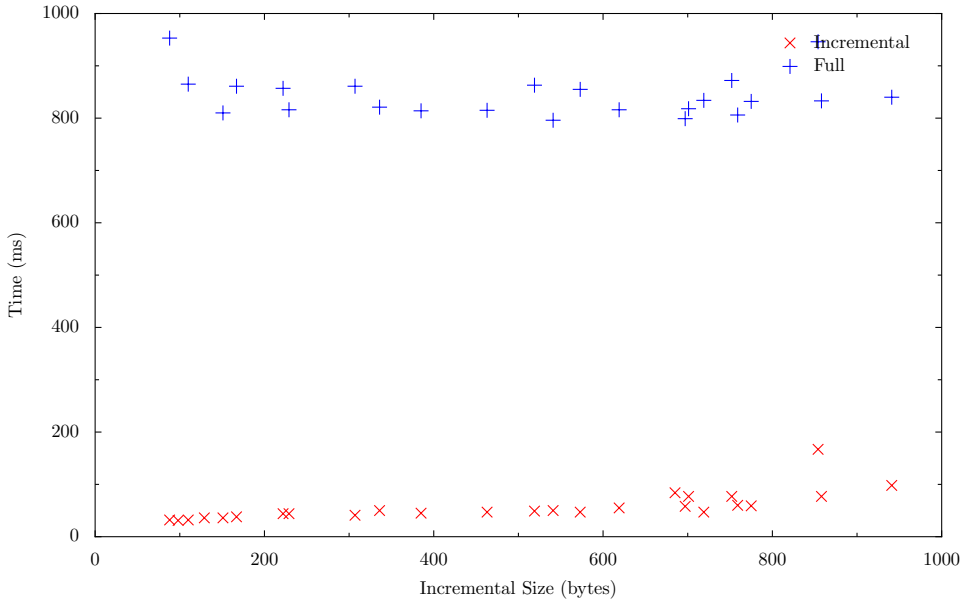
Figure 11: Running time difference between hand-coded and bddbddb-based implementation as a function of the input size.

wire. Our measurements 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 scraped from Google code search.

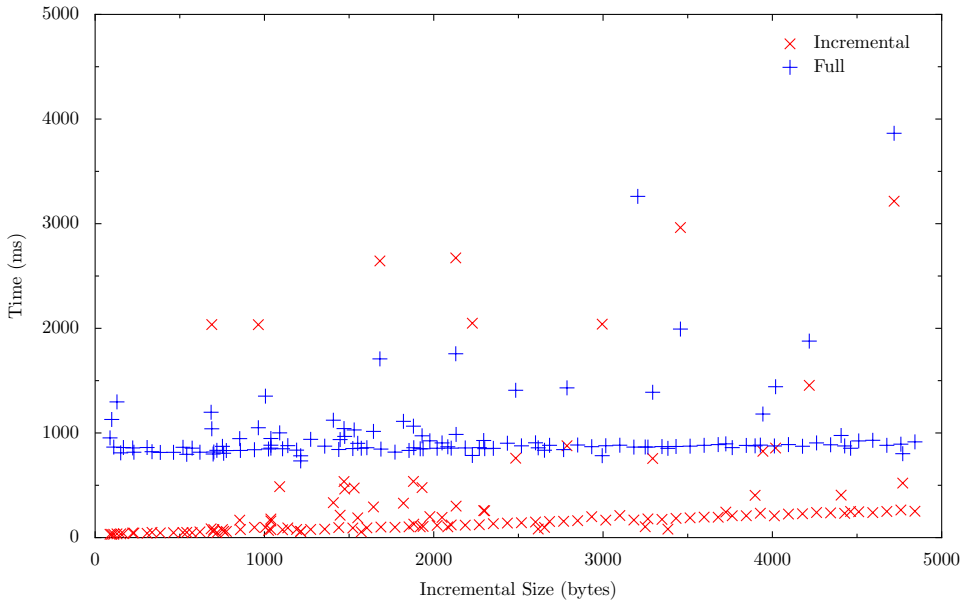
5.1 Analysis Running Time

Figure 12 shows both full and incremental analysis on the same scale. For this experiment, in the case of incremental analysis we used a base of 30 KB of code. We see that incremental analysis is consistently faster than full analysis. In the cases of smaller incremental updates, the difference in running times can be as significant as a couple of orders of magnitude.

