Information Flow Control in WebKit’s JavaScript Bytecode

Abhishek Bichhawat¹, Vineet Rajani², Deepak Garg², and Christian Hammer¹

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 [3,11], 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 [4,15]. 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].

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 the technical appendix (Section 8).

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 \cite{26} presents a formal model of a browser based on a reactive model that resembles that of Bohannon et al. \cite{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 $L \sqsubseteq H \sqsubseteq \star$ \cite{13}. For now, readers may ignore $\star$. Our instrumentation works over a more general powerset lattice, whose individual elements are Web domains. We write $r^\ell$ for a value $r$ tagged with label $\ell$.

Information flows can be categorized as explicit and implicit \cite{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 $1 = 0; \text{if } h \{1 = 1;\}$, there is an implicit flow from $h$ to the final value of $1$ (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 $1 = 1$ is executed, the final value of 1 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 $h = 0$, 1 ends with $0^L$; when $h = 1$, 1 ends with $1^H$. Since $0^L$ and $1^H$ 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 \cite{8}.

This problem can be resolved through the well-known no-sensitive-upgrade (NSU) check \cite{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 $\ell$ 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.

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

\footnote{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 \textit{real} IPD, which is outside of this method, is already on the \textit{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 \texttt{throw 9} to \( \diamond \) because \texttt{throw 9} is not handled within \( g \). \( \square \) denotes the IPD of the handler \texttt{catch(e)}

\[
\begin{align*}
\text{function } f() = & \{ \\
& l = 0; \\
& \text{try} \{ g(); \} \text{catch(e)} \{ l = 1; \} \\
& \square \text{return } 1; \\
& \} \diamond \\
\end{align*}
\]

\[
\begin{align*}
\text{function } g() = & \{ \\
& \text{if (h) } \{ \text{throw 9; } \} \\
& \} \diamond \\
\end{align*}
\]

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 \textit{pc}-stack. When executing the condition \texttt{if (h)} we do \textit{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 \texttt{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 \( \diamond L \). If \( h = 1 \), control reaches \( \diamond \) with an unhandled exception. At this point, following the semantics of SEN, we find the exception handler \texttt{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 \textit{pc}-stack. For this, when pushing an IPD onto the \textit{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 \textit{pc}-stack by adding an extra Boolean field to each entry of the \textit{pc}-stack. In summary, each entry of our \textit{pc}-stack is a quadruple containing a security label, a node in the intra-procedural 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, \( l = 0; \text{if} (h) \{ l = 1; \} \), 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 \( \text{if} (h) \{ l = 1; \} \)). 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 \( l = 0; \text{if} (h) \{ l = 1; \} \); \( \text{if} (l) \{ l' = 2 \} \) is halted at the command \( \text{if} (l) \) 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 \( l = 0; \text{if} (h) \{ l = 1; \} \); \( \text{obj.a} = 1 \). This program is insecure in our model: The heap location \( \text{obj.a} \), 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 \( \text{obj.a} \) (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 termina-
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-arguments 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 propid value:r direct:b
  | del-by-id dst:r base:r propid
  | get-pnames dst:r base:r in:size:n breaktarget:offset
  | next-pname dst:r base:r in:size:n target:offset
  | put-getter-setter base:r propid getter:r setter:r
scope-ins := resolve dst:r prop:id
  | resolve-skip dst:r prop:id skip:n
  | resolve-global dst:r propid
  | resolve-base dst:r propid isStrict:bool
  | resolve-with-base bDst:r pDst:r propid
  | 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 \texttt{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 \texttt{inst\_name list\_of\_args} \rangle. The arguments to the instruction are of the form \langle \texttt{var} : \texttt{type} \rangle, where \texttt{var} is the variable name and \texttt{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 \texttt{eval} statement. A code block is generated when a function is created or an \texttt{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 $\zeta$. 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 $\zeta$ and a non-branching node $i \in \zeta$, $\text{Succ}(\zeta, i)$ denotes
$i$’s unique successor. For a conditional branching node $i$, $\text{Left}(\zeta, i)$ and $\text{Right}(\zeta, i)$
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 $(i, \theta, \sigma, \rho)$, where $i$ represents the current
node that is being executed, $\theta$ represents the heap, $\sigma$ represents the call-stack
and $\rho$ is the pc-stack.

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

- An ordinary JS object $N = \{(p_i \mapsto v_i)_{i=0}^{n}, \_\_\text{proto}\_ \mapsto a^\ell_p, \ell_s\}$, 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 $\ell_p$
  and $\ell_s$, $\ell_p$ records the pc where the object was created. $\ell_s$ is the so-called
  structure label, which is an upper bound on all pcs that have influenced
  which fields exist in the object.\footnote{The $\_\_\text{proto}\_\_$ 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.}

- A function object $F = (N, \zeta, \Sigma)$, where $N$ is an ordinary object, $\zeta$ is a CFG,
  which corresponds to the the function stored in the object, and $\Sigma$ is the
  scope chain (closing context) of the function.

