1-1-2011

Information Flow Analysis for JavaScript

Seth Just

Alan Cleary

Brandon Shirley

Christian Hammer

Information Flow Analysis for JavaScript

Seth Just [*]
Reed College
Portland, OR
seth.just@gmail.com

Alan Cleary [*]
Western State College of Colorado
alan.cleary@western.edu

Brandon Shirley
Utah State University
b.l.s@aggiemail.usu.edu

Christian Hammer
Utah State University
c.hammer@acm.org

Abstract

Modern Web 2.0 pages combine scripts from several sources into a single client-side JavaScript program with almost no isolation. In order to prevent attacks from an untrusted third-party script or cross-site scripting, tracking provenance of data is imperative. However, no browser offers this security mechanism. This work presents the first information flow control mechanism for full JavaScript. We track information flow dynamically as much as possible but rely on intra-procedural static analysis to capture implicit flow. Our analysis handles even the dreaded eval function soundly and incorporates flow based on JavaScript’s prototype inheritance. We implemented our analysis in a production JavaScript engine and report both qualitative as well as quantitative evaluation results.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program Analysis; K.6.5 [*Management of Computing and Information Systems*]: Security and Protection

General Terms Algorithms, Languages, Security

Keywords information flow control, implicit flow, hybrid program analysis, eval, unstructured control flow

1. Introduction

Client-side scripting languages such as JavaScript are ubiquitous in modern, internet-connected computing, but pose a serious security risk to the client. In particular, 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 puts the browser in jeopardy. Apart from that, injection attacks like cross site scripting are still — even after years of research — the top vulnerabilities on the web.[1] While a variety of security mechanisms do exist in current browser environments (e.g. the “same origin policy”, SSL certificates) these approaches often lack the flexibility necessary to cope with these real-world threats. They only protect the client from executing scripts, or isolate scripts in different frames. However, there is almost no isolation between scripts loaded into the same browser frame, and the user has no control over how confidential information is processed once it has been inserted into the system. Furthermore, traditional 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). *Information flow control* (IFC) is a technique to ensure that confidential information cannot illicitly leak to unauthorized output channels, and to enforce that untrusted data may not influence critical computation. To that end, it needs to track *explicit flow*, which arises due to computations being dependent on the values of their parameters, and *implicit flow*, which arises from predicates that control the execution of certain code blocks. For example, in Figure 1, the predicate i & 1 has explicit flow from the parameter i. Depending on the value of predicates i & 0, one of the return statements is executed, so both have implicit flow from this predicate.

IFC can be enforced in static, dynamic, or even hybrid ways. As JavaScript is an extremely dynamic language, static analyses needs to be very conservative or make assumptions that are not satisfied in practice. Dynamic IFC on the other side is precise with respect to explicit flow, and has been shown to be able to enforce security properties like noninterference [3]. The downside is that dynamic IFC suffers from a lack of theory on how to account for implicit flow when a language offers features like `break`, `continue`, `return`, and `try ... catch ... finally` blocks. Therefore several dynamic analyses for realistic languages have resorted to static analysis (e.g. [6]). However,

[*] This research was conducted at USU supported by NSF grant IIS-0850825: REU Site Program in Computer Vision and Multi-Agent Systems.

[1] https://www.owasp.org/index.php/Top_10_2010-Main

JavaScript's eval function, that takes a string argument and executes it as code, can dynamically alter control flow and thus renders even local static analysis challenging.

Besides, dynamic IFC has long been considered infeasible as information flow control is not a property, i.e. it cannot be characterized by analyzing a single execution in isolation [17], which dynamic analysis is based on. Ignoring alternative executions (e.g. the other branch of an if-block) may lead to *omission errors*, where implicit flow occurs because there was *no* information flow in the intercepted trace. However, recent research found techniques to conservatively reject programs that try to transmit information using these channels. For example, Zdancewic [22] proposed a *no-sensitive-upgrade* check, which only allows assignment to variables that have a security level at least as high as the level of the predicate that governs the assignment. This prevents upgrading of variables in only one branch of execution. Austin and Flanagan [3] show that this check suffices to guarantee non-interference, which stipulates that no secret input of the program may influence public output.

Finally, the extremely dynamic nature of JavaScript imposes several subtle dependences that need to be considered in flow tracking. In particular, dynamic dispatch uses dynamic lookup to determine the closure to be invoked, and thus information flow control needs to account for the prototype chain of the receiver object.

