Dynamic Information Flow Labeling in Javascript
Alan Cleary and Seth Just
August 5, 2011

1 Introduction
Clientside scripting languages such as JavaScript are ubiquitous in modern, internet-connected computing, but pose a definite security risk to those who allow their execution. The widespread inclusion of third-party scripts into major websites increases the risks of malicious scripts interfering with the desired behavior of a page, and consequently decreases the level of security available to web users. While a variety of security mechanisms do exist in current browser environments (e.g. the “same origin policy”, SSL certificates, etc.) these approaches often lack the flexibility necessary to cope with real-world threats. Furthermore, these measures fail to fully account for subtle threats posed by JavaScript, such as unwanted information access, as in cross-site request forgery (CSRF) and cross-site scripting (XSS). Recent academic attention focusses on increasing the security of JavaScript, either by reducing its functionality (for example Facebook JavaScript (FBJS) or Google’s Caja) or by augmenting the execution environment. The latter can take a wide variety of forms (viz. Section 2)

This paper presents a novel approach to instrumenting a JavaScript interpreter, using a hybrid of static and dynamic approaches to label values during the course of execution. Our work dynamically tracks direct information flow during execution, as well as utilizing static analysis to support dynamic context tracking for intra- and inter-procedural implicit information flow tracking. Furthermore, this flow tracking handles complex features of JavaScript that have caused issues in the past, such as eval [viz. 5] and dynamic dispatch on a prototype chain.

General Terms JavaScript
Keywords Control flow

2 Related Work
Hammer et al. [5] present a policy enforcement mechanism for JavaScript based on monitoring and history collection. Dynamic, user-specified policies dynamically determine actions to be taken in response to an execution history. They utilize a system to roll-back rejected histories to ensure compatibility with (well-behaved) legacy scripts. However, they do not make use of full label tracking as our system does, and in fact mention its potential utility, particularly in handling eval.

Austin and Flanagan [2] presents two complete semantics for label propagation for a version of the lambda calculus with references, one that assigns labels explicitly to our values and one that leaves them implicit until information flows between contexts. Our work closely mirrors the former semantics, but introduces mechanisms to handle eval and prototype inheritance, as well as applying such a labeling system to a real-world language implementation. However, their sparse labeling semantics suggest a method to reduce the overhead inherent in our implementation.

In [3], Austin and Flanagan extend their previous work in labeling by replacing the no-sensitive-upgrade rule (to prevent half-bit leaks from branches) with a permissive-upgrade rule that allows for the management of partially-leaked information. This work suggests a fruitful extension of our system, which does not account for half-bit leaks.

Chandra and Franz [4] use a hybrid of static and dynamic analysis to instrument compiled Java code to label values, respecting both explicit and implicit flows. Their static analysis annotates code with information regarding the action of both sides of a code branch, to prevent half-bit leaks, while ours focuses only on control flow for context tracking.

Yu et al. [10] present a system of instrumentation for JavaScript code running in a web browser that relies on pre-execution code rewriting to allow for the instrumentation of dynamically generated code, as well as changing potentially unsafe operations to only execute after approval by a security automaton.
Schneider [8] defines a class of security policies that base their decisions only in observing the history of steps in the execution of a program. He establishes that policies enforceable through such a mechanism are a subset of safety policies, which reject those programs for which some “bad thing” happens in the course of execution.

Askarov and Sabelfeld [1] deal with enforcing security conditions over a two-element security lattice in a language with declassification primitives, as well as eval. They offer proof that their presented monitor configurations enforce security conditions based on the indistinguishability of initial memories.

Xin and Zhang [9] develop a mechanism for control dependence detection utilizing immediate post-dominator information for a program’s flowgraph, with dynamic code, running at each program node that maintains a control dependence stack. While their work does not focus on information flow labeling, we use their mechanism to handle both intra- and inter-procedural context tracking for implicit flow labeling.

