Information Flow Control in WebKit's JavaScript Bytecode

Abhishek Bichhawat[1], Vineet Rajani[2], Deepak Garg[2], and Christian Hammer[1]

[1] Saarland University, Germany
[2] MPI-SWS, Germany

Abstract. Websites today routinely combine JavaScript from multiple sources, both trusted and untrusted. Hence, JavaScript security is of paramount importance. A specific interesting problem is information flow control (IFC) for JavaScript. In this paper, we develop, formalize and implement a dynamic IFC mechanism for the JavaScript engine of a production Web browser (specifically, Safari's WebKit engine). Our IFC mechanism works at the level of JavaScript *bytecode* and hence leverages years of industrial effort on optimizing both the source to bytecode compiler and the bytecode interpreter. We track both explicit and implicit flows and observe only moderate overhead. Working with bytecode results in new challenges including the extensive use of unstructured control flow in bytecode (which complicates lowering of program context taints), unstructured exceptions (which complicate the matter further) and the need to make IFC analysis permissive. We explain how we address these challenges, formally model the JavaScript bytecode semantics and our instrumentation, prove the standard property of termination-insensitive non-interference, and present experimental results on an optimized prototype.

Keywords: Dynamic information flow control, JavaScript bytecode, taint tracking, control flow graphs, immediate post-dominator analysis

1 Introduction

JavaScript (JS) is an indispensable part of the modern Web. More than 95% of all websites use JS for browser-side computation in Web applications [1]. Aggregator websites (e.g., news portals) integrate content from various mutually untrusted sources. Online mailboxes display context-sensitive advertisements. All these components are glued together with JS. The dynamic nature of JS permits easy inclusion of external libraries and third-party code, and encourages a variety of code injection attacks, which may lead to integrity violations. Confidentiality violations like information stealing are possible wherever third-party code is loaded directly into another web page [2]. Loading third-party code into separate iframes protects the main frame by the same-origin policy, but hinders interaction that mashup pages crucially rely on and does not guarantee absence of attacks [3]. *Information flow control* (IFC) is an elegant solution for such

problems. It ensures security even in the presence of untrusted and buggy code. IFC for JS differs from traditional IFC as JS is extremely dynamic [3,1], which makes sound static analysis difficult.

Therefore, research on IFC for JS has focused on dynamic techniques. These techniques may be grouped into four broad categories. First, one may build an IFC-enabled, custom interpreter for JS source [4,5]. This turns out to be extremely slow and requires additional code annotations to handle semi-structured control flow like exceptions, return-in-the-middle, break and continue. Second, we could use a black-box technique, wherein an off-the-shelf JS interpreter is wrapped in a monitor. This is nontrivial, but doable with only moderate overhead and has been implemented in secure multi-execution (SME)[6,7]. However, because SME is a black-box technique, it is not clear how it can be generalized beyond *non-interference* [8] to handle *declassification* [9,10]. Third, some variant of inline reference monitoring (IRM) might inline taint tracking with the client code. Existing security systems for JS with IRM require subsetting the language in order to prevent dynamic features that can invalidate the monitoring process. Finally, it is possible to instrument the runtime system of an existing JS engine, either an interpreter or a just-in-time compiler (JIT), to monitor the program on-the-fly. While this requires adapting the respective runtime, it incurs only moderate overhead because it retains other optimizations within the runtime and is resilient to subversion attacks.

In this work, we opt for the last approach. We instrument a production JS engine to track taints dynamically and enforce *termination-insensitive non-interference* [11]. Specifically, we instrument the bytecode interpreter in WebKit, the JS engine used in Safari and other open-source browsers. The major benefit of working in the bytecode interpreter as opposed to source is that we retain the benefits of these years of engineering efforts in optimizing the production interpreter and the source to bytecode compiler.

We describe the key challenges that arise in dynamic IFC for JS bytecode (as opposed to JS source), present our formal model of the bytecode, the WebKit JS interpreter and our instrumentation, present our correctness theorem, and list experimental results from a preliminary evaluation with an optimized prototype running in Safari. In doing so, our work significantly advances the state-of-the-art in IFC for JS. Our main contributions are:

– We formally model WebKit's bytecode syntax and semantics, our instrumentation for IFC analysis and prove non-interference. As far as we are aware, this is the first formal model of bytecode of an in-production JS engine. This is a nontrivial task because WebKit's bytecode language is large (147 bytecodes) and we built the model through a careful and thorough understanding of approximately 20,000 lines of actual interpreter code.[3]

[3] Unlike some prior work, we are not interested in modeling semantics of JS specified by the ECMAScript standard. Our goal is to remain faithful to the production bytecode interpreter. Our formalization is based on WebKit build #r122160, which was the last build when we started our work.

– Using ideas from prior work [12], we use on-the-fly intra-procedural static analysis of immediate post-dominators to restrict overtainting, even with bytecode's pervasive unstructured conditional jumps. We extend the prior work to deal with exceptions. Our technique covers all unstructured control flow in JS (including break and continue), without requiring additional code annotations of prior work [5] and improves permissiveness.
– To make IFC execution more permissive, we propose and implement a bytecode-specific variant of the *permissive-upgrade* check [13].
– We implement our complete IFC mechanism in WebKit and observe moderate overheads.

Limitations We list some limitations of our work to clarify its scope. Although our instrumentation covers all WebKit bytecodes, we have not yet instrumented or modeled native JS methods, including those that manipulate the Document Object Model (DOM). This is ongoing work, beyond the scope of this paper. Like some prior work [4], our sequential non-interference theorem covers only single invocations of the JS interpreter. In reality, JS is reactive. The interpreter is invoked every time an event (like a mouse click) with a handler occurs and these invocations share state through the DOM. We expect that generalizing to *reactive non-interference* [14] will not require any instrumentation beyond what we already plan to do for the DOM. Finally, we do not handle JIT-compilation as it is considerably more engineering effort. JIT can be handled by inlining our IFC mechanism through a bytecode transformation.

Due to lack of space, several proofs and details of the model have been omitted from this paper. They can be found in a technical appendix available from the authors' homepages.

2 Related Work

Three classes of research are closely related to our work: formalization of JS semantics, IFC for dynamic languages, and formal models of Web browsers. Maffeis et al. [15] present a formal semantics for the entire ECMA-262 specification, the foundation for JS 3.0. Guha et al. [16] present the semantics of a core language which models the essence of JS and argue that all of JS 3.0 can be translated to that core. S5 [17] extends [16] to include accessors and eval. Our work goes one step further and formalizes the core language of a production JS engine (WebKit), which is generated by the source-to-bytecode compiler included in WebKit. Recent work by Bodin et al. [18] presents a Coq formalization of ECMAScript Edition 5 along with an extracted executable interpreter for it. This is a formalization of the English ECMAScript specification whereas we formalize the JS bytecode implemented in a real Web browser.