A labeled value $v = r^\ell$ is a value $r$ paired with a security label $\ell$. 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 $\sigma$ contains one call-frame for each incomplete function call. A
call-frame $\mu$ contains an array of registers for local variables, a CFG $\zeta$ 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, $\Sigma$, 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 $S ::= O \ | \ V$ and $V ::= v_1 : \ldots : v_n$.\footnote{The $\_\_\text{proto}\_\_$ 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 $\rho$ is a triple $(\ell, \iota, p)$, where $\ell$ is a security label, $\iota$ is a node in a CFG, and $p$ is a pointer to some call-frame on the call stack $\sigma$. (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 $\ell$ together with the IPD $\iota$ of the entry point of the control context and a pointer $p$ to current call-frame. The pair $(\iota, p)$ 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 $\sigma$, 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 $(\ell, \iota, \sigma)$ onto $\rho$ only when $(\iota, \sigma)$ (the IPD) differs from the corresponding pair on the top of the stack and, to handle exceptions correctly, we also require that $\iota$ not be the SEN. Otherwise, we just join $\ell$ 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: $(\iota, \theta, \sigma, \rho) \leadsto (\iota', \theta', \sigma', \rho')$. 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 $\iota'$. 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 $\zeta$ 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 $\Gamma(\text{cond})$ 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
Importantly, when the chain is traversed, the resulting value is labeled with the 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 *. 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 *. The call creates a new call-frame and initializes arguments, the scope chain pointer (initialized with the function object’s $\Sigma$ 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 ($\gamma$).

**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.

The semantics of other bytecodes have been described in Section 8.3.

**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 $\beta$ 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 $\langle \iota_1, \theta_1, \sigma_1, \rho_1 \rangle \sim^{\beta}_\ell \langle \iota_2, \theta_2, \sigma_2, \rho_2 \rangle$, which means that the states on the left and right are equivalent to an observer at level $\ell$, up to the bijection $\beta$ on heap locations. The details have been presented in Section 8.4.

**Theorem 1 (Termination-insensitive non-interference)** Suppose:
1. $\langle \iota_1, \theta_1, \sigma_1, \rho_1 \rangle \sim^{\beta}_\ell \langle \iota_2, \theta_2, \sigma_2, \rho_2 \rangle$, 2. $\langle \iota_1, \theta_1, \sigma_1, \rho_1 \rangle \leadsto^* \langle \text{end}, \theta'_1, [], [] \rangle$, and 3. $\langle \iota_2, \theta_2, \sigma_2, \rho_2 \rangle \leadsto^* \langle \text{end}, \theta'_2, [], [] \rangle$. Then, $\exists \beta' \supseteq \beta$ such that $\theta'_1 \sim^{\beta'}_\ell \theta'_2$.

**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.

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 initial 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 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. The average time for running the benchmark tests on the uninstrumented interpreter with JIT disabled was about 636.11ms with a standard deviation of about 0.30% of the mean. The average time for running the same benchmark tests on the instrumented interpreter and the optimized version was about 874.31ms and 880.85ms respectively, with a standard deviation of about 4.09% and 5.04% of the mean in the two cases.

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., Nikforakis, 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 non-interference. 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., Naudzuniene, 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

8 Appendix

8.1 Data Structures

The formal model described in Section 5 was typechecked in C++. The various data structures used for defining the functions used in the semantics of the language are given in Figure 5. The source-code of the JavaScript program is represented as a structure containing the source and a Boolean flag indicating the strict mode is set or not. The instruction as indicated before is a structure consisting of the opcode and the list of operands. The opcode is a string indicating the operation and the operand is a union of registerIndex, immediateValue, identifier, Boolean, funcIndex and offset. The immediateValue denotes the directly supplied value to an opcode, registerIndex is the index of the register containing the value to be operated upon, identifier represents the string name directly used by the opcode, Boolean is a often a flag indicating the truth value of some parameter and offset represents the offset where the control jumps to. Similarly, functionIndex indicates the index of the function object being invoked.

The function’s source code is represented in the form of a control flow graph (CFG). Formally, it is defined as a struct with a list of CFG nodes, each of which contain the instructions that are to be performed and the edges point to the next instruction in the program. Multiple outgoing edges indicate a branching instruction. It also contains variables indicating the number of variables used by the function code and a reference to the globalObject.

The labels are interpreted as a structure consisting of long integer label. The label represents the value of the label, which are interpreted as bit vectors. A special label star which represents partially leaked data, is used for deferred nonsensitive-upgrade check. The program counter (pc) is represented as a stack of pc-nodes, each of which contains the context label and the IPD of the operation that pushed the node, the callframe of the current node and the handler flag indicating the presence of an exception handler.

Different types of values are used as operands for performing the operations. They include Boolean, integer, string, double and objects or special values like NaN or undefined. These values are associated with a label each and are wrapped by the JSValue class. All the values used in the data structures have the type JSValue. The objects consist of properties, a prototype chain pointer with an associated label and a structure label for the object. The properties are represented as a structure of the propertyName and its descriptor. The descriptor of the property contains the value, some Boolean flags and a property label. The
struct SourceCode{
    String programSrc;
    bool strictMode;
};

typedef char* Opcode;
union Operand{
    int immediateValue;
    String identifier;
    int registerIndex;
    int funcIndex;
    bool flag;
    int offset;
};

struct Instruction{
    Opcode opc;
    Operand* opr;
};

struct CFGNode{
    Instruction* inst;
    struct CFGNode* left;
    struct CFGNode* right;
    struct CFGNode* succ;
};

struct CFG{
    struct CFGNode* cfgNode;
    JSGlobalObject* globalObject;
    int numVars;
    int numFns;
    bool strictMode;
};

union ValueType{
    bool b;
    int n;
    String s;
    double d;
    JSObject* o;
};

union valueTemplate{
    Specials s;
    ValueType v;
};

struct JSValue{
    valueTemplate data;
    JSLabel label;
};

struct PropertyDescriptor{
    JSValue value;
    bool writable;
    bool enumerable;
    bool configurable;
    JSLabel structLabel;
};

struct Property{
    String propertyName;
    PropertyDescriptor* pDesc;
};