3 Problem Statement

Due to a constant drive for dynamic and interactive content and functionality, modern web pages contain an array of JavaScript components that are loaded from a variety of providers, including third parties unrelated to the primary content provider of a page. These components may come from a combination of trusted and untrusted sources that the browser regulates using the “same origin policy” which restricts scripts’ access privileges to data and methods from the same origin, typically a webserver or domain. While this policy provides a level of security in JavaScript execution, it does not prevent JavaScript attacks from taking place and only provides coarse-grained control over the behavior of scripts.

One approach users take to further prevent attacks is to install an ad blocker or other third party protection software. Unfortunately these often rely on depreciated methods such as black listing or are disabled as a result of affecting website functionality. Also, these programs are often incapable of recognizing attacks that use unconventional methods. Of course all this is unknown to the layperson; The most vulnerable demographic.

The logical way to prevent attacks without instilling obligations on the user and minimizing overhead would be to integrate a complete information flow labeling mechanism into the web browser. This would be the ideal foundation for a powerful integrated policy enforcement mechanism.

4 Methods

We have implemented a labeling system in WebKit’s JavaScript engine, based loosely on the work of Austin and Flanagan [2] in universal labeling and Chandra and Franz [4] in using both static and dynamic approaches to instrument unmodified code. By extending a browser to track the flow of information from various locations, both local and remote, we hope to extend the current capabilities of JavaScript security systems. Particularly, the JavaScript-focused work reviewed in Section 2 [5, 10] seems to have an emphasis on policy specification and enforcement, while lacking robust mechanisms to track information flow, which has the potential to greatly increase the utility of any enforcement mechanism. For example, Hammer et al. [5, p. 10] point to the difficulty of handling eval statements in their current system – an issue that they claim could be mitigated with proper information flow tracking.

In direct contrast to this policy-facing approach, Austin and Flanagan focus solely on propagating labels dynamically and correctly, although in the context of an extension of lambda calculus. This work was extremely useful in our implementation, as their semantics guarantee correctness for both universal and sparse labeling, as well as addressing complex language features. Furthermore, the natural development from universal to sparse labeling in their work allows us a simple way to improve performance of our system while maintaining this correctness property. However, in terms of concrete implementation, we also find inspiration in Chandra and Franz [4]; although their work is aimed at the JVM, it introduces a static/dynamic hybrid mechanism for flow tracking, as well as offering much insight into implementing information flow tracking systems in real-world language systems.

4.1 Direct Flow

Our system handles direct flow within by updating labels on modified values with the labels of those values that influence the final result. While this form of label propagation is simple to understand and implement, it lays the groundwork for handling complex information flow.
4.2 Implicit Flow

Besides direct flow tracking, we use a system of control dependence tracking based on the work Xin and Zhang [9]. Their algorithm relies on pre-runtime static analysis to generate a post-dominator tree and subsequent runtime maintenance of a control dependence stack. As in the work of Sabelfeld and Russo [7] and Austin and Flanagan [2], we utilize this stack to track a history of security contexts. Whenever values are created/modified at runtime their labels are joined with the set of labels currently on the stack, thus propagating labels for implicit flow. Because we never need access to individual labels on the stack, when pushing label \( l \) to the stack we join \( l \) to the current label at the head and write that value as the new head. This guarantees that the head of the stack represents the join of all the labels currently on the stack, avoiding a linear-time traversal each time we consult the stack for the current context.

4.2.1 Static Analysis

Our implicit flow tracking relies on statically generating control flow information before running a code block. Before executing any code we generate a control flow graph for that code. This graph is then processed into a post-dominator tree using the fast algorithm presented by Lengauer and Tarjan [6]. This tree is then stored as a table mapping nodes to post-dominators for later access.

A short piece of code, its flow graph and post-dominator tree are shown in Figure 1.