Information flow control is an active area of security research. With the widespread use of JS, research in dynamic techniques for IFC has regained momentum. Nonetheless, static analyses are not completely futile. Guarnieri et al. [19] present a static abstract interpretation for tracking taints in JS. However, the omnipresent `eval` construct is not supported and this approach does

not take implicit flows into account. Chugh et al. propose a staged information flow approach for JS [20]. They perform server-side static policy checks on statically available code and generate residual policy-checks that must be applied to dynamically loaded code. This approach is limited to certain JS constructs excluding dynamic features like dynamic field access or the `with` construct.

Austin and Flanagan [21] propose purely dynamic IFC for dynamically-typed languages like JS. They use the no-sensitive-upgrade (NSU) check [22] to handle implicit flows. Their permissive-upgrade strategy [13] is more permissive than NSU but retains termination-insensitive non-interference. We build on the permissive-upgrade strategy. Just et al. [12] present dynamic IFC for JS bytecode with static analysis to determine implicit flows precisely even in the presence of semi-unstructured control flow like `break` and `continue`. Again, NSU is leveraged to prevent implicit flows. Our overall ideas for dealing with unstructured control flow are based on this work. In contrast to this paper, there was no formalization of the bytecodes, no proof of correctness, and implicit flow due to exceptions was ignored.

Hedin and Sabelfeld propose a dynamic IFC approach for a language which models the core features of JS [4], but they ignore JS's constructs for semi-structured control flow like `break` and `continue`. Their approach leverages a dynamic type system for JS source. To improve permissiveness, their subsequent work [23] uses testing. It detects security violations due to branches that have not been executed and injects annotations to prevent these in subsequent runs. A further extension introduces annotations to deal with semi-structured control flow [5]. Our approach relies on analyzing CFGs and does not require annotations.

Secure multi-execution (SME) [6] is another approach to enforcing non-interference at runtime. Conceptually, one executes the same code once for each security level (like low and high) with the following constraints: high inputs are replaced by default values for the low execution, and low outputs are permitted only in the low execution. This modification of the semantics forces even unsafe scripts to adhere to non-interference. FlowFox [7] demonstrates SME in the context of Web browsers. Executing a script multiple times can be prohibitive for a security lattice with multiple levels. Further, all writes to the DOM are considered publicly visible output, while tainting allows persisting a security label on DOM elements. It is also unclear how declassification may be integrated into SME. Austin and Flanagan [24] introduce a notion of faceted values to simulate multiple executions in one run. They keep n values for every variable corresponding to n security levels. All the values are used for computation as the program proceeds but the mechanism enforces non-interference by restricting the leak of high values to low observers.

Browsers work reactively; input is fed to an event queue that is processed over time. Input to one event can produce output that influences the input to a subsequent event. Bohannon et al. [14] present a formalization of a reactive system and compare several definitions of reactive non-interference. Bielova et al. [25] extend reactive non-interference to a browser model based on SME. This

is currently the only approach that supports reactive non-interference for JS. We will extend our work to the reactive setting as the next step.

Finally, Featherweight Firefox [26] presents a formal model of a browser based on a reactive model that resembles that of Bohannon et al. [14]. It instantiates the consumer and producer states in the model with actual browser objects like window, page, cookie store, mode, connection, etc. Our current work entirely focuses on the formalization of the JS engine and taint tracking to monitor information leaks. We believe these two approaches complement each other and plan to integrate such a model into our future holistic enforcement mechanism spanning JS, the DOM and other browser components.

3 Background

We provide a brief overview of basic concepts in dynamic enforcement of information flow control (IFC). In dynamic IFC, a language runtime is instrumented to carry a security label or taint with every value. The taint is an element of a pre-determined lattice and is an upper bound on the security levels of all entities that have influenced the computation that led to the value. For simplicity of exposition, we use throughout this paper a three-point lattice $\{L, H, \star\}$ (L = low or public, H = high or secret, $\star$ = partially leaked secret), with $L \sqsubseteq H \sqsubseteq \star$ [13]. For now, readers may ignore $\star$. Our instrumentation works over a more general powerset lattice, whose individual elements are Web domains. We write r^ℓ for a value r tagged with label ℓ.

Information flows can be categorized as *explicit* and *implicit* [27]. Explicit flows arise as a result of variables being assigned to others, or through primitive operations. For instance, the statement `x = y + z` causes an explicit flow from values in both `z` and `y` to `x`. Explicit flows are handled in the runtime by updating the label of the computed value (`x` in our example) with the least upper bound of the labels of the operands in the computation (`y`, `z` in our example).

Implicit flows arise from control dependencies. For example, in the program `l = 0; if (h) {l = 1;}`, there is an implicit flow from `h` to the final value of `l` (that value is `1` iff `h` is `1`). To handle implicit flows, dynamic IFC systems maintain the so-called *pc* label (program-context label), which is an upper bound on the labels of values that have influenced the control flow thus far. In our last example, if the value in `h` has label H, then *pc* will be H within the `if` branch. After `l = 1` is executed, the final value of `l` inherits not only the label of `1` (which is L), but also of the *pc*; hence, that label is also H. This alone does not prevent information leaks: When `h = 0`, `l` ends with $\texttt{0}^L$; when `h = 1`, `l` ends with $\texttt{1}^H$. Since $\texttt{0}^L$ and $\texttt{1}^H$ can be distinguished by a public attacker, this program leaks the value of `h` despite correct propagation of implicit taints. Formally, the instrumented semantics so far fail the standard property of *non-interference* [8].

This problem can be resolved through the well-known *no-sensitive-upgrade* (NSU) check [22,21], which prohibits assignment to a low-labeled variable when *pc* is high. This recovers non-interference if the adversary cannot observe program termination (*termination-insensitive non-interference*). In our example, when `h`

`= 0`, the program terminates with `l` $= 0^L$. When `h = 1`, the instruction `l = 1` gets stuck due to NSU. These two outcomes are deemed observationally equivalent for the low adversary, who cannot determine whether or not the program has terminated in the second case. Hence, the program is deemed secure.

Roughly, a program is termination-insensitive non-interferent if any two terminating runs of the program starting from low-equivalent heaps (i.e., heaps that look equivalent to the adversary) end in low-equivalent heaps. Like all sound dynamic IFC approaches, our instrumentation renders any JS program termination-insensitive non-interferent, at the cost of modifying semantics of programs that leak information.

4 Design, Challenges, Insights and Solutions

We implement dynamic IFC for JS in the widely used WebKit engine by instrumenting WebKit's bytecode interpreter. In WebKit, bytecode is generated by a source-code compiler. Our goal is to not modify the compiler, but we are forced to make slight changes to it to make it compliant with our instrumentation. The modification is explained in Section 6. Nonetheless, almost all our work is limited to the bytecode interpreter.

WebKit's bytecode interpreter is a rather standard stack machine, with several additional data structures for JS-specific features like scope chains, variable environments, prototype chains and function objects. Local variables are held in registers on the call stack. Our instrumentation adds a label to all data structures, including registers, object properties and scope chain pointers, adds code to propagate explicit and implicit taints and implements a more permissive variant of the NSU check. Our label is a word size bit-set (currently 64 bits); each bit in the bit-set represents taint from a distinct domain (like google.com). Join on labels is simply bitwise or.