Figure 11 compares how both the GULFSTREAM hand-coded full points-to analysis and the bddbddb analysis time compare against the incremental analysis. The graph shows the difference in time, with positive values meaning that incremental analysis is faster. We see that the full analysis is faster by a more or less constant value. We also see that incremental analysis appears to scale better for larger updates exceeding 5 KB of code. This is encouraging: it means that we can implement the hand-coded 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

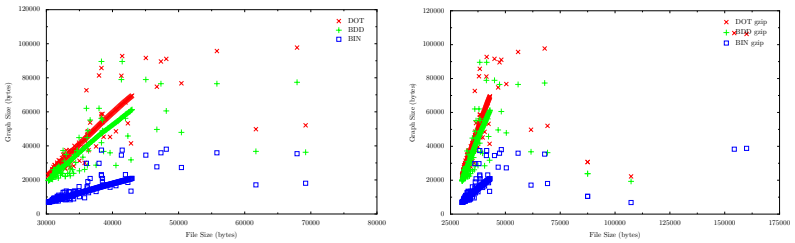


(a) Zoom 0-1 KB



(b) Zoom 0-5 KB

Figure 12: Running times for full and incremental analyses in ms as a function of the input size.



(a) Zoom (b) Full Graph

Figure 13: Pointer analysis graph size as a function of the input JavaScript file size (`gzip`-ed).

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 14: Device settings used for experiments across CPU speeds and network parameters. Devices are roughly ordered in by computing and network capacity.

of BDDs.

5.2 Space Considerations

Figure 13 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. The graph shows both a zoomed-in version ranging from 30,000 bytes to 80,000 bytes and a version ranging from 0 bytes to 175,000 bytes. All numbers presented in the figure are after applying industry-standard gzip compression.

We were not surprised to discover that the DOT version is most verbose, even after gzip has been applied. To our surprise, our simple binary format beats the compact bddbddb format in most cases, making us believe that a lightweight hand-coded analysis implementation is a good candidate for being integrated within a

Graph	Incremental	Settings									
Graph	Size	1	2	3	4	5	6	7	8	9	10
6,914	88	+	+	+	+	+	-	+	+	-	+
7,608	619	+	+	+	+	+	-	-	+	-	+
8,332	1,138	+	+	+	+	+	-	-	+	-	+
11,045	1,644	+	+	+	+	+	-	-	-	-	+
9,400	2,186	+	+	+	+	+	-	-	+	-	+
10,058	2,767	+	+	+	+	+	-	-	-	-	+
12,846	3,293	+	+	+	+	+	-	-	-	-	+
11,269	3,846	+	+	+	+	+	-	-	-	-	+
12,494	4,406	+	+	+	+	+	-	-	-	-	+
12,578	5,008	+	+	+	+	+	-	-	-	-	+
9,526	5,559	+	+	+	+	+	-	-	-	-	+
13,788	6,087	+	+	+	+	+	-	-	-	-	+
14,447	6,668	+	+	+	+	+	-	-	-	-	+
15,095	7,249	+	+	+	+	+	-	-	-	-	+
15,751	7,830	+	+	+	+	+	-	-	-	-	+
16,306	8,333	+	+	+	+	+	-	-	-	-	+
16,866	8,861	+	+	+	+	+	-	-	-	-	+
17,413	9,389	+	+	+	+	+	-	-	-	-	+
17,969	9,917	+	+	+	+	+	-	-	-	-	+
18,520	10,445	+	+	+	-	+	-	-	-	-	+
19,075	10,973	+	+	+	-	+	-	-	-	-	+
19,633	11,501	+	+	+	-	+	-	-	-	-	+
20,184	12,029	+	+	+	-	+	-	-	-	-	+
20,750	12,557	-	-	+	-	+	-	-	-	-	+
34,570	14,816	+	+	+	+	+	-	-	+	-	+
27,699	16,485	-	-	-	-	-	-	-	-	-	-
35,941	17,103	-	-	+	-	+	-	-	-	-	+
38,054	17,909	-	-	-	-	-	-	-	-	-	-
27,296	20,197	-	-	-	-	-	-	-	-	-	-
35,945	25,566	-	-	-	-	-	-	-	-	-	-
17,108	31,465	-	-	-	-	-	-	-	-	-	-
35,411	37,689	-	-	-	-	-	-	-	-	-	-
18,056	38,986	-	-	-	-	-	-	-	-	-	-
10,488	57,254	+	+	+	+	+	-	-	-	-	+
6,836	77,074	+	+	+	+	+	-	-	+	-	+
38,310	124,136	-	-	-	-	-	-	-	-	-	-
38,804	129,739	-	-	-	-	-	-	-	-	-	-

Figure 15: Analysis tradeoff in different environments. "+" means that staged incremental analysis is advantageous compared to full analysis on the client.

web browser.

5.3 Incremental vs Full Analysis Tradeoff

To fully explore the tradeoff between computing analysis results on the client and transferring it over the wire, we consider 10 device configurations. These configuration vary 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 14. 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]. We believe these benchmark numbers to be a reasonable proxy for the over overall computing capacity of a particular device.