4.2.2 Intraprocedural Context Tracking

Within a method our context tracking operates simply, using the algorithm presented by Xin and Zhang [9], with a straightforward extension to track labels. Prior to execution we store a table of post-dominators for each opcode within a method’s code block, as discussed in Section 4.2.1. This table is generated using the fast dominator algorithm of Lengauer and Tarjan [6] from a control flow graph.

During execution we periodically call two methods to maintain our context stack, which (for the strictly intraprocedural case) is a stack of (label, post-dominator) pairs. Pseudocode for these methods are shown in Figure 2. The first is called at branching opcodes (those which have more than child in the control flow graph), while the second is called before executing each opcode, as any opcode can be an immediate post-dominator. These methods require access to the context stack (\( \text{programCounter} \)), immediate post-dominator table (\( \text{idom} \)) and the current instruction pointer (\( \text{vPC} \)), and our branch method takes a context label as an argument.

As shown by Xin and Zhang, these algorithms correctly track control dependence, so when any given line of code executes, \( \text{programCounter} \) contains exactly the labels of the values on which that line’s execution depends.

4.2.3 Interprocedural Context Tracking

Our interprocedural context tracking, as in Xin and Zhang [9], subsumes the intraprocedural approach. Because the intraprocedural approach guarantees that the entry and exit of a method will operate in the same context (see Xin and Zhang [9], Theorem 4), we can maintain our context stack across function calls. However, while any push during a function call will be matched with a pop, while in a function call we must insure that we do not join to a label pushed before the call or pop a label pushed before the call. This is accomplished by extending the context stack to hold (label, post-dominator, call-stack-pointer) triples and modifying the branch and merge point algorithms as shown in Figure 3.

Note that within a single method this approach is equivalent to the intraprocedural case, as the
\( x = 5; \)
while (x) {
    x--; 
    y = 2; 
    while (y--) {
        if (x == 3) {break; }
        print(x)
    }
}

(a) Code

(b) Control Flow Graph

(c) Post-Dominator tree

Figure 1: A short JavaScript method and its associated static analysis. Note that the control flow graph is only used to generate post-dominators.
Figure 3: Algorithms for branch and merge points in purely interprocedural context tracking, where programCounter is the context stack, idom is the post-dominator table, callStack is the call stack pointer, _ is a wildcard and | = is the join-and-assign operator on labels.

callStack does not change within a single procedure. Thus calling the interprocedural methods in the same manner as the intraprocedural ones guarantees full control context tracking. However, JavaScript’s object-oriented and functional nature allows (or necessitates) one extension to fully propagate labels. Because functions are objects they have associated labels; thus a function should execute in a context that contains its label. To accomplish this we simply push that label to the context stack before executing a function, and then pop it after returning.

4.3 String eval

We handle eval similarly to a function call, but because no function object is created, we apply the label of the string to the context stack before transferring control. This accomplishes the desired behavior, and for all other purposes eval is treated like a function call.

4.4 Prototypes

JavaScript, while object oriented, has no classing or inheritance system. Instead it makes use of prototypes; each object has a special pointer to a prototype object, and any attribute lookup that fails is repeated on the prototype, thus forming a chain of objects that may be consulted on any lookup. Furthermore, prototypes can be reassigned, or an object’s prototype can be changed between lookups. This dynamism creates a difficult situation for information labeling, as any object’s properties can be changed in a wide variety of ways. To ensure that values are appropriately labeled, even through prototype chain lookups, we accumulate labels over the course of a lookup, and any returned value has the union of these labels attached, as shown in Figure 4.

5 Implementation

Similar to the work done by Hammer et al. [5], we instrumented WebKit’s JavaScript engine, JavaScriptCore (JSC). This consisted of creating eight classes, two for static analysis, three for labeling, two for our program counter, and a logger for debugging and documentation. We also instrumented six classes with 1,411 lines of code. JSC’s Interpreter class has the most modification with instrumentation in nearly all the opcodes and custom macros for banching and merging operations. With 2,537 total lines of code, our instrumentation is very unobtrusive compared to WebKit’s nearly 3.5 million lines.