This paper presents a novel approach to information flow tracking, using a hybrid of static and dynamic approaches to label values during the course of program execution. Our work dynamically tracks explicit information flow during execution, as well as utilizing static analysis to support dynamic context tracking for intra- and inter-procedural implicit flow tracking. We elaborate that eval does not interfere with the local static analysis that determines implicit flow and define how dynamic lookup needs to be accounted for in order to track explicit flow faithfully.

We implemented a prototype of this analysis in the production JavaScript engine WebKit, and demonstrate its ability to track information flow inter-procedurally and through the eval function.

In summary, this work makes the following contributions:

- We present a novel hybrid information flow analysis that can handle the full JavaScript language. Explicit flow and inter-procedural flow are tracked fully dynamically.
- We use intra-procedural static analysis to determine implicit flow even in the case of unstructured control flow.
- We handle explicit and implicit flow of the `eval` function that notoriously precludes static analysis or requires intolerable conservative approximations.
- We present a new technique to incorporate dependences induced by dynamic lookup and dispatch for prototype inheritance.

```
function leakBit(i) {
  r = 1;
  if (i & 1)
    return r;
  return 0;
}
s="";
for (i = 0; i < 32; i++)
  s += leakBit(secret = secret >> 1);
s;
```

Figure 1: Implicit flow due to unstructured control flow. This example demonstrates the limitation of ignoring control dependence due to unstructured control flow by tainting only variables in block scopes with the label of the corresponding predicate.

```
  while (p1) {
    if (p2)
      return;
    s1; p1 = x > 0;
  }
```

Figure 2: Implicit flow due to unstructured control flow. This example demonstrates the limitation of ignoring control dependence due to unstructured control flow by tainting only variables in block scopes with the label of the corresponding predicate.

- The proposed analysis has been implemented in a production JavaScript engine (sans exception handling), and we present initial results of flow tracking and runtime overhead.

2. Background

Recent work on information flow control for dynamic languages has gone two ways: Either it defined a system for a theoretic language and proved a security property like non-interference for this system, e.g. [3, 4, 16]; or it defines an analysis for a realistic language like JavaScript, but remains rather ad-hoc without security guarantees [18]. The goal of this work is to merge the two branches of research in order to achieve sound information flow tracking for the realistic programming language JavaScript. JavaScript is a prototype for many dynamic languages as it contains the most dynamic features available to similar languages. Dynamic information flow labeling allows for the tracking of information flow without requiring the source code to be altered or annotated. In order to preserve the correctness guarantees provided by Austin and Flanagan [3], we need to extend their labeling rules to language constructs that exceed their adapted λ-calculus. For example, their references do not model objects or arrays, which are predominant in JavaScript. Nor does it cover unstructured control flow and the resultant implicit flow. Fortunately, program analysis has been studying these features and presented theories on how to handle them. In particular, the correspondence between information flow control and program dependence has long been assumed [10], but only recently been formally proved [19]. This allows us to leverage results from program analysis to extend theoretical languages such that their security properties carry over to realistic languages. In particular, we know that explicit flow corresponds to data dependence in program analysis, which is between an expression that uses a value and the expression that computed it. And that implicit flow corresponds to control dependence, which arises between a predicate that governs whether a certain program path will be executed, and the statement that executes depending on this decision.

3. Information Flow Control for JavaScript

In the following we will elaborate on the analyses to track information flow. In order to have a base implementation that we can tune and compare to later on, we decided to implement universal labeling semantics [3] where every value in the JavaScript interpreter has a security label attached. At present, we encode information flow as a power set of all possible provenances, which we currently limit to 64. This allows for efficient computation of least upper bounds as the join becomes a bitwise OR.

3.1 Explicit Flow

Our system handles explicit flow by updating labels on modified values with the labels of those values that influence the final result. In particular, the result of an operation like addition is labeled with the join of both operand's labels. Altering the structure of an object or array by adding or removing properties results in an update of the object's label with the label of the current context, as these changes can be observed by an attacker.