Unlike the ECMAScript specification of JS semantics, the actual implementation does *not* treat scope chains or variable environments like ordinary objects. Consequently, we model and instrument taint propagation on all these data structures separately. Working at the low-level of the bytecode also leads to several interesting conceptual and implementation issues in taint propagation as well as interesting questions about the threat model, all of which we explain in this section. Some of the issues are quite general and apply beyond JS. For example, we combine our dynamic analysis with a bit of static analysis to handle unstructured control flow and exceptions.

Threat model and compiler assumptions We explain our high-level threat model. Following standard practice, our adversary may observe all low-labeled values in the heap (more generally, an adversary at level ℓ in a lattice can observe all heap values with labels $\leq \ell$). However, we do not allow the adversary to directly observe internal data structures like the call stack or scope chains. This is consistent with actual interfaces in a browser that third-party scripts can access. In our non-interference proofs we must also show low-equivalence of these

internal data structures across two runs to get the right induction invariants, but assuming that they are inaccessible to the adversary allows more permissive program execution, which we explain in Section 4.1.

The bytecode interpreter executes in a shared space with other browser components, so we assume that those components do not leak information over side channels, e.g., they do not copy heap data from secret to public locations. This also applies to the compiler, but we do not assume that the compiler is functionally correct. Trivial errors in the compiler, e.g., omitting a bytecode could result in a leaky program even when the source code has no information leaks. Because our IFC works on the compiler's output, such compiler errors are not a concern. Formally, we assume that the compiler is an unspecified deterministic function of the program to compile and of the call stack, but not of the heap. This assumption also matches how the compiler works within WebKit: It needs access to the call stack and scope chain to optimize generated bytecode. However, the compiler never needs access to the heap. We ignore information leaks due to other side channels like timing.

4.1 Challenges and Solutions

IFC for JS is known to be difficult due to JS's highly dynamic nature. Working with bytecode instead of source code makes IFC harder. Nonetheless, solutions to many JS-specific IFC concerns proposed in earlier work [4] also apply to our instrumentation, sometimes in slightly modified form. For example, in JS, every object has a fixed parent, called a prototype, which is looked up when a property does not exist in the child. This can lead to implicit flows: If an object is created in a high context (when the *pc* is high) and a field missing from it, but present in the prototype, is accessed later in a low context, then there is an implicit leak from the high *pc*. This problem is avoided in both source- and bytecode-level analysis in the same way: The "prototype" pointer from the child to the parent is labeled with the *pc* where the child is created, and the label of any value read from the parent after traversing the pointer is joined with this label. Other potential information flow problems whose solutions remain unchanged between source- and bytecode-level analysis include implicit leaks through function pointers and handling of `eval` [12,4].

Working with bytecode both leads to some interesting insights, which are, in some cases, even applicable to source code analysis and other languages, and poses new challenges. We discuss some of these challenges and insights.

Unstructured control flow and CFGs To avoid overtainting *pc* labels, an important goal in implicit flow tracking is to determine when the influence of a control construct has ended. For block-structured control flow limited to `if` and `while` commands, this is straightforward: The effect of a control construct ends with its lexical scope, e.g., in (`if (h) {l = 1;}; l = 2`), `h` influences the control flow at `l = 1` but not at `l = 2`. This leads to a straightforward *pc* upgrading and downgrading strategy: One maintains a *stack* of *pc* labels [22]; the effective *pc* is the top one. When entering a control flow construct like `if` or `while`, a new

pc label, equal to the join of labels of all values on which the construct's guard depends with the previous effective *pc*, is pushed. When exiting the construct, the label is popped.

Unfortunately, it is unclear how to extend this simple strategy to non-block-structured control flow constructs such as exceptions, `break`, `continue` and `return`-in-the-middle for functions, all of which occur in JS. For example, consider the program `l = 1; while(1) {... if (h) {break;}; l = 0; break;}` with `h` labeled H. This program leaks the value of `h` into `l`, but no assignment to `l` appears in a block-scope guarded by `h`. Indeed, the *pc* upgrading and downgrading strategy just described is ineffective for this program. Prior work on source code IFC either omits some of these constructs [4,28], or introduces additional classes of labels to address these problems — a label for exceptions [4], a label for each loop containing `break` or `continue` and a label for each function [5]. These labels are more restrictive than needed, e.g., the code indicated by dots in the example above is executed irrespective of the condition `h` in the first iteration, and thus there is no need to raise the *pc* before checking that condition. Further, these labels are programmer annotations, which we cannot support as we do not wish to modify the compiler.

Importantly, unstructured control flow is a *very serious* concern for us, because WebKit's bytecode has completely unstructured branches like jump-if-false. In fact, all control flow, except function calls, is unstructured in bytecode.

To solve this problem, we adopt a solution based on static analysis of generated bytecode [29,12]. We maintain a control flow graph (CFG) of known bytecodes and for each branch node, compute its immediate post-dominator (IPD). The IPD of a node is the first instruction that will definitely be executed, no matter which branch is taken. Our *pc* upgrading and downgrading strategy now extends to arbitrary control flow: When executing a branch node, we push a new *pc* label on the stack *along with* the node's IPD. When we actually reach the IPD, we pop the *pc* label. In [30,31], the authors prove that the IPD marks the end of the scope of an operation and hence the security context of the operation, so our strategy is sound. In our earlier example, the IPD of `if(h) ...` is the end of the `while` loop because of the first `break` statement, so when `h == 0`, the assignment `l = 1` fails due to the NSU check and the program is termination-insensitive non-interference secure.

JS requires dynamic code compilation. We are forced to extend the CFG and to compute IPDs whenever code for either a function or an `eval` is compiled. Fortunately, the IPD of a node in the CFG lies either in the same function as the node or some function earlier in the call-chain (the latter may happen due to exceptions), so extending the CFG does not affect computation of IPDs of earlier nodes. This also relies on the fact that code generated from `eval` cannot alter the CFG of earlier functions in the call stack [12]. In the actual implementation, we optimize the calculation of IPDs further by working only intra-procedurally, as explained below. At the end, our IPD-based solution works for all forms of unstructured control flow, including unstructured branches in the bytecode, and

semi-structured `break`, `continue`, `return`-in-the-middle and exceptions in the source code.

Exceptions and synthetic exit nodes Maintaining a CFG in the presence of exceptions is expensive. An exception-throwing node in a function that does not catch that exception should have an outgoing control flow edge to the next exception handler in the call-stack. This means that (a) the CFG is, in general, inter-procedural, and (b) edges going out of a function depend on its calling context, so IPDs of nodes in the function must be computed *every time the function is called*. Moreover, in the case of recursive functions, the nodes must be replicated for every call. This is rather expensive. Ideally, we would like to build the function's CFG once when *the function is compiled* and work intra-procedurally (as we would had there been no exceptions). We explain how we attain this goal in the sequel.[4]

In our design, every function that may throw an unhandled exception has a special, *synthetic exit node* (SEN), which is placed after the regular return node(s) of the function. Every exception-throwing node, whose exception will not be caught within the function, has an outgoing edge to the SEN, which is traversed when the exception is thrown. The semantics of SEN (described below) correctly transfer control to the appropriate exception handler. By doing this, we eliminate all cross-function edges and our CFGs become intra-procedural. The CFG of a function can be computed when the function is compiled and is never updated. (In our implementation, we build two variants of the CFG, depending on whether or not there is an exception handler in the call stack. This improves efficiency, as we explain later.)