5.1 Static Analysis

JavaScript code is fed into JSC which then compiles it into a bytecode code block which is sent to the interpreter for evaluation. Our implementation statically analyzes code blocks before evaluation in our instrumented interpreter.
The static analyzer was implemented using a StaticAnalyzer class and FlowGraph class. StaticAnalyzer determines the post-dominators for each opcode in a code block as discussed in Section 4.2.1 and stores them. StaticAnalyzer objects are stored by code blocks to avoid repeated generation of post-dominator information, but the analysis is performed immediately before a code block is executed.

5.2 Labeling

Similar to the labeling technique used by Chandra and Franz [4], we implement labels with long integers. However, unlike Chandra and Franz, we interpret our labels as bit vectors with no inherent ordering, while they consider labels to form a totally ordered lattice. We take each bit to represent a different source; as long integers are 64 bits, we can track 64 sources. This representation of labels makes joining simple and fast, using a bitwise OR. Our labels are wrapped by our JSLabel (label) class simply for syntactic convenience.

Each code block carries its source, generally a URL, as an attribute. Prior to evaluation in the interpreter the sources are read from the code blocks and pushed to our URLMap class. This static class is responsible for storing sources and assigning them labels. Unlike previous work [4, 5], our mechanism focusses on control flow, not policy, assigning labels incrementally, with no hierarchy.

JavaScript values, objects, and functions are represented in the interpreter by JSC’s JSValue (value) and JSCell (cell) classes. While all JavaScript items are represented as values in the interpreter, values used in multiple code blocks (i.e. global variables/functions) are preserved in cells and later reconstructed. We have labeled both values and cells allowing complete and accurate control flow between interpreters.

As previously mentioned, the URLMap stores the label associated with a code blocks source. The propagation of labels in the interpreter begins here; during construction of a new value or cell its label is set to that of its source. Note that the labels of values constructed from cells are joined with the cell’s label since this is a flow of information. In the interpreter labels are propagated explicitly during evaluation in the opcodes, as discussed in Section 4.1. That is, a result is labeled with the join of labels from the values which directly influenced that result. Finally, any result is labeled with the current execution context, retrieved from the context stack.

5.3 Execution Context

As discussed in Section 4.2, we call OP_BRANCH at each opcode that performs a branch, while OP_MERGE is called before each opcode is executed; we implemented these two functions as macros, and integrated OP_MERGE with the interpreter’s built-in NEXT_INSTRUCTION macro. These interact with our ProgramCounter class which implements a stack of (label, instruction, call stack) triples. To track the current call stack we use the current register pointer, which slides down and up in memory as calls are made and finished.

When a function call is made the interpreter moves its register pointer down to make room for a function’s arguments, stores the current instruction pointer as the return point, and moves its instruction pointer to point into the called code block. As discussed in Section 4.2.2, we execute function in a context that includes the function’s label. To push a label to the stack appropriately we call OP_BRANCH before passing control to the called function. Because the post dominator of the call opcode is always its immediate successor OP_MERGE will pop appropriately from the ProgramCounter when the function returns. We also ensure that the called code block has static analysis information generated before jumping into it.

While our handling of eval is very similar to that of function calls (see Section 4.3), it is handled differently by the interpreter. When a call to eval is made the interpreter passes the string argument to its parser and recursively calls its execute method on the resulting code block. Similar to a normal function call, we want eval’ed code to execute in the context of its string argument. However, unlike in the case of a normal call, the recursive call to the interpreter returns control to the same point in the interpreter logic. Thus full control context tracking for eval then becomes as simple as pushing the label of the string to the context stack, evaluating the string, and popping.