3.2 Implicit Flow

Theoretical papers on information flow control usually restrict themselves to languages without unstructured control flow. However, realistic imperative languages predominately contain some variant of goto like break, continue, throw, return, or try ... catch ... finally. The implicit flow that arises from these language constructs must not be ignored, but that requires control dependence instead of analyzing code block structure only. In particular, we found that both Vogt et al. [18] and Dhawan and Ganapathy [8] ignore control dependence due to unstructured control flow by tainting only variables in block scopes with the label of the corresponding predicate. As an example, consider Figure 2, where predicates p1 and p2 determine whether statement s1 will be executed. As s1 is control dependent on both predicates, it must be tainted with the join of both labels, however, these approaches only create a security context around the enclosing control structure, which in this case for the return statement only comprises the if. As a consequence, statement s1 would only be in the security context of the while statement, missing the dependence on the second predicate.

In our work we define security contexts independent from blocks, from a predicate – or more generally an operation that has more than one successor in the control flow graph – to its immediate post-dominator [5, 6]. This is equivalent to dynamic control dependence [20]. Generating a post-dominator tree for each method requires intra-procedural static analysis. At runtime a stack of security contexts determines the label currently induced by implicit flow. As in the work of Sabelfeld and Russo [16] and Austin and Flanagan [3], we utilize this stack to track a history of security contexts. Whenever a value is created or modified at runtime its label is joined with the set of labels currently on the stack, thus propagating implicit flow. We never need access to individual labels on the stack. Therefore, when pushing label l to the stack we join l to the current label on the top and push that value as the new top. This guarantees that the top of the stack represents the join of all implicit flows, avoiding a linear-time traversal each time we consult the stack for the current context.

3.2.1 Intraprocedural Context Tracking

Prior to execution we store a table of post-dominators for each operation within a method's code block. This table is generated from the control flow graph using the fast dominator algorithm of Lengauer and Tarjan [12]. During execution we execute some bookkeeping code before each operation to maintain our context stack. At each operation that has more than one successor in the control flow graph, we push the label of its predicate onto the stack, together with the immediate post-dominator for this operation. As every operation can be an immediate post-dominator, we need to check whether we reached the operation specified on the top of the stack, in which case we can pop off the label induced by the corresponding control dependence region.

3.2.2 Interprocedural Context Tracking

Inter-procedural control dependence is well-known when no unstructured control flow is possible. In this case, on top of the branching operations and its respective immediate post-dominator the call stack needs to be taken into account. Fortunately, a branching operation always has its immediate post-dominator in the same method when only structured control flow is possible. If both operations are guaranteed to be in the same method, intra-procedural control dependence can be extended to the inter-procedural case: The stack elements need to be extended by the current stack frame pointer such that when the immediate post-dominator of a branching point is reached, we can assert that this is not another iteration of the same method, for example in a re-

```
begin before_op(vPC, callStack)
  if context.top() = (_, vPC, callStack)
   then
         context.pop()
  fi
  if isBranching(vPC)
   then if
            context.top() = (_, idom[vPC], callStack)
            then
                  context.top().first ⊔= label[vPC]
            else
                  context.push(label, idom[vPC], callStack)
          fi
  fi
end
```

Figure 3: Algorithm for inter-procedural context tracking. In this example context is the context stack, vPC the current instruction pointer, idom the post-dominator table, callStack the call stack pointer, _ a wildcard, and $\sqcup =$ the join-and-assign operator on labels.

cursive method. Xin and Zhang [20] show that this information suffices to correctly compute dynamic inter-procedural control dependence. Figure 3 shows pseudocode for our algorithm to compute the label induced by implicit flow for the inter-procedural case. Before each opcode, we execute the macro before_op, which takes the current instruction pointer vPC, and the call stack pointer callStack into account. First it determines if the current operation is the immediate post-dominator of the branching point that was last executed. If this is the case, then the top element of the context stack is removed. If the current instruction is a branching point, we need to push the label of the current operation (i.e. predicate) onto the stack, together with the immediate post-dominator of this operation, and the current call stack. As an optimization, we can just join the label with the label of the top stack element if it already contains the same immediate post-dominator and call stack. Note that within a single method this approach is equivalent to the intra-procedural case, as the callStack does not change within a single procedure. The object-oriented and functional nature of JavaScript necessitates an extension to fully propagate labels: Functions are first-class objects that have associated labels specifying by which principal they were installed under which calling context. Thus a function must execute in a context that contains its label. To that end, we push that label on the context stack before executing a function, and pop it after returning.