Control flows to the SEN when the function returns normally or when an exception is thrown but not handled within the function. If no unhandled exception occurred within the function, then the SEN transfers control to the caller (we record whether or not an unhandled exception occurred). If an unhandled exception occurred, then the SEN triggers a special mechanism that searches the call stack backward for the first appropriate exception handler and transfers control to it. (In JS, exceptions are indistinguishable, so we need to find only the first exception handler.) Importantly, we pop the call-stack up to the frame that contains the first exception handler but do *not* pop the *pc*-stack, which ensures that all code up to the exception handler's IPD executes with the same *pc* as the SEN, which is indeed the semantics one would expect if we had a CFG with cross-function edges for exceptions. This prevents information leaks.

If a function does not handle a possible exception but there is an exception handler on the call stack, then all bytecodes that could potentially throw an exception have the SEN as one successor in the CFG. Any branching bytecode will thus need to push to the *pc*-stack according to the security label of its condition. However, we do *not* push a new *pc*-stack entry if the IPD of the current node is the same as the IPD on the top of the *pc*-stack (this is just

[4] This problem and our solution are not particular to JS; they apply to dynamic IFC analysis in all languages with exceptions and functions.

an optimization) or if the IPD of the current node is the SEN, as in this case the *real* IPD, which is outside of this method, is already on the *pc*-stack. These semantics emulate the effect of having cross-function exception edges.

For illustration, consider the following two functions `f` and `g`. The $\diamond$ at the end of `g` denotes its SEN. Note that there is an edge from `throw 9` to $\diamond$ because `throw 9` is not handled within `g`. $\square$ denotes the IPD of the handler `catch(e) { l = 1; }`.

```
function f() = {                        function g() = {
 l = 0;                                  if (h) {throw 9;}
 try { g(); } catch(e) { l = 1; }        return 7;
 □ return l;                            } ◇
}
```

It should be clear that in the absence of instrumentation, when `f` is invoked with $pc = L$, the two functions together leak the value of `h` (which is assumed to have label H) into the return value of `f`. We show how our SEN mechanism prevents this leak. When invoking `g()` we do not know if there will be an exception in this function. Depending on the outcome of this method call, we will either jump to the exception handler or continue at $\square$. Based on that branch, we push the current pc and IPD $(L, \square)$ on the *pc*-stack. When executing the condition `if (h)` we do *not* push again, but merely update the top element to $(H, \square)$. If `h == 0`, control reaches $\diamond$ without an exception but with $pc = H$ because the IPD of `if (h)` is $\diamond$. At this point, $\diamond$ returns control to `f`, thus $pc = H$, but at $\square$, *pc* is lowered to L, so `f` ends with the return value 0^L. If `h == 1`, control reaches $\diamond$ with an unhandled exception. At this point, following the semantics of SEN, we find the exception handler `catch(e) { l = 1; }` and invoke it with the same *pc* as the point of exception, i.e., H. Consequently, NSU prevents the assignment `l = 1`, which makes the program termination-insensitive non-interferent.

Because we do not wish to replicate the CFG of a function every time it is called recursively, we need a method to distinguish the same node corresponding to two different recursive calls on the *pc*-stack. For this, when pushing an IPD onto the *pc*-stack, we pair it with a pointer to the current call-frame. Since the call-frame pointer is unique for each recursive call, the CFG node paired with the call-frame identifies a unique merge point in the real control flow graph.

In practice, even the intra-procedural CFG is quite dense because many JS bytecodes can potentially throw exceptions and, hence, have edges to the SEN. To avoid overtainting, we perform a crucial common-case optimization: When there is no exception handler on the call stack we do not create the SEN and the corresponding edges from potentially exception-throwing bytecodes at all. This is safe as a potentially thrown exception can only terminate the program instantly, which satisfies termination-insensitive non-interference if we ensure that the exception message is not visible to the attacker. Whether or not an exception handler exists is easily tracked using a stack of Booleans that mirrors the call-stack; in our design we overlay this stack on the *pc*-stack by adding an extra Boolean field to each entry of the *pc*-stack. In summary, each entry of our *pc*-stack is a quadruple containing a security label, a node in the intraprocedural CFG, a call-frame pointer and a Boolean value. In combination with SENs, this

design allows us to work only with intraprocedural CFGs that are computed when a function is compiled. This improves efficiency.

Permissive-upgrade check, with changes The standard NSU check halts program execution whenever an attempt is made to assign a variable with a low-labeled value in a high *pc*. In our earlier example, `l = 0; if (h) {l = 1;}`, assuming that `h` stores a H-labeled value, program execution is halted at the command `l = 1`. As Austin and Flanagan (AF in the sequel) observe [13], this may be overly restrictive when `l` will not, in fact, have observable effects (e.g., `l` may be overwritten by a constant immediately after `if (h) {l = 1;}`). So, they propose propagating a special taint called $\star$ into `l` at the instruction `l = 1` and halting a program when it tries to *use* a value labeled $\star$ in a way that will be observable (AF call this special taint P for "partially leaked"). This idea, called the *permissive-upgrade* check, allows more program execution than NSU would, so we adopt it. In fact, this additional permissiveness is absolutely essential for us because the WebKit compiler often generates dead assignments within branches, so execution would pointlessly halt if standard NSU were used.

We differ from AF in *what* constitutes a use of a value labeled $\star$. As expected, AF treat occurrence of $\star$ in the guard of a branch as a use. Thus, the program `l = 0; if (h) {l = 1;}; if (l) {l' = 2}` is halted at the command `if (l)` when `h == 1` because `l` obtains taint $\star$ at the assignment `l = 1` (if the program is not halted, it leaks `h` through `l'`). However, they allow $\star$-tainted values to flow into the heap. Consider the program `l = 0; if (h) {l = 1;}; obj.a = l`. This program is insecure in our model: The heap location `obj.a`, which is accessible to the adversary, ends with 0^L when `h == 0` and with $1^\star$ when `h == 1`. AF deem the program secure by assuming that any value with label $\star$ is low-equivalent to any other value (in particular, 0^L and $1^\star$ are low-equivalent). However, this definition of low-equivalence for dynamic analysis is virtually impossible to enforce if the adversary has access to the heap outside the language: After writing 0^L to `obj.a` (for `h == 0`), a dynamic analysis cannot determine that the alternate execution of the program (for `h == 1`) *would have* written a $\star$-labeled value and, hence, cannot prevent the adversary from seeing 0^L.