struct PropertySlot{
    Property prop;
    PropertySlot* next;
};

struct PropertySlot{
    Property prop;
    PropertySlot* next;
};

struct JSObject{
    Property property[MAX_PROPS];
    struct proto{
        JSLabel l;
        JSObject* __proto__;
    } prototype;
    JSLabel structLabel;
};

struct Heap{
    unsigned location;
    JSObject o;
}[HEAP_SIZE];

enum FunctionType{
    JSFunction, HostFunction
};

struct JSFunctionObject:
    JSObject{
        CFG* funcCFG;
        ScopeChainNode* scopeChain;
        FunctionType fType;
    };}

struct JSGlobalObject:
    JSObject{
        JSFunctionObject evalFunction;
        JSObject* objectPrototype;
        JSObject* functionPrototype;
    };}

struct Register{
    JSValue value;
};

struct CallFrameNode{
    Register rf[REGISTER_FILE_SIZE];
    CFG cfg;
    CFGNode* returnAddress;
    ScopeChainNode* sc;
    JSFunctionObject* callee;
    JSLabel calleeLabel;
    int argCount;
    bool getter;
    int dReg;
};

struct CallFrameStack{
    CallFrameNode cFN;
    CallFrameStack* previous;
};

struct PCNode{
    JSLabel l;
    CFGNode* ipd;
    CallFrameNode* cFN;
    bool handler;
};

struct PCStack{
    PCNode node;
    PCStack* previous;
};

struct JSActivation{
    CallFrameNode* callFrameNode;
    JSLabel structLabel;
};

enum ScopeChainObjectType{
    LexicalObject, VariableObject
};

union SChainObject{
    JSObject* obj;
    JSActivation* actObj;
};

struct ScopeChainNode{
    SChainObject Object;
    ScopeChainObjectType scObjType;
    ScopeChainNode* next;
};

Fig. 5. Data Structures
heap is a collection of objects with an associated memory address. It is essentially a map from location to object.

There are subtypes of JSObject that define the function object and the global object. The function object contains a pointer to the associated CFG and the scope chain. It also contains a field defining the type of function it represents, namely, host or user-defined.

The call-stack is made up of various call-frame nodes, each of which contains a set of registers, the associated CFG, the return address of the function, a pointer to the scope chain, and an exception table. The registers store values and objects and are used as operands for performing the operations. The exception table contains the details about the handlers associated with different instructions in the CFG of the call-frame. The scope chain is a list of nodes containing objects or activation objects along with a label indicating the context in which the object in that node was added. The activation object is a structure containing a pointer to the call-frame node for which the activation object was created.

The next section defines the different procedures used in the semantics of the language. The statement stop implies that the program execution hangs.

### 8.2 Algorithms

The different meta-functions used in the semantics presented in Section 8.3 are described below:

```plaintext
procedure isInstanceOf(JSLabel context, JSValue obj, JSValue protoVal)
  oProto := obj.prototype.__proto__
  while oProto do
    if oProto = protoVal then
      ret := JSValue::construct(true)
      ret.label := context
      return ret
    end if
    oProto := oProto.prototype.__proto__
    context := context.Join(oProto.prototype.l)
  end while
  ret := JSValue::construct(false)
  ret.label := context
  return ret
end procedure

procedure opRet(CallFrameStack* callStack, int ret)
  JSValue retValue = callStack.cFN.rf[ret].value
  if hostCallFrameFlag then
    callStack.pop()
  end if
  return nil, callStack, retValue
end procedure

procedure opCall(CallFrameStack* callStack, CFGNode* ip, int func, int argCount)
  JSValue funcValue := callStack.cFN.rf[func].value
  JSFunctionObject fObj
  CallFrameNode *sigmaTop := new CallFrameNode()
  CallFrameNode *prevTop := callStack.top()
  callStack.push(sigmaTop)
  CallType callType := getCallData(funcValue, &fObj)
```

if callType = CallTypeJS then
    ScopeChainNode* sc := fObj.scopeChain
callStack.cFN.cfg := fObj.funcCFG
callStack.cFN.returnAddress := ip.Succ
callStack.cFN.sc := sc
    args.noOfArgs := argCount
callStack.cFN.argCount := argCount
    for i ← 0, argCount do
        callStack.cFN.rf[sigmaTop.baseRegister() + i].value :=
            prevTop.rf[prevTop.headRegister()-i].value
    end for
    ip := callStack.cFN.cfg.cfgNode
else if callType = CallTypeHost then
    stop
end if
retState.ip := ip
retState.sigma := callStack
return retState
end procedure

procedure opCallEval(JSLabel contextLabel, CallFrameStack* callStack, CFGNode* ip, int func,
                        int argCount)
    JSValue funcValue := callStack.cFN.rf[func].value
    JSFunctionObject* fObj
    JSObject* variableObject
    Argument* arguments
    if isHostEval(funcValue) then
        ScopeChainNode* sc := fObj.scopeChain
callStack.cFN.returnAddress := ip + 1
callStack.cFN.sc := sc
callStack.cFN.argCount := argCount
        SourceCode progSrc := funcValue.getSource()
        Compiler::preparse(progSrc)
        CFG* evalCodeBlock := Compiler::compile(progSrc)
        unsigned numVars := evalCodeBlock.numVariables()
        unsigned numFuncs := evalCodeBlock.numFuncDecls()
        if numVars || numFuncs then
            if evalCodeBlock.strictMode then
                JSActivation* variableObject := new JSActivation()
                variableObject.create(callStack)
                SChainObject* scObj
                scObj.actObj := variableObject
                sc.push(scObj, variableObject, contextLabel)
            else
                for (ScopeChainNode* n := sc;; n := n.next) do
                    if n.isVariableObject() && !n.isLexicalObject() then
                        variableObject := n.getObject()
                        break
                end for
            end if
        end if
        for i ← 0, numVars do
            Identifier iden := evalCodeBlock.variable(i)
            if !variableObject.hasProperty(iden) then
                variableObject.insertVariable(iden)
        end for
        for i ← 0, numFuncs do
            JSFunctionObject* fObj := evalCodeBlock.funcDecl(i)
            variableObject.insertFunction(fObj)
        end for
    end if
    callStack.cFN.cfg := evalCodeBlock
    ip := evalCodeBlock.cfgNode
    retState.ip := ip
    retState.sigma := callStack
    return retState
else
  return opCall(contextLabel, callStack, ip, func, argCount)
end if