6 Performance

To establish the overhead induced by our instrumentation of JavaScriptCore we compared the runtimes of 10 tests from the SunSpider-0.9.1 JavaScript benchmarking suite. These 10 were chosen because they ran reliably without crashing the interpreter; the others reliably triggered errors in memory management.
Figure 5: Results for 1 run of 10 SunSpider tests
code that, ostensibly, was unaffected by our modifications. The benchmarks were run on a late 2008 MacBook with a 2.1 GHz Intel Core 2 Duo and 4 GB of RAM, using the jsc command line JavaScript interpreter and the unix time utility. The times shown are calculated as time user + time system to compensate for scheduler differences. Both the baseline and instrumented versions of the code had just-in-time compilation and computed-goto interpretation disabled. Figure 5 and Table 1 show the results of the testing; note that these figures only represent one run of each test.

7 Conclusion

This paper has presented a novel and complete approach to propagating information flow labels in JavaScript. Our approach handles explicit flow, including prototype chain lookups, as well as implicit flow through a hybrid static-dynamic system that tracks execution context both intra- and interprocedurally. Our context tracking system also handles eval, executing eval'ed code in the context of the source string’s label.

We implemented our system in WebKit’s JavaScript interpreter, demonstrating its practicality and real-world application to a production web browser. Standard benchmarks show that this implementation has a worst case overhead of almost 1120%, with the best case showing no slowdown.

7.1 Further Work

While our labeling system solves many issues in tracking information flow in JavaScript, it is only a starting point for an information-flow-tracking security system for JavaScript. A major consideration is the performance of our system; while our implementation serves as a proof of concept, it imposes severe overhead on execution times. While none of our code was written with performance in mind, there is much room for speedup in the static analysis code, which utilizes computationally expensive data structures for the sake of easy implementation. We could also significantly reduce the overhead from label propagation by utilizing the sparse labeling semantics from Austin and Flanagan [2].

We also have not addressed the issue of exceptions in information flow − our static analysis currently does not treat exceptions. The modification to control flow caused by exceptions, particularly unhandled exceptions, raises several issues with control context tracking that deserve individual attention, work that is beyond the scope of this project.

Finally, while we have introduced new approaches to tracking information flow in JavaScript, our work lacks any form of security policy or enforcement. Because our goal in this work was to increase the security of client-side JavaScript, we hope to see work towards a mechanism that makes use of complete information flow tracking.
<table>
<thead>
<tr>
<th>Test</th>
<th>Before (sec)</th>
<th>After (sec)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>3d-cube</td>
<td>0.095</td>
<td>0.092</td>
<td>-3.2%</td>
</tr>
<tr>
<td>access-fannkuch</td>
<td>1.404</td>
<td>15.585</td>
<td>1010.0%</td>
</tr>
<tr>
<td>bitops-3bit-bits-in-byte</td>
<td>0.558</td>
<td>5.337</td>
<td>856.5%</td>
</tr>
<tr>
<td>bitops-bits-in-byte</td>
<td>0.573</td>
<td>6.296</td>
<td>998.8%</td>
</tr>
<tr>
<td>bitops-bitwise-and</td>
<td>0.865</td>
<td>5.252</td>
<td>507.2%</td>
</tr>
<tr>
<td>bitops-nsieve-bits</td>
<td>0.79</td>
<td>9.628</td>
<td>1118.7%</td>
</tr>
<tr>
<td>controlflow-recursive</td>
<td>0.585</td>
<td>3.484</td>
<td>495.6%</td>
</tr>
<tr>
<td>math-partial-sums</td>
<td>1.012</td>
<td>5.045</td>
<td>765.4%</td>
</tr>
<tr>
<td>math-spectral-norm</td>
<td>0.583</td>
<td>5.045</td>
<td>765.4%</td>
</tr>
<tr>
<td>regexp-dna</td>
<td>0.847</td>
<td>0.845</td>
<td>-0.2%</td>
</tr>
</tbody>
</table>

Table 1: Results for 1 run of 10 SunSpider tests

References