Consequently, in our design, we use a modified permissive-upgrade check, which we call the *deferred NSU check*, wherein a program is halted at any construct that may potentially flow a $\star$-labeled value into the heap. This includes all branches whose guard contains a $\star$-labeled value and any assignments whose target is a heap location and whose source is $\star$-labeled. However, we do not constrain flow of $\star$-labeled values in data structures that are invisible to the adversary in our model, e.g., local registers and variable environments. This design critically relies on treating internal data structures differently from ordinary JS objects, which is not the case, for instance, in the ECMAScript specification.

5 Formal Model and IFC

We formally model WebKit's JS bytecode and the semantics of its bytecode interpreter with our instrumentation of dynamic IFC. We prove termination-

```
ins  := prim-ins  | obj-ins
      | func-ins  | scope-ins  | exc-ins
prim-ins  := prim dst:r src1:r src2:r
           | mov dst:r src:r
           | jfalse cond:r target:offset
           | loop-if-less src1:r src2:r target:offset
           | typeof dst:r src:r
           | instanceof dst:r value:r cProt:r
func-ins  := enter
           | ret result:r
           | end result:r
           | call func:r args:n
           | call-put-result res:r
           | call-eval func:r args:n
           | create-arguments dst:r
           | new-func dst:r func:f
           | create-activation dst:r
           | construct func:r args:n
           | create-this dst:r
obj-ins  := new-object dst:r
          | get-by-id dst:r base:r prop:id
          | put-by-id base:r prop:id value:r direct:b
          | del-by-id dst:r base:r prop:id
          | get-pnames dst:r base:r i:n size:n breaktarget:offset
          | next-pname dst:r base:r i:n size:n iter:n target:offset
          | put-getter-setter base:r prop:id getter:r setter:r
scope-ins  := resolve dst:r prop:id
            | resolve-skip dst:r prop:id skip:n
            | resolve-global dst:r prop:id
            | resolve-base dst:r prop:id isStrict:bool
            | resolve-with-base bDst:r pDst:r prop:id
            | get-scoped-var dst:r index:n skip:n
            | put-scoped-var index:n skip:n value:r
            | push-scope scope:r
            | pop-scope
            | jmp-scope count:n target:offset
exc-ins  := throw ex:r
          | catch ex:r
```

Fig. 1. Instructions

insensitive non-interference for programs executed through our instrumented interpreter. We do not model the construction of the CFG or computation of IPDs; these are standard. To keep presentation accessible, we present our formal model at a somewhat high-level of abstraction. Details are resolved in our technical appendix.

5.1 Bytecode and Data Structures

The version of WebKit we model uses a total of 147 bytecodes or instructions, of which we model 69. The remaining 78 bytecodes are redundant from the perspective of formal modeling because they are *specializations* or wrappers on other bytecodes to improve efficiency. The syntax of the 69 bytecodes we model is shown in Fig. 1. The bytecode `prim` abstractly represents 34 primitive binary and unary (with just the first two arguments) operations, all of which behave similarly. For convenience, we divide the bytecodes into primitive instructions (prim-ins), instructions related to objects and prototype chains (obj-ins), instructions related to functions (func-ins), instructions related to scope chains (scope-ins) and instructions related to exceptions (exc-ins). A bytecode has the form $\langle inst\_name \;\; list\_of\_args \rangle$. The arguments to the instruction are of the form ⟨var⟩:⟨type⟩, where *var* is the variable name and *type* is one of the following: r, n, bool, id, prop and offset for register, constant integer, constant Boolean, identifier, property name and jump offset value, respectively.

In WebKit, bytecode is organized into code blocks. Each code block is a sequence of bytecodes with line numbers and corresponds to the instructions for a function or an `eval` statement. A code block is generated when a function is created or an `eval` is executed. In our instrumentation, we perform control flow analysis on a code block when it is created and in our formal model we abstractly represent a code block as a CFG, written ζ. Formally, a CFG is a directed graph, whose nodes are bytecodes and whose edges represent possible control flows. There are no cross-function edges. A CFG also records the IPD of each node. IPDs are computed using an algorithm by Lengauer and Tarjan [32]

when the CFG is created. If the CFG contains uncaught exceptions, we also create a SEN. For a CFG ζ and a non-branching node $\iota \in \zeta$, $Succ(\zeta, \iota)$ denotes ι's unique successor. For a conditional branching node ι, $Left(\zeta, \iota)$ and $Right(\zeta, \iota)$ denote successors when the condition is true and false, respectively.

The bytecode interpreter is a standard stack machine, with support for JS features like scope chains and prototype chains. The state of the machine (with our instrumentation) is a quadruple $\langle \iota, \theta, \sigma, \rho \rangle$, where ι represents the current node that is being executed, θ represents the heap, σ represents the call-stack and ρ is the *pc*-stack.

We assume an abstract, countable set $\mathcal{A} = \{a, b, \ldots\}$ of heap locations, which are references to objects. The heap θ is a partial map from locations to objects. An object O may be:

- An ordinary JS object $N = (\{p_i \mapsto v_i\}_{i=0}^{n}, \_\_proto\_\_ \mapsto a^{\ell_p}, \ell_s)$, containing properties named $p_0, \ldots, p_n$ that map to labeled values $v_0, \ldots, v_n$, a prototype field that points to a parent at heap location a, and two labels ℓ_p and ℓ_s. ℓ_p records the *pc* where the object was created. ℓ_s is the so-called structure label, which is an upper bound on all *pc*s that have influenced which fields exist in the object.[5]
- A function object $F = (N, \zeta, \Sigma)$, where N is an ordinary object, ζ is a CFG, which corresponds to the the function stored in the object, and Σ is the scope chain (closing context) of the function.

A labeled value $v = r^{\ell}$ is a value r paired with a security label ℓ. A value r in our model may be a heap location a or a JS primitive value n, which includes integers, Booleans, regular expressions, arrays, strings and the special JS values `undefined` and `null`.

The call-stack σ contains one call-frame for each incomplete function call. A call-frame μ contains an array of registers for local variables, a CFG ζ for the function represented by the call-frame, the return address (a node in the CFG of the previous frame), and a pointer to a scope-chain that allows access to variables in outer scopes. Additionally, each call-frame has an exception table which maps each potentially exception-throwing bytecode in the function to the exception handler within the function that surrounds the bytecode; when no such exception handler exists, it points to the SEN of the function (we conservatively assume that any unknown code may throw an exception, so bytecodes `call` and `eval` are exception-throwing for this purpose). $|\sigma|$ denotes the size of the call-stack and $!\sigma$ its top frame. Each register contains a labeled value.

A scope chain, Σ, is a sequence of scope chain nodes (SCNs), denoted S, paired with labels. In WebKit, a scope chain node S may either be an object or a variable environment V, which is an array of labeled values. Thus, $\Sigma ::= (S_1, \ell_1) : \ldots : (S_n, \ell_n)$ and $S ::= O \mid V$ and $V ::= v_1 : \ldots : v_n$.