We compare between two options: 1) performing full analysis on the client and 2) transferring a partial result across the wire and performing incremental analysis on the client. The equation below summarizes this comparison with B being the bandwidth, L being the latency, b being the main page, Δ being the incremental JavaScript update, size being the size of the points-to data needed to run the incremental analysis, and F and I being the full and incremental analysis respectively.

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

Figure 15 summarizes the results of this comparison over our range of 10 configurations. A $+$ indicates that incremental analysis is faster. Overall, we see that for all configurations except 6, 7, and 9, incremental 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 incremental approach advocate by GULFSTREAM excels on mobile devices and underpowered laptops. Given the growing popularity of web-connected mobile devices, we believe that the incremental analysis approach advocated in this paper will become increasingly important in the future.

6 Related Work

In this section, we focus on static and runtime analysis approaches for JavaScript. The approaches accomplish their analysis through the use of type systems, lan-

guage restrictions, and modifications to the browser or the runtime. We describe these strategies in turn below.

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 [9] 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 with other FBJS programs on the same page.

An interesting recent development in JavaScript language standards committees is the strict mode (`use strict`) for JavaScript [10], page 223. Strict mode accomplishes many of the goals that many current analyses try to achieve: `eval` is largely prohibited, bad coding practices such as assigning to the `arguments` array are prevented, `with` is no longer allowed, etc.

A project by Chugh et al. focuses on staged analysis of JavaScript and finding information flow violations in client-side code [7]. 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 [15] 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 incremental 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 [6] 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 incremental 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 [16]. 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. While the techniques in this paper are somewhat similar, to our knowledge, Gulfstream is the first project to perform staged static analysis on multiple tiers (server and client-side browser).

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. traverse the JavaScript document and rewrite based on a security policy [26]. Unlike Caja and WebSandbox, they prove the correctness of their rewriting with operational semantics for a subset of JavaScript called CoreScript. BrowserShield [23] similarly uses dynamic and recursive rewriting to ensure that JavaScript and HTML are safe, for a chosen version of safety, and all content generated by the JavaScript and HTML is also safe. Instrumentation can be used for more than just enforcing security policies. AjaxScope [19] rewrites JavaScript to insert instrumentation that sends runtime information, such as error reporting and memory leak detection, back to the content provider.

We feel that sound static analysis may provide a more systematic way to reason about what code can do, especially in the long run, as it pertains to issues of security, reliability, and performance. While the soundness of the native environment and exhaustiveness of our runtime checks might be weak points of our approach, we feel that we can address these challenges as part of future work.

6.3 Runtime and Browser Support

Current browser infrastructure and the HTML standard require a page to fully trust foreign JavaScript if they want the foreign JavaScript to interact with their site. The alternative is to place foreign JavaScript in an isolated environment, which disallows any interaction with the hosting page. This leads to web sites trusting untrustworthy JavaScript code in order to provide a richer web site. One solution to get around this all-or-nothing trust problem is to modify browsers and the HTML standard to include a richer security model that allows untrusted JavaScript controlled access to the hosting page.

MashupOS [17] proposes a new browser that is modeled after an OS and modifies the HTML standard to provide new tags that make use of new browser functionality. They provide rich isolation between execution environments, including resource sharing and communication across instances. In a more lightweight modification to the browser and HTML, Felt et al. [13] add a new HTML tag that labels a `div` element as untrusted and limits the actions that any JavaScript inside of it can take. This would allow content providers to create a sand box in which to place untrusted JavaScript.

7 Conclusions

Static analysis is a useful technique for applications ranging from program optimization to bug finding. This paper explores incremental 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 incremental 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, we find that in normal use, where updates to the code are small, we can incrementally update static analysis results quickly enough to be acceptable for everyday use. We demonstrate this hybrid staged analysis approach to be advantageous in a wide variety of settings, especially in mobile devices.

References

[1] Ajaxian. `http://ajaxian.com/archives/iphone-3gs-runs-faster-than-claims-if-you-go-by-sunspider`, 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 *In Proceedings of the European Conference on Object-Oriented Programming*, pages 429–452, July 2005.

[5] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 103–114, 2003.

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

[7] 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.

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

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

[10] ECMA. Ecma-262: Ecma/tc39/2009/025, 5th edition, final draft. `http://www.ecma-international.org/publications/files/drafts/tc39-2009-025.pdf`, Apr. 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. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to strangers without taking their candy: isolating proxied content. In *Proceedings of the Workshop on Social Network Systems*, pages 25–30, 2008.

[14] 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 *PLDI*, pages 465–478, 2009.

[15] 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.

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

[17] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[18] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium, SAS '09*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[19] 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.

[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. *Autom. Softw. Eng.*, 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] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2006.

[24] P. Thiemann. Towards a type system for analyzing JavaScript programs. 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.