3.3 Exceptions

In practice, realistic programming languages offer ways for leaving a method abruptly. For example a throw statement in JavaScript leaves the current method if there is no handler specified, and resumes execution of a handler in a method further up the call stack, if available. This semantic invalidates the assumption that a method call is post-dominated by its succeeding operation, as it is in structured control flow. Again, results from program analysis can be leveraged to model unstructured constructs such that assumptions from structured control flow are not invalidated [1, 9]. In particular, we ensure that each branching operation has its immediate post-dominator in the same method. To that end, each operation that might throw an exception not protected by an exception handler receives an edge to a synthetic EXIT node in the control flow graph whose predecessors comprise all return statements. Then, a synthetic node is added at the corresponding call site where the predicate checks whether an exception had been thrown or not. The successors of this node are the operation succeeding the call in the original control flow graph, and either a corresponding exception handler or the EXIT node of the containing method.

3.4 String eval

We handle eval similarly to a function call, however, eval does not have a function object as parameter, but a string. As the provenance of the string determines whether it should be trusted, we apply the label of the string to the context stack before transferring control. Treating eval equivalent to a function call is only safe if we can assert that its execution cannot invalidate the static analysis results of the callee. Fortunately the language definition precludes eval from changing control flow of the enclosing code block other than by throwing an exception. But an exception needs to be accounted for in any case, as even parsing the string parameter could result in a syntax error. Other ways to change the control flow, like break, continue, return, or try ... catch ... finally cannot be injected into the block that is calling eval, as the specification states that these must be enclosed in a block "not crossing function boundaries." Thus eval does not change the intra-procedural post-dominator function, and as the correctness of our inter-procedural control dependence relies only on the fact that a branching operation always has its immediate post-dominator in the same method – which we guarantee even for code generated by eval – our analysis remains sound.

3.5 Prototypes

JavaScript is an object oriented languaged, but it has no classes or inheritance system. Instead it makes use of *prototypes*: Each object has a special pointer to a prototype object (__proto__), and any property 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 subtle ways. Not only can one change the property itself, but (if it is not defined in an object then) changing the prototype may also change the lookup result. 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,

```
begin getProperty(value, property)
   label = new Label
   while value do
          label $\sqcup$= value.label
          if value.lookup(property)
            then
                  return value.lookup(property)
                   labeled with label
          fi
          value = value.__proto__
   od
end
```

Figure 4: Algorithm for prototype chain lookup. This is the pseudocode for the algorithm used to appropriately label values through prototype chain lookups.

as shown in Figure 4. This is particularly essential for dynamic method dispatch.

4. Implementation

We instrumented WebKit's production-level 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 eight classes. The Interpreter class required the most modifications, with instrumentation in nearly all the opcodes and custom macros for branching and merging operations. Prior to our instrumentation jsc had approximately 199,000 lines total lines of code. Our modified version of jsc has about 205,000 lines; this includes whitespace.

4.1 Static Analysis

JavaScript code is fed into the command-line JavaScript interpreter jsc which then compiles it into a bytecode code block sent to the instrumented interpreter for evaluation. Our implementation statically analyzes code blocks before evaluation. The static analyzer is implemented as a StaticAnalyzer and a FlowGraph class. StaticAnalyzer determines the post-dominators for each opcode in a code block as discussed in Section 3.2.2 and stores them for access during interpretation. StaticAnalyzer objects are stored alongside code blocks to avoid repeated generation of post-dominator trees, and the analysis is performed immediately before a code block is executed. At present we do not model flow due to exceptions and exception handling. We will add this in the next version of our implementation.

4.2 Labeling

Similar to the labeling technique used by Chandra and Franz [6], we implement labels with long integers and interpret the labels as bit vectors representing elements of the power set of information sources, which forms a totally ordered lattice. We take each bit to represent a different source and thus currently can track 64 sources. This representation of labels makes joining fast, using a bitwise OR. Our labels are wrapped by the JSLabel class 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 labels to them. JavaScript values, objects, and functions are represented in the interpreter by the JSValue class that represent primitive values and JSCell classes for JavaScript objects. JSCells can be boxed into JSValues for a unified data model. We labeled both values and cells, allowing complete and accurate information flow tracking. This also required modification of the Register class in order to propagate labels. A primitive value loses its label when it is pushed into a register. Thus, the Register class had to be modified to track these labels when JSValues are moved in and out of a given register. As previously mentioned, the URLMap stores the label associated with a code block's 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 3.1. 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.