[5] The *__proto__* field is the parent of the object; it is not the same as the prototype field of a function object, which is an ordinary property. Also, in our actual model, fields p_i map to more general property descriptors that also contain attributes along with labeled values. We elide attributes here to keep the presentation simple.

Each entry of the *pc*-stack ρ is a triple (ℓ, ι, p), where ℓ is a security label, ι is a node in a CFG, and p is a pointer to some call-frame on the call stack σ. (For simplicity, we ignore a fourth Boolean field described in Section 4.1 in this presentation.) When we enter a new control context, we push the new *pc* ℓ together with the IPD ι of the entry point of the control context and a pointer p to current call-frame. The pair (ι, p) uniquely identifies where the control of the context ends; p is necessary to distinguish the same branch point in different recursive calls of the function [12]. In our semantics, we use the meta-function *isIPD* to pop the stack. It takes the current instruction, the current *pc*-stack and the call stack σ, and returns a new *pc*-stack.

$$isIPD(\iota, \rho, \sigma) := \begin{cases} \rho.pop() & \text{if } !\rho = (\_, \iota, !\sigma) \\ \rho & \text{otherwise} \end{cases}$$

As explained in Section 4.1, as an optimization, we push a new node (ℓ, ι, σ) onto ρ only when (ι, σ) (the IPD) differs from the corresponding pair on the top of the stack and, to handle exceptions correctly, we also require that ι not be the SEN. Otherwise, we just join ℓ with the label on the top of the stack. This is formalized in the function $\rho.push(\ell, \iota, \sigma)$, whose obvious definition we elide.

If x is a pair of any syntactic entity and a security label, we write $\Upsilon(x)$ for the entity and $\Gamma(x)$ for the label. In particular, for $v = r^{\ell}$, $\Upsilon(v) = r$ and $\Gamma(v) = \ell$.

5.2 Semantics and IFC with Intra-procedural CFGs

We now present the semantics, which faithfully models our implementation using intra-procedural CFGs with SENs. The semantics is defined as a set of state transition rules that define the judgment: $\langle \iota, \theta, \sigma, \rho \rangle \rightsquigarrow \langle \iota', \theta', \sigma', \rho' \rangle$. Fig. 2 shows rules for selected bytecodes. For reasons of space we omit rules for other bytecodes and formal descriptions of some meta-function like *opCall* that are used in the rules. $C \Rightarrow A \diamond B$ is shorthand for a meta-level (if (C) then A else B).

`prim` reads the values from two registers `src1` and `src2`, performs a binary operation generically denoted by $\oplus$ on the values and writes the result into the register `dst`. `dst` is assigned the join of the labels in `src1`, `src2` and the head of the *pc*-stack ($!\rho$). To implement deferred NSU (Section 4.1), the existing label in `dst` is compared with the current *pc*. If the label is lower than the *pc*, then the label of `dst` is joined with $\star$. Note that the premise $\rho' = isIPD(\iota', \rho, \sigma)$ pops an entry from the *pc*-stack if its IPD matches the new program node ι'. This premise occurs in all semantic rules.

`jfalse` is a conditional jump. It skips `offset` number of successive nodes in the CFG if the register `cond` contains `false`, else it falls-through to the next node. Formally, the node it branches to is either $Right(\zeta, \iota)$ or $Left(\zeta, \iota)$, where ζ is the CFG in $!\sigma$. In accordance with deferred NSU, the operation is performed only if `cond` is not labeled $\star$. `jfalse` also starts a new control context, so a new node is pushed on the top of the *pc*-stack with a label that is the join of $\Gamma(\texttt{cond})$ and the current label on the top of the stack (unless the IPD of the branch point is already on top of the stack or it is the SEN, in which case we join the new

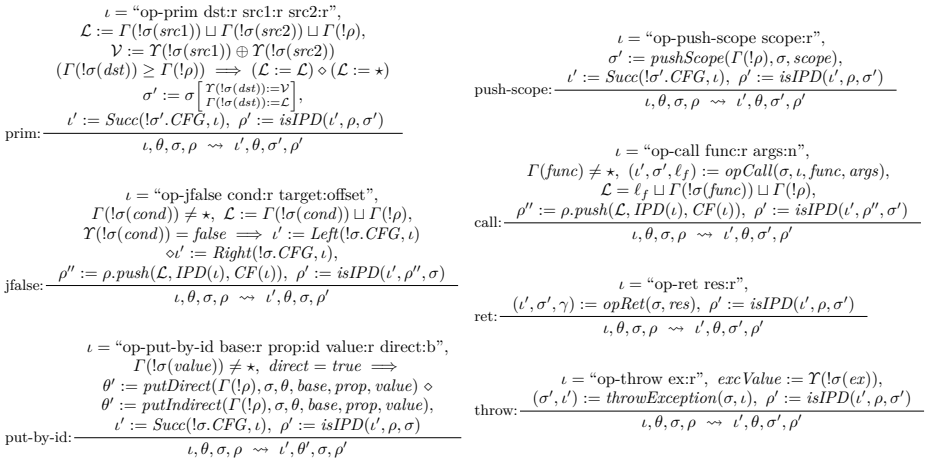

Fig. 2. Semantics, selected rules

label with the previous). Traversed from bottom to top, the *pc*-stack always has monotonically non-decreasing labels.

`put-by-id` updates the property `prop` in the object pointed to by register `base`. As explained in Section 4.1, we allow this only if the value to be written is not labeled $\star$. The flag `direct` states whether or not to traverse the prototype chain in finding the property; it is set by the compiler as an optimization. If the flag is `true`, then the chain is not traversed (meta-function *putDirect* handles this case). If `direct` is `false`, then the chain is traversed (meta-function *putIndirect*). Importantly, when the chain is traversed, the resulting value is labeled with the join of prototype labels ℓ_p and structure labels ℓ_s of all traversed objects. This is standard and necessary to prevent implicit leaks through the *___proto___* pointers and structure changes to objects.

`push-scope`, which corresponds to the start of the JS construct `with(obj)`, pushes the object pointed to by the register `scope` into the scope chain. Because pushing an object into the scope chain can implicitly leak information from the program context later, we also label all nodes in the scope-chain with the *pc*'s at which they were added to the chain. Further, deferred NSU applies to the scope chain pointer in the call-frame as it does to all other registers.

`call` invokes a function of the target object stored in the register `func`. Due to deferred NSU, the call proceeds only if $\Gamma($`func`$)$ is not $\star$. The call creates a new call-frame and initializes arguments, the scope chain pointer (initialized with the function object's Σ field), CFG and the return node in the new frame. The CFG in the call-frame is copied from the function object pointed to by `func`. All this is formalized in the meta-function *opCall*, whose details we omit here. Call is a branch instruction and it pushes a new label on the *pc*-stack which is the join of the current *pc*, $\Gamma($`func`$)$ and the structure label ℓ_f of the function object (unless the IPD of the current node is the SEN or already on the top of the *pc*-stack, in which case we join the new *pc*-label with the previous). `call` also initializes the

new registers' labels to the new *pc*. A separate bytecode, not shown here and executed first in the called function, sets register values to `undefined`. `eval` is similar to call but the code to be executed is also compiled.