procedure createArguments(Heap h, CallFrameStack callStack)
  JSObject jsArgument := JSArgument::create(h, callStack)
  h.o[++] := *jsArgument
  retState.theta := h
  retState.val := JSValue::construct(jsArgument)
  return retState
end procedure

procedure newFunc(CallFrameStack callStack, Heap heap, int funcIndex, JSLabel context)
  CFG cBlock := callStack.cFN.cfg
  SourceCode fcCode := cBlock.getFunctionSrc(funcIndex)
  CFG fcBlock := Compiler::compile(fcCode, callStack.cFN)
  JSFunctionObject fObj := JSFunctionObject::create(fcBlock, callStack.cFN.sc)
  fObj.structLabel := context
  heap.o[++] := *fObj
  retState.theta := heap
  retState.val := JSValue::construct(fObj)
  return retState
end procedure

procedure createActivation(CallFrameStack callStack, JSLabel contextLabel)
  JSActivation jsActivation := new JSActivation()
  jsActivation.create(callStack)
  jsActivation.structLabel := contextLabel
  SChainObject scObj
  scObj.actObj := jsActivation
  JSValue vActivation := JSValue::jsValuefromActivation(jsActivation)
  if callStack.cFN.scopeLabel ≥ contextLabel then
    callStack.cFN.sc.push(scObj, VariableObject, contextLabel)
    callStack.cFN.scopeLabel := contextLabel
  else
    stop
  end if
  return retState
end procedure

procedure createThis(JSLabel contextLabel, CallFrameStack callStack, Heap h)
  JSFunctionObject callee := callStack.cFN.callee
  PropertySlot p(callee)
  String str := "prototype"
  JSValue proto := p.getValue(str)
  JSObject obj := new JSObject()
  obj.structLabel := contextLabel
  obj.prototype.__proto__ := proto.toObject()
  obj.prototype.l := proto.toObject().structLabel.join(contextLabel)
  h.o[++] := *obj
  retState.theta := h
  retState.val := JSValue::construct(obj)
  return retState
end procedure

procedure newObject(Heap h, JSLabel contextLabel)
  JSObject obj := emptyObject()
  obj.structLabel := contextLabel
  obj.prototype.__proto__ := ObjectPrototype::create()
  obj.prototype.l := contextLabel
  h.o[++] := *obj
  retState.theta := h

else
  return opCall(contextLabel, callStack, ip, func, argCount)
end if
retState.val := JSValue::construct(obj)
return retState
end procedure

procedure getPropertyById(JSValue v, String p, int dst)
JSObject* O := v.toObject()
JSLabel label := O.structLabel
JSValue ret := jsUndefined()
if O.isUndefined() then
  ret.label := label
  return ret
end if
while O ≠ null do
  if O.containsProperty(p) then
    if p.isGetter() then
      JSValue v = p.getValue()
      JSFunctionObject* funcObj = (JSFunctionObject*) v.toObject()
      CallFrameNode *sigmaTop = new CallFrameNode()
      callStack.push(sigmaTop)
      ScopeChainNode* sc = fObj.scopeChain
      CFG* newCodeBlock = fObj.funcCFG
      callStack.cFN.cfg = *newCodeBlock
      callStack.cFN{returnAddress = ip + 1
      callStack.cFN.sc = sc
      callStack.cFN.getter = true
      callStack.cFN.dReg = dst
      ip = newCodeBlock.cfgNode
      interpreter.iota = ip
      interpreter.sigma = callStack
    else
      ret := getProperty(p).getValue()
      ret.label := label
      return ret
    end if
  else
  O := O.prototype.__proto__
  label := label.join(label)
end while
end procedure

procedure putDirect(JSLabel contextLabel, CallFrameStack* callStack, Heap* h, int base, String property, int propVal)
JSValue baseValue := callStack.cFN.rf[base].value
JSValue propValue := callStack.cFN.rf[propVal].value
JSObject* obj := baseValue.toObject()
PropertyDescriptor dataPD := PropertyDescriptor::createPD(true, true, true)
dataPD.value := propValue
obj.setProperty(property, dataPD)
obj.structLabel := obj.structLabel.join(contextLabel)
h.o[++(h.location)] := *obj
return h
end procedure

procedure putIndirect(JSLabel contextLabel, CallFrameStack* callStack, Heap* h, int base, String property, int val)
JSValue baseValue := callStack.cFN.rf[base].value
JSValue propValue := callStack.cFN.rf[val].value
JSObject* obj := baseValue.toObject()
bool isStrict := callStack.cFN.cfg.isStrictMode()
contextLabel := obj.structLabel.join(contextLabel)
if obj.containsPropertyInItself(property) && obj.getProperty(property).isDataProperty() && !isStrict && obj.isWritable() then
  obj.getProperty(property).setValue(propValue)
  h.o[++(h.location)] := *obj
return h
end if
return putDirect(contextLabel, callStack, h, base, property, val)
end procedure