4.3 Execution Context

As discussed in Section 3.2, we call before_op of Figure 3 before each opcode is executed, the second part is only called before a branching opcode; we implemented these two functions as macros, and integrated them with the interpreter's built-in functionality to jump to the next opcode to be executed. 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 return. When a function call is made, the interpreter moves its register pointer down to make room for function 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 3.2.1, we execute a 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 3.4), it is handled differently by the interpreter. When a call to eval is made the interpreter

```
 1 >var passwords =
 2    { "Hans": "jlasdfj", "Emily": "ajsdfo" };
 3
 4    ending. PC has length 0 and head 0x0
 5  undefined (labeled 0x4)
 6 >var user = "Hans";
 7
 8    ending. PC has length 0 and head 0x0
 9  undefined (labeled 0x4)
10 >eval("passwords." + user + " == \"" + readline() + "\"");
11   abc"; passwords['Emily'] + "
12
13    calling eval + pushing. PC has length 0 and head 0x0
14    PC has length 1 and head 0x404
15    popping after eval. PC has length 1 and head 0x404
16    PC has length 0 and head 0x0
17  ajsdfo (labeled 0x406)
```

Figure 5: Information control flow analysis. Example of correct flow analysis of a script injection attack via string construction for execution by `eval`. Comments are shown in blue and have been added to illustrate the state of the PC during execution.

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 context tracking for eval consists of pushing the label of the string on the context stack, evaluating the string, and restoring the previous context stack, which is exemplified in section 5.1.

5. Evaluation

Our instrumentation of jsc was evaluated qualitatively to demonstrate its effectiveness and quantitatively to determine the overhead induced by our flow analysis. Section 5.1 elaborates on a script injection attack example that demonstrates the correctness of our instrumentation and Section 5.2 benchmarks our instrumented jsc against the jsc that ships with WebKit using the SunSpider-0.9.1 JavaScript benchmarking suite.

5.1 Qualitative Experiment

The following example illustrates the ability of our flow analysis instrumentation to correctly track flow in a script injection attack. The attack demonstrated here is similar in nature to a XSS attack, a significant issue as mentioned in Section 1. This attack constructs a string from trusted and untrusted sources that will be executed by `eval`.

In order to demonstrate this attack, `jsc` has been modified to treat `readline` as an untrusted source giving it a unique label of `0x400`. All other labels are automatically assigned when a `JSValue` is created. The initial value is the context the label is created in, this is the URL the value comes from or the current PC as explained in Section 4.2. The interpreter is denoted by a label of `0x4`. If the source of the label is `NULL`, then a label of `0x2` is assigned. This can happen during prototype-chain traversal. This means that any label assigned from a non-`NULL` source should have a value greater than `0x6`. Labels less than or equal to `0x6`, i.e. `0x2` and `0x4`, or their combination, can be treated as trusted.

The entire attack is outlined in Figure 5. On line 1 an associative array, named `passwords`, is constructed using a trusted source. This array is assigned a label of `0x4`. The associative array's label will be joined with the head label, i.e. `0x0`. On line 6 a string is created from our trusted source, it is assigned the value of "Hans". This string is treated the same as the associative array as far as labeling is concerned.

The the attack starts on line 10. The `eval` statement is being fed a string that is constructed from two sources, the trusted source and the `readline` or untrusted source. `eval` evaluates the string as JavaScript. The expression is designed to validate a user's password. Using a carefully constructed string we are able to obtain Emily's password instead. Our instrumentation correctly pushes the value of `0x404` onto the stack as the resultant string is composed of both sources.

After `eval` terminates, the label at the top of the stack is popped off. The result of the expression has been correctly labeled `0x406`. One might have expected a label of `0x404`, but this actually comes out as `0x406` as explained above. The `0x2` and `0x4` that comprise the `0x6` portion of the label are treated as trusted.

The label denotes that the string's value has been influenced by the sources that correspond with `0x6` and `0x400`. This flow analysis could be fed to a policy mechanism that would decide how to handle the propagation of trusted or secure data. This is a straightforward example but it demonstrates some of our implementation's capabilities.