`ret` exits a function. It returns control to the caller, as formalized in the meta-function *opRet*. The return value is written to an interpreter variable (γ).

`throw` throws an exception, passing the value in register `ex` as argument to the exception handler. Our *pc*-stack push semantics ensure that the exception handler, if any, is present in the call-frame pointed to by the *top* of the *pc*-stack. The meta-function *throwException* pops the call-stack up to this call-frame and transfers control to the exception handler, by looking it up in the exception table of the call-frame. The exception value in the register `ex` is transferred to the handler through an interpreter variable.

Correctness of IFC We prove that our IFC analysis guarantees termination-insensitive non-interference [11]. Intuitively, this means that if a program is run twice from two states that are observationally equivalent for the adversary and both executions terminate, then the two final states are also equivalent for the adversary. To state the theorem formally, we formalize equivalence for various data structures in our model. The only nonstandard data structure we use is the CFG, but graph equality suffices for it. A well-known complication is that low heap locations allocated in the two runs need not be identical. We adopt the standard solution of parametrizing our definitions of equivalence with a partial bijection β between heap locations. The idea is that two heap locations are related in the partial bijection if they were created by corresponding allocations in the two runs. We then define a rather standard relation $\langle \iota_1, \theta_1, \sigma_1, \rho_1 \rangle \sim_\ell^\beta \langle \iota_2, \theta_2, \sigma_2, \rho_2 \rangle$, which means that the states on the left and right are equivalent to an observer at level ℓ, up to the bijection β on heap locations. We defer details to the appendix.

Theorem 1 (Termination-insensitive non-interference) *Suppose:* *(1)* $\langle \iota_1, \theta_1, \sigma_1, \rho_1 \rangle \sim_\ell^\beta \langle \iota_2, \theta_2, \sigma_2, \rho_2 \rangle$, *(2)* $\langle \iota_1, \theta_1, \sigma_1, \rho_1 \rangle \rightsquigarrow^* \langle \texttt{end}, \theta_1', [], [] \rangle$, *and* *(3)* $\langle \iota_2, \theta_2, \sigma_2, \rho_2 \rangle \rightsquigarrow^* \langle \texttt{end}, \theta_2', [], [] \rangle$. *Then,* $\exists \beta' \supseteq \beta$ *such that* $\theta_1' \sim_\ell^{\beta'} \theta_2'$.

6 Implementation

We instrumented WebKit's JS engine (JavaScriptCore) to implement the IFC semantics of the previous section. Before a function starts executing, we generate its CFG and calculate IPDs of its nodes by static analysis of its bytecode. We modify the source-to-bytecode compiler to emit a slightly different, but functionally equivalent bytecode sequence for `finally` blocks; this is needed for accurate computation of IPDs. For evaluation purposes, we label each source script with the script's domain of origin; each seen domain is dynamically allocated a bit in our bit-set label. In general, our instrumentation terminates a script that violates IFC. However, for the purpose of evaluating overhead of our instrumentation, we ignore IFC violations in all experiments described here.

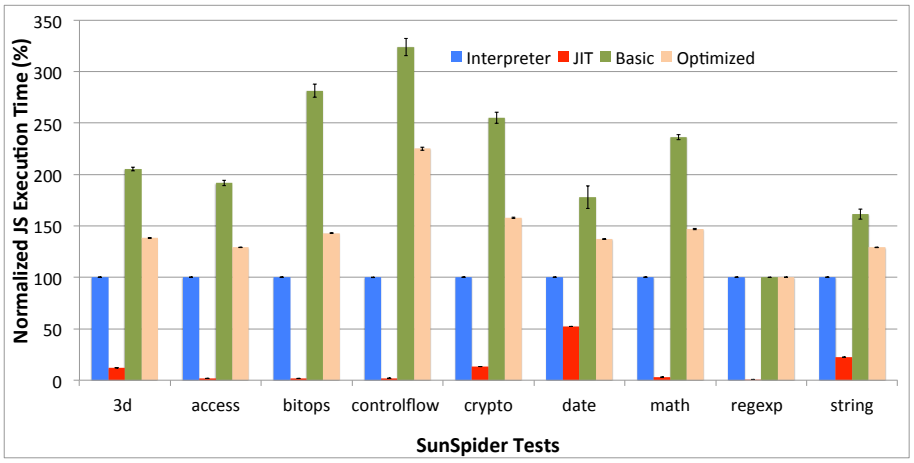

Fig. 3. Overheads of basic and optimized IFC in SunSpider benchmarks

We also implement and evaluate a variant of *sparse labeling* [21] which optimizes the common case of computations that mostly use local variables (registers in the bytecode). Until a function reads a value from the heap with a label different from the *pc*, we propagate taints only on heap-writes, but not on in-register computations. Until that point, all registers are assumed to be implicitly tainted with *pc*. This simple optimization reduces the overhead incurred by taint tracking significantly in microbenchmarks. For both the basic and optimized version, our instrumentation adds approximately 4,500 lines of code to WebKit.

Our baseline for evaluation is the uninstrumented interpreter with JIT disabled. For comparison, we also include measurements with JIT enabled. Our experiments are based on WebKit build #r122160 running in Safari 6.0. The machine has a 3.2GHz Quad-core Intel Xeon processor with 8GB RAM and runs Mac OS X version 10.7.4.

Microbenchmark We executed the standard SunSpider 1.0.1 JS benchmark suite on the uninstrumented interpreter with JIT disabled and JIT enabled, and on the basic and the optimized IFC instrumentations with JIT disabled. Results are shown in Figure 3. The x-axis ranges over SunSpider tests and the y-axis shows the average execution time, normalized to our baseline (uninstrumented interpreter with JIT disabled) and averaged across 100 runs. Error bars are standard deviations. Although the overheads of IFC vary from test to test, the average overheads over our baseline are 121% and 45% for basic IFC and optimized IFC, respectively. The test *regexp* has almost zero overhead because it spends most time in native code, which we have not yet instrumented. We also note that, as expected, the JIT-enabled configuration performs extremely well on the SunSpider benchmarks.