procedure delById(JSLabel contextLabel, CallFrameStack* callStack, Heap* h, int base, Identifier property)
JSValue baseValue := callStack.cFN.rf[base].value
JSObject* obj := baseValue.toObject()
int loc := h.findObject(obj)
Property prop := obj.getProperty(property)
PropertyDescriptor pd := prop.getPropertyDescriptor()
if obj.getPropertyValue(prop).label ≥ contextLabel then
  if !obj.containsPropertyInItself(property) then
    retState.theta := h
    retState.val := JSValue::construct(true)
    return
  end if
  if obj.containsPropertyInItself(property) && prop.isConfigurable() then
    if !(callStack.cFN.cfg.isStrictMode()) then
      pd.value := JSValue::constructUndefined()
      obj.setProperty(property, pd)
      h.o[loc] := *obj
      retState.theta := h
      retState.val := JSValue::construct(true)
      return
    end if
  end if
  retState.theta := h
  retState.val := JSValue::construct(false)
  return
else
  stop
end if
end procedure

procedure putGetterSetter(JSLabel contextLabel, CallFrameStack* callStack, Heap* h, int base, Identifier property, JSValue getterValue, JSValue setterValue)
JSValue baseValue := callStack.cFN.rf[base].value
JSObject* obj := baseValue.toObject()
int loc := h.findObject(obj)
JSFunctionObject *getterObj, *setterObj
JSFunctionObject *getterFuncObj := null, *setterFuncObj := null
if !getterValue.isUndefined() then
  getterFuncObj := getterValue.toFunctionObject (callStack.cFN.cfg, callStack.cFN.sc)
end if
if !setterValue.isUndefined() then
  setterFuncObj := setterValue.toFunctionObject (callStack.cFN.cfg, callStack.cFN.sc)
end if
if getterFuncObj = null then
  obj.setGetter(property, getterObj)
end if
if setterFuncObj ≠ null then
  obj.setSetter(setterObj)
end if
PropertyDescriptor accessor := PropertyDescriptor ::createPD(false, false, false, true)
JSValue v := JSValue::constructUndefined()
v.label := contextLabel
accessor.value := v
obj.setProperty(property, accessor)
obj.structLabel := contextLabel
h.o[loc] := *obj
return h
end procedure
procedure getPropNames(CallFrameStack* callStack, Instruction* ip, int base, int i, int size, int breakOffset)
    JSValue baseVal := callStack.cFN.rf[base].value
    JLObject* obj := baseVal.toObject()
    PropertyIterator* propItr := obj.getProperties()
    if baseVal.isUndefined() || baseVal.isNull() then
        retState.v1 := jsUndefined()
        retState.v2 := jsUndefined()
        retState.v3 := jsUndefined()
        retState.ip := ip + breakOffset
        return retState
    end if
    retState.v1 := JSValue::construct(propItr)
    retState.v2 := JSValue::construct(0)
    retState.v3 := JSValue::construct(propItr.size())
    retState.ip := ip + 1
    return retState
end procedure

procedure getNextPropName(CallFrameStack* cStack, Instruction* ip, JSValue base, int i, int size, int iter, int offset, int dst)
    JLObject* obj := base.toObject()
    PropertyIterator* propItr := cStack.cFN.rf[iter].value. toPropertyIterator()
    int b := rFile[i].value.toInteger()
    int e := rFile[size].value.toInteger()
    while b < e do
        String key := propItr.get(b)
        retState.value1 := JSValue::construct(b + 1)
        if !(key.isNull()) then
            retState.value2 := JSValue::construct(key)
            ip := ip + offset
            break
        end if
        b++
    end while
    return retState
end procedure

procedure resolveInSc(JSLabel contextLabel, ScopeChainNode* scopeHead, String property)
    JSValue v
    JSLabel l
    ScopeChainNode* scn := scopeHead
    while scn \neq NULL do
        PropertySlot pSlot := scn.getPropertySlot()
        if pSlot.contains(property) then
            v := pSlot.getValue(property)
            v.label := contextLabel
            return v
        end if
        scn := scn.next
        if scn.scObjType = VariableObject then
            contextLabel = contextLabel.join(scn.Object. actObj.structLabel)
        else if scn.scObjType = LexicalObject then
            contextLabel = contextLabel.join(scn.Object. obj.structLabel)
        end if
        contextLabel := contextLabel.join(scn. scopeNextLabel)
    end while
    v := jsUndefined()
    v.label := contextLabel
    return v
end procedure

procedure resolveInScWithSkip(JSLabel contextLabel, ScopeChainNode* scopeHead, String property, int skip)
    JSValue v
JSLabel l
ScopeChainNode* scn := scopeHead
while skip—— do
    scn := scn.next
    if scn.scObjType = VariableObject then
        contextLabel := contextLabel.join( scn.Object. actObj.structLabel)
    else if scn.scObjType = LexicalObject then
        contextLabel := contextLabel.join(scn.Object. obj.structLabel)
    end if
    contextLabel := contextLabel.join(scn. scopeNextLabel)
end while
while scn ≠ null do
    PropertySlot pSlot := scn.getPropertySlot()
    if pSlot.contains(property) then
        v := pSlot.getValue(property)
        v.label := contextLabel
        return v
    end if
    scn := scn.next
    if scn.scObjType = VariableObject then
        contextLabel := contextLabel.join(scn.Object. actObj.structLabel)
    else if scn.scObjType = LexicalObject then
        contextLabel := contextLabel.join(scn.Object. obj.structLabel)
    end if
    contextLabel := contextLabel.join(scn. scopeNextLabel)
end while
v := jsUndefined()
return v
end procedure