5.2 Quantitative Experiment

In order to establish the overhead induced by our instrumentation of `jsc` we compared the run times of ten of the tests from the SunSpider-0.9.1 JavaScript benchmarking suite. We selected this suite as it is standard, independent of the DOM in the browser, and covers a large spectrum of language features. The ten tests were chosen in order to represent the different SunSpider test groups, e.g., 3d, access, and math. The benchmarks were run on a late 2008 MacBook Pro with a 2.4 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 `user + sys` in order to get the actual CPU time spent on execution. Both the baseline and instrumented versions of the code had just-in-time compilation and computed-goto interpretation disabled. Figure 6 shows the results of the tests; note that these figures represent the average of ten runs of each test. The average overhead of our flow analysis was about 150%, meaning that our implementation took 2 to 3 times longer to execute each test. A small portion of the our implementation's execution time was spent performing static

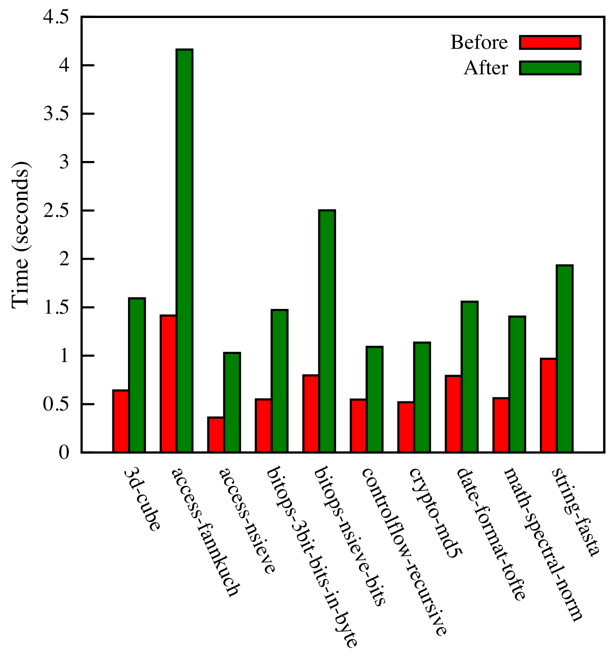

Figure 6: SunSpider Benchmark Results. These results show the average of 10 runs for each of the listed SunSpider tests.

analysis, as mentioned in Sections 4.1 and 3.2.2; on average about 0.25% of our execution time was spent in static analysis.

6. Related Work

Recent academic attention focusses on increasing the security of JavaScript, either by reducing its functionality (for example Facebook JavaScript or Google's Caja) or by augmenting the execution environment (e.g. [14].) Other approaches rely on code rewriting (e.g. [21]) 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. None of these approaches takes information flow in to account.

Schneider [17] defines a class of security policies that base their decisions only on 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. Unfortunately, information flow control is not enforceable by such a mechanism.

Li et al. [13] propose an information flow mechanism to control the flow between mutually untrusting code of mashups. They statically determine which paths could leak private data to untrusted output and "fingerprint" these by their call sequences. All other paths are automatically deemed benign. The overhead for the static analysis is high, however, subsequent executions of the same code have very low overhead. While pragmatic, this approach lacks an argument which security policy it guarantees. In particular, its disregard of implicit flow allows subversion.

Austin and Flanagan [3] presents two semantics for dynamic information flow label propagation for a version of the lambda calculus with references, one that assigns labels explicitly to values and a spare labeling that leaves them implicit where possible. Our work closely mirrors the former, but introduces mechanisms to handle eval and other features of a real-world language. Preliminary experiments suggest an overhead reduction of 50% for sparse labeling. Their later work [4] replaces the no-sensitive-upgrade rule with a permissive-upgrade rule that allows for the management of partially-leaked information.

Chandra and Franz [6] use a hybrid of static and dynamic analysis to instrument Java bytecode with labels, respecting both explicit and implicit flows. Their static analysis annotates code with information regarding the action of both sides of a code branch, instead of the conservative no-sensitive-upgrade rule.

Chugh et al. [7] separate programs into statically verifiable components and parts that must be checked at run-time. Their staged information flow verifier achieves a false positive rate of 33%, but it makes unrealistic assumptions for their static analysis that allow subverting the system.

Askarov and Sabelfeld [2] 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.

Russo and Sabelfeld [15] compare static and dynamic flow-sensitive information flow control systems and show that the fully-dynamic approach (in particular [3]) is not strictly more permissive than flow-sensitive static analysis [11].

Xin and Zhang [20] develop a mechanism for dynamic control dependence detection based on immediate post-dominator information. While their work does not focus on information flow, we base our mechanism to handle implicit flow on their definitions, and adapted their algorithms to information flow labeling.