Macrobenchmarks We measured the execution time of the intial JS on 9 popular English language Websites. We load each Website in Safari and measure the total time taken to *execute* the JS code without user interaction. This excludes

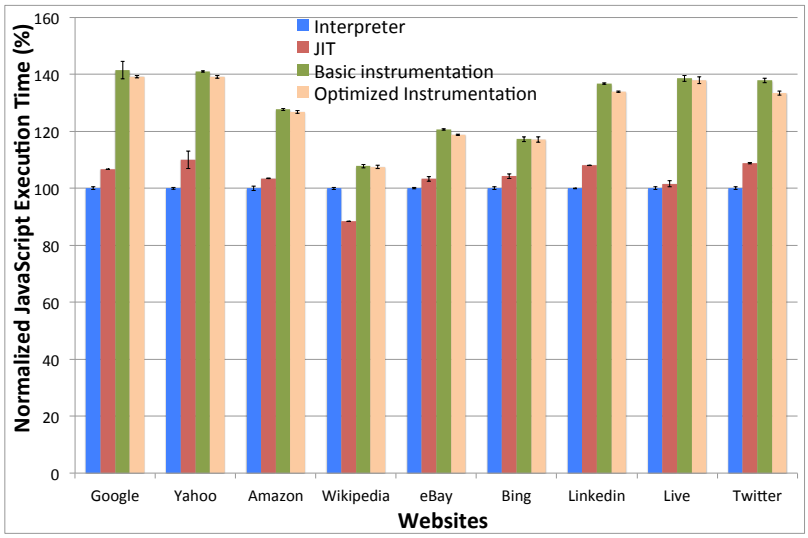

Fig. 4. Overheads of basic and optimized IFC in real websites

time for network communication and internal browser events and establishes a very conservative baseline. The results, normalized to our baseline, are shown in Fig. 4. Our overheads are all less than 42% (with an average of around 29% in both instrumentations). Interestingly, we observe that our optimization is less effective on real websites indicating that real JS accesses the heap more often than the SunSpider tests. When compared to the amount of time it takes to fetch a page over the network and to render it, these overheads are negligible. Enabling JIT worsens performance compared to our baseline indicating that, for the code executed here, JIT is not useful. We also experimented with JSBench [33], a sophisticated benchmark derived from JS code in the wild. The average overhead on all JSBench tests (a total 23 iterations) is approximately 38% for both instrumentations. Details are present in our technical appendix.

7 Conclusion and Future Work

We have explored dynamic information flow control for JS bytecode in WebKit, a production JS engine. We formally model the bytecode, its semantics, our instrumentation and prove the latter correct. We identify challenges, largely arising from pervasive use of unstructured control flow in bytecode, and resolve them using very limited static analysis. Our evaluation indicates only moderate overheads in practice.

In ongoing work, we are instrumenting the DOM and other native JS methods. We also plan to generalize our model and non-interference theorem to take into account the reactive nature of Web browsers. Going beyond non-interference, the design and implementation of a policy language for representing allowed information flows looks necessary.

Acknowledgments This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG) grant "Information Flow Control for Browser Clients" under the priority program "Reliably Secure Software Systems" (RS3) and the German Federal Ministry of Education and Research (BMBF) within the Centre for IT-Security, Privacy and Accountability (CISPA) at Saarland University.

References

1. Richards, G., Hammer, C., Burg, B., Vitek, J.: The eval that men do – a large-scale study of the use of eval in JavaScript applications. In Mezzini, M., ed.: ECOOP '11. Volume 6813 of LNCS. (2011) 52–78
2. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An empirical study of privacy-violating information flows in JavaScript web applications. In: Proc. 17th ACM Conference on Computer and Communications Security. (2010) 270–283
3. Richards, G., Hammer, C., Zappa Nardelli, F., Jagannathan, S., Vitek, J.: Flexible access control for javascript. In: Proc. 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. OOPSLA '13 (2013) 305–322
4. Hedin, D., Sabelfeld, A.: Information-flow security for a core of JavaScript. In: Proc. 25th IEEE Computer Security Foundations Symposium. (2012) 3–18
5. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: Tracking information flow in JavaScript and its APIs. In: Proc. 29th ACM Symposium on Applied Computing. (2014)
6. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: Proc. 2010 IEEE Symposium on Security and Privacy. (2010) 109–124
7. De Groef, W., Devriese, D., Nikiforakis, N., Piessens, F.: Flowfox: a web browser with flexible and precise information flow control. In: Proc. 2012 ACM Conference on Computer and Communications Security. (2012) 748–759
8. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proc. 1982 IEEE Symposium on Security and Privacy. (1982) 11–20
9. Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: Proc. 16th ACM Symposium on Operating Systems Principles. (1997) 129–142
10. Zdancewic, S., Myers, A.C.: Robust declassification. In: Proc. 14th IEEE Computer Security Foundations Workshop. (2001) 15–23
11. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2-3) (January 1996) 167–187
12. Just, S., Cleary, A., Shirley, B., Hammer, C.: Information flow analysis for JavaScript. In: Proc. 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients. (2011) 9–18
13. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: Proc. 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security. (2010) 3:1–3:12
14. Bohannon, A., Pierce, B.C., Sjöberg, V., Weirich, S., Zdancewic, S.: Reactive noninterference. In: Proc. 16th ACM Conference on Computer and Communications Security. (2009) 79–90
15. Maffeis, S., Mitchell, J.C., Taly, A.: An operational semantics for JavaScript. In: Proc. 6th Asian Symposium on Programming Languages and Systems. APLAS '08 (2008) 307–325

16. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of JavaScript. In: Proc. 24th European Conference on Object-Oriented Programming. (2010) 126–150
17. Politz, J.G., Carroll, M.J., Lerner, B.S., Pombrio, J., Krishnamurthi, S.: A tested semantics for getters, setters, and eval in JavaScript. In: Proceedings of the 8th Dynamic Languages Symposium. (2012) 1–16
18. Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., Smith, G.: A trusted mechanised javascript specification. In: Proc. 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (2014)
19. Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., Berg, R.: Saving the world wide web from vulnerable javascript. In: Proc. 2011 International Symposium on Software Testing and Analysis. ISSTA '11 (2011) 177–187
20. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for JavaScript. In: Proc. 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation. (2009) 50–62
21. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Proc. ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security. (2009) 113–124
22. Zdancewic, S.A.: Programming Languages for Information Security. PhD thesis, Cornell University (August 2002)
23. Birgisson, A., Hedin, D., Sabelfeld, A.: Boosting the permissiveness of dynamic information-flow tracking by testing. In: Computer Security – ESORICS 2012. Volume 7459 of LNCS. Springer Berlin Heidelberg (2012) 55–72
24. Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: Proc. 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (2012) 165–178
25. Bielova, N., Devriese, D., Massacci, F., Piessens, F.: Reactive non-interference for a browser model. In: 5th International Conference on Network and System Security (NSS),. (2011) 97–104
26. Bohannon, A., Pierce, B.C.: Featherweight firefox: formalizing the core of a web browser. In: Proc. 2010 USENIX conference on Web application development. WebApps'10 (2010) 11–22
27. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5) (May 1976) 236–243
28. Dhawan, M., Ganapathy, V.: Analyzing information flow in JavaScript-based browser extensions. In: Proc. 2009 Annual Computer Security Applications Conference. ACSAC '09 (2009) 382–391
29. Robling Denning, D.E.: Cryptography and Data Security. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1982)
30. Xin, B., Zhang, X.: Efficient online detection of dynamic control dependence. In: Proc. 2007 International Symposium on Software Testing and Analysis. (2007) 185–195
31. Masri, W., Podgurski, A.: Algorithms and tool support for dynamic information flow analysis. Information & Software Technology **51**(2) (2009) 385–404
32. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. **1**(1) (January 1979) 121–141
33. Richards, G., Gal, A., Eich, B., Vitek, J.: Automated construction of javascript benchmarks. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. (2011) 677–694