procedure resolveGlobal(JSLabel contextLabel, CallFrameStack* cStack, String property)
JSValue v
struct CFG* cBlock := cStack.cFN.cfg
JSGlobalObject* globalObject := cBlock. getGlobalObject()
PropertySlot pSlot(globalObject)
if pSlot.contains(property) then
    v := pSlot.getValue(property)
    v.label := contextLabel
    return v
end if
v := jsUndefined()
return v
end procedure

procedure resolveBase(JSLabel contextLabel, CallFrameStack* cStack, ScopeChainNode* scope-Head, String property, bool strict)
JSValue v
ScopeChainNode* scn := scopeHead
CFG *cBlock := cStack.cFN.cfg
JSGlobalObject *gObject := cBlock. getGlobalObject()
while scn ≠ null do
    JSObject* obj := scn.get()
    contextLabel := obj.structLabel.join(contextLabel)
    PropertySlot pSlot(obj)
    if scn.next = null &
    PropertySlot pSlot(obj)
    if scn.next = null &
        & !pSlot.contains (property) then
        v := emptyJSValue()
        v.label := contextLabel
        return v
    end if
    if pSlot.contains(property) then
        v := JSValueContainingObject(obj)
        v.label := contextLabel
        return v
    end if
end if
procedure resolveBaseAndProperty(JSLabel contextLabel, CallFrameStack cStack, int bRegister, int pRegister, String property)

JSValue v
ScopeChainNode* scn := cStack.cFN.sc
while scn \neq null do
    JSObject* obj := scn.get()
    contextLabel := obj.structLabel.join(contextLabel)
    PropertySlot pSlot(obj)
    if pSlot.contains(property) then
        v := pSlot.getValue(property)
        v.label := contextLabel
        ret.val1 := v
        v := JSValueContainingObject(obj)
        v.label := contextLabel
        ret.val2 := v
        return ret
    end if
    scn := scn.next
    if scn \neq null then
        contextLabel := contextLabel.join(scn.scopeNextLabel)
    end if
end while
end procedure

procedure getScopedVar(JSLabel contextLabel, CallFrameStack* callStack, Heap* h, int index, int skip)

JSValue v
ScopeChainNode* scn := callStack.cFN.sc
while skip-- do
    if scn.scObjType = VariableObject then
        contextLabel := contextLabel.join(scn.Object.actObj.structLabel)
    else if scn.scObjType = LexicalObject then
        contextLabel := contextLabel.join(scn.Object.obj.structLabel)
    end if
    contextLabel := contextLabel.join(scn.scopeLabel)
    scn := scn.next
end while
v := scn.registerAt(index)
if scn.scObjType = VariableObject then
    v.label := contextLabel.join(scn.Object.actObj.structLabel)
else if scn.scObjType = LexicalObject then
    v.label := contextLabel.join(scn.Object.obj.structLabel)
end if
return v
end procedure

procedure putScopedVar(JSLabel contextLabel, CallFrameStack* callStack, Heap* h, int index, int skip, int value)

CallFrameStack* cStack
ScopeChainNode* scn := callStack.cFN.sc
JSValue val := callStack.cFN.rf[value].value
while skip-- do
    if scn.scObjType = VariableObject then
        contextLabel := contextLabel.join(scn.Object.actObj.structLabel)
    else if scn.scObjType = LexicalObject then
        contextLabel := contextLabel.join(scn.Object.obj.structLabel)
    end if
end while
v := scn.registerAt(index)
if scn.scObjType = VariableObject then
    v.label := contextLabel.join(scn.Object.actObj.structLabel)
else if scn.scObjType = LexicalObject then
    v.label := contextLabel.join(scn.Object.obj.structLabel)
end if
return v
end procedure
contextLabel := contextLabel.join(scn.Object. obj.structLabel)
end if
contextLabel := contextLabel.join(scn. scopeLabel)
scn := scn.next
end while
cStack := scn.setRegisterAt(contextLabel, index, val)
return cStack
end procedure

procedure pushScope(JSLabel contextLabel, CallFrameStack* callStack, Heap* h, int scope)
ScopeChainNode* sc := callStack.cFN.sc
JSValue v := callStack.cFN.rf[scope].value
JSObject* o := v.toObject()
SChainObject* scObj
if sc.scopeLabel ≥ contextLabel then
  scObj.obj := o
  sc.push(scObj, LexicalObject, contextLabel)
callStack.cFN.sc := sc
else if sc.scopeLabel = star then
  scObj.obj := o
  sc.push(scObj, LexicalObject, star)
callStack.cFN.sc := sc
end if
return callStack
end procedure

procedure popScope(JSLabel contextLabel, CallFrameStack* callStack, Heap* h)
ScopeChainNode* sc := callStack.cFN.sc
JSLabel l := sc.scopeLabel
if l ≥ contextLabel then
  sc.pop()
callStack.cFN.sc := sc
else
  stop
end if
return callStack
end procedure

procedure jmpScope(JSLabel contextLabel, CallFrameStack* callStack, Heap* h, int count)
ScopeChainNode* sc := callStack.cFN.sc
while count-- > 0 do
  JSLabel l := sc.scopeLabel
  if l ≥ contextLabel then
    sc.pop()
callStack.cFN.sc := sc
  else
    stop
  end if
end while
return callStack
end procedure

procedure throwException(CallFrameStack* callStack, CFGNode* iota)
CFGNode* handler
while callStack.cFN.hasHandler()==false do
  callStack.pop()
end while
while callStack.cFN.sc.length() - callStack.cFN.getHandlerScopeDepth() do
  callStack.cFN.sc.pop()
end while
handler := callStack.cFN.getHandler(iota)
interpreter.iota := handler
interpreter.sigma := callStack
end procedure
8.3 Semantics

\( i = \text{“prim } dst:r \ src1:r \ src2:r” \)

\[
\begin{align*}
\mathcal{L} & := \Gamma(\sigma(\text{src1})) \cup \Gamma(\sigma(\text{src2})) \cup \Gamma(\langle p \rangle) \\
\mathcal{V} & := \Upsilon(\sigma(\text{src1})) \oplus \Upsilon(\sigma(\text{src2})) \\
(\Gamma(\sigma(\text{dst})) \geq \Gamma(\langle p \rangle)) & \Rightarrow (\mathcal{L} := \mathcal{L} \circ (\mathcal{L} := \ast)) \\
\sigma’ & := \sigma \left[ \Upsilon(\sigma(\text{dst})) := \mathcal{V} \right] \\
\iota’ & := \text{Succ}(\sigma’.\text{CFG}, i) \\
\rho’ & := \text{isIPD}(\iota’, \rho, \sigma’)
\end{align*}
\]

`prim` reads the values from two registers (src1 and src2), performs the binary operation generically denoted by \( \oplus \), and writes the result into the dst register. The label assigned to the value in dst register is the join of the label of value in src1, src2 and the head of the pc-stack (\( \langle p \rangle \)). In order to avoid implicit leak of information, the label of the existing value in dst is compared with the current context label. If the label is lower than the context label, the label of the value in dst is set to \( \ast \).

\( i = \text{“mov } dst:r \ src:r” \)

\[
\begin{align*}
\mathcal{L} & := \Gamma(\sigma(\text{src})) \\
\mathcal{V} & := \Upsilon(\sigma(\text{src})) \\
(\Gamma(\sigma(\text{dst})) \geq \Gamma(\langle p \rangle)) & \Rightarrow (\mathcal{L} := \mathcal{L} \circ (\mathcal{L} := \ast)) \\
\sigma’ & := \sigma \left[ \Upsilon(\sigma(\text{dst})) := \mathcal{V} \right] \\
\iota’ & := \text{Succ}(\sigma’.\text{CFG}, i) \\
\rho’ & := \text{isIPD}(\iota’, \rho, \sigma’)
\end{align*}
\]

`mov` copies the value from the src register to the dst register. The label assigned to the value in dst register is the join of the label of value in src and the head of the pc-stack (\( \langle p \rangle \)). In order to avoid implicit leak of information, the label of the existing value in dst is compared with the current context label. If the label is lower than the context label, the label of the value in dst is joined with \( \ast \).

\( i = \text{“jfalse } cond:r \ target:offset” \)

\[
\begin{align*}
\Gamma(\sigma(\text{cond})) & \neq \ast \\
\rho'' & := \rho.\text{push}(\Gamma(\sigma(\text{cond})) \cup \Gamma(\langle p \rangle), \text{IPD}(i), \text{CF}(i), \text{false}) \\
\Upsilon(\sigma(\text{cond})) & = \text{false} \Rightarrow \iota’ := \text{Left}(\sigma.\text{CFG}, i) \circ \iota’ := \text{Right}(\sigma.\text{CFG}, i) \\
\rho’ & := \text{isIPD}(\iota’, \rho'', \sigma)
\end{align*}
\]

`jfalse` is a branching instruction. Based on the value in the cond register, it decides which branch to take. The operation is performed only if the value in cond is not labelled with a \( \ast \). If it contains a \( \ast \), we terminate the execution to prevent possible leak of information. The push function defined in the rule does the following: A node is pushed on the top of the pc-stack containing the IPD of the branching instruction and the label of the value in cond joined with the context, to define the context of this branch. If the IPD of the instruction is SEN or the same as the top of the pc-stack, then we just join the label on top of the pc-stack with the context label determined by the cond register.

\( i = \text{“loop-if-less } src1:r \ src2:r \ target:offset” \)

\[
\begin{align*}
\Gamma(\sigma(\text{src1})) & \neq \ast \\
\Gamma(\sigma(\text{src2})) & \neq \ast \\
\mathcal{L} & := \Gamma(\sigma(\text{src1})) \cup \Gamma(\sigma(\text{src2})) \cup \Gamma(\langle p \rangle) \\
\mathcal{V} & := \Upsilon(\sigma(\text{src1})) \oplus \Upsilon(\sigma(\text{src2})) \\
(\Gamma(\sigma(\text{src1})) < \mathcal{V}) & \Rightarrow \iota’ := \text{Left}(\sigma.\text{CFG}, i) \\
\rho'' & := \rho.\text{push}(\mathcal{L}, \text{IPD}(i), \text{CF}(i), \text{false}) \\
\rho’ & := \text{isIPD}(\iota’, \rho'', \sigma)
\end{align*}
\]

`loop-if-less` is another branching instruction. If the value of src1 is less than src2, then it jumps to the target, else continues with the next instruction. The
operation is performed only if the values in src\textsubscript{1} and src\textsubscript{2} are not labelled with a \( \star \). If any one of them contains a \( \star \), we abort the execution to prevent possible leak of information. The push function defined in the rule does the following: A node is pushed on the top of the pc-stack containing the IPD of the branching instruction and the join of the label of the values in src\textsubscript{1} and src\textsubscript{2} joined with the context, to define the context of this branch. If the IPD of the instruction is SEN or the same as the top of the pc-stack, then we just join the label on top of the pc-stack with the context label determined above.

\[
\begin{align*}
\iota &= \text{"typeof dst: r src:r"} \\
\mathcal{L} &= (\Gamma(\text{src}) \sqcup \Gamma(\text{!}(\rho))) \quad \mathcal{V} := \text{determineType}(\sigma(\text{src})) \\
(\Gamma(\text{!}(\sigma(\text{dst}))) \supseteq \Gamma(\text{!}(\rho))) &\Rightarrow (\mathcal{L} := \mathcal{L}) \circ (\mathcal{L} := \star)
\end{align*}
\]

typeof: 
\[
\begin{align*}
\sigma' := \sigma \left[ \begin{array}{l}
\mathcal{Y}(\sigma(\text{dst})) := \mathcal{V} \\
\mathcal{Y}(\sigma(\text{src})) := \mathcal{L}
\end{array} \right] \\
\iota' := \text{Succ}(\sigma', \text{CFG}, \iota) \\
\rho' := \text{isIPD}(\iota' , \rho, \sigma')
\end{align*}
\]

typeof determines the type string for src according to ECMAScript rules, and puts the result in register dst. We do a deferred NSU check on dst before writing the result in it. The determineType function returns the data type of the value passed as the parameter.

\[
\begin{align*}
\iota &= \text{"instanceof dst: r value: r cProt: r"} \\
v := \text{isInstanceOf}(\Gamma(\rho), \sigma(\text{value}), \sigma(\text{cProt})) \quad \mathcal{L} = \mathcal{Y}(v) \quad \mathcal{V} = \mathcal{Y}(v), \\
(\Gamma(\sigma(\text{dst})) \supseteq \Gamma(\text{!}(\rho))) &\Rightarrow (\mathcal{L} := \mathcal{L}) \circ (\mathcal{L} := \star),
\end{align*}
\]

instanceof: 
\[
\begin{align*}
\sigma' := \sigma \left[ \begin{array}{l}
\mathcal{Y}(\sigma(\text{dst})) := \mathcal{V} \\
\mathcal{Y}(\sigma(\text{src})) := \mathcal{L}
\end{array} \right] \\
\iota' := \text{Succ}(\sigma', \text{CFG}, \iota) \\
\rho' := \text{isIPD}(\iota' , \rho, \sigma')
\end{align*}
\]

instanceof tests whether the cProt is in the prototype chain of the object in register value and puts the Boolean result in the dst register after deferred NSU check.

\[
\begin{align*}
\iota &= \text{"enter"} \\
\iota' := \text{Succ}(\sigma.CFG, \iota) \\
\rho' := \text{isIPD}(\iota', \rho, \sigma)
\end{align*}
\]

tenter marks the beginning of a code block.

\[
\begin{align*}
\iota &= \text{"ret res: r"} \\
(i', \sigma', \gamma) := \text{opRet}(\sigma, \text{res}) \\
\rho' := \text{isIPD}(\iota', \rho, \sigma')
\end{align*}
\]

ret is the last instruction to be executed in a function. It pops the call-frame and returns the control to the callee’s call-frame. The return value of the function is written to a local variable in the interpreter (\( \gamma \)), which can be read by the next instruction being executed.

\[
\begin{align*}
\iota &= \text{"end res: r"} \\
\text{opEnd}(\sigma, \text{res})
\end{align*}
\]

dend marks the end of a program. opEnd passes the value present in res register to the caller (the native function that invoked the interpreter).
\[\iota = \text{"op-call func: args:n"} \]
\[
\text{call: } \rho'' := \rho.\text{push}(\ell_f \sqcup \Gamma(\iota, \sigma', \text{func}, \text{args})) \quad \text{\(i' = \text{opCall}(\sigma, \iota, \text{func}, \text{args})\)}
\]

\[\iota, \theta, \sigma, \rho \leadsto i', \theta, \sigma', \rho'\]

\(\text{call, initially, checks the function object's label for } \star \text{ and if the label contains a } \star, \text{ the program execution is aborted. The reason for termination is the possible leak of information as explained above. If not, call creates a new call-frame, copies the arguments, initializes the registers, scope-chain pointer, codeblock and the return address. The registers are initialized to } \text{undefined} \text{ and assigned a label obtained by joining the label of the context in which the function was created and the label of the function object itself. We treat } \text{call} \text{ as a branching instruction and hence, push a new node on the top of the pc-stack with the label determined above along with its IPD and call-frame. The field } \mathcal{H} \text{ in the push function is determined by looking up the exception table. If it contains an associated exception handler, it sets the field to } \text{true} \text{ else it is set to } \text{false}. \text{ If the IPD is the SEN then we just join the label on the top of the stack with the currently calculated label. It then points the instruction pointer to the first instruction of the new code block.}\]

\[\iota = \text{"call-put-result res:r"} \]
\[
\mathcal{L} := \Gamma(\gamma) \sqcup \Gamma(\rho) \quad \mathcal{V} := \Upsilon(\gamma) \quad (\Gamma(\iota(\mathcal{res})) = \mathcal{V}) \Rightarrow \mathcal{L} := \mathcal{L} \circ \mathcal{L} := \star
\]

\[\text{call-put-result: } \sigma' := \sigma' \left[\mathcal{U}(\iota(\mathcal{res})) = \mathcal{V} \mid \gamma = \Gamma(\sigma(\mathcal{res})) \sqcup \Gamma(