7. Conclusion

This paper 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 the execution context both intra- and inter-procedurally. Our context tracking system also handles eval, executing eval'ed code in the context of the source string's label, and we showed that eval cannot invalidate the results of our static analysis.

We implemented our system in WebKit's JavaScript interpreter, demonstrating its practicality and real-world application to a production web browser. The experiments demonstrate that this implementation has an average overhead of about 150%, ranging from 95% to 215%.

7.1 Future Work

While our labeling system solves many issues in tracking information flow in JavaScript, it is only a starting point for an information flow control system for JavaScript. A major consideration is the performance of our system; while our implementation serves as a proof of concept, none of our code was written with performance in mind, so there is much room for speedup. In particular, We assume a significant reduction in overhead could be achieved by a sparse labeling semantics similar to Austin and Flanagan [3], whose preliminary experiments suggest a 50% overhead reduction.

Like related work [6], we 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 implicit and unhandled exceptions, raises several issues with control context tracking that deserve individual attention. Unfortunately, like its cousin Java, JavaScript has a significant number of operations that might throw an exception under certain circumstances. Including these would make the control flow graph too dense, which results in too many spurious control dependences and thus legal programs would be rejected due to spurious implicit flow.

Finally, while we have introduced new approaches to track information flow in JavaScript, our work needs to be embedded into a security policy enforcement mechanism in a browser environment to increase the security of client-side JavaScript.

References

[1] Matthew Allen and Susan Horwitz. Slicing Java programs that throw and catch exceptions. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 44–54, New York, NY, USA, 2003. ACM. doi: 10.1145/777388.777394.

[2] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 43–59, Washington, DC, USA, 2009. IEEE Computer Society. doi: 10.1109/CSF.2009.22.

[3] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 113–124, New York, NY, USA, 2009. ACM. doi: 10.1145/1554339.1554353.

[4] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM. doi: 10.1145/1814217.1814220.

[5] C. Bernardeschi, N. De Francesco, G. Lettieri, and L. Martini. Checking secure information flow in Java bytecode by code transformation and standard bytecode verification. *Software: Practice and Experience*, 34(13):1225–1255, 2004. doi: 10.1002/spe.611.

[6] Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *23rd Annual Computer Security Applications Conference*, pages 463–475, Los Alamitos, CA, USA, December 2007. IEEE Computer Society. doi: 10.1109/ACSAC.2007.37.

[7] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 50–62, New York, NY, USA, 2009. ACM. doi: 10.1145/1542476.1542483.

[8] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Annual Computer Security Applications Conference, 2009. (ACSAC '09)*, pages 382 –391, December 2009. doi: 10.1109/ACSAC.2009.43.

[9] Christian Hammer. *Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), Fak. f. Informatik, July 2009.

[10] C. Samuel Hsieh, Elizabeth A. Unger, and Ramon A. Mata-Toledo. Using program dependence graphs for information flow control. *Journal of Systems and Software*, 17(3):227 – 232, 1992. doi: 10.1016/0164-1212(92)90111-V.

[11] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–90, New York, NY, USA, 2006. ACM Press. doi: 10.1145/1111037.1111045.

[12] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979. doi: 10.1145/357062.357071.

[13] Zhou Li, Kehuan Zhang, and XiaoFeng Wang. Mash-IF: Practical information-flow control within client-side mashups. *Dependable Systems and Networks, International Conference on*, 0:251–260, 2010. doi: 10.1109/DSN.2010.5544312.

[14] Leo A. Meyerovich and Benjamin Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *SP '10: Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 481–496, Los Alamitos, CA, USA, May 2010. IEEE Computer Society. doi: 10.1109/SP.2010.36.

[15] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 186 –199, july 2010. doi: 10.1109/CSF.2010.20.

[16] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *In Proceedings of Andrei Ershov*

International Conference on Perspectives of System Informatics, LNCS. Springer-Verlag, June 2009. doi: 10.1007/978-3-642-11486-1_30.

[17] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, February 2000. doi: 10.1145/353323.353382.

[18] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium*, 2007.

[19] Daniel Wasserrab and Denis Lohner. Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In *6th International Verification Workshop - VERIFY-2010*.

[20] Bin Xin and Xiangyu Zhang. Efficient online detection of dynamic control dependence. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 185–195, New York, NY, USA, 2007. ACM. doi: 10.1145/1273463.1273489.

[21] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–249, New York, NY, USA, 2007. ACM. doi: 10.1145/1190216.1190252.

[22] Stephan A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, August 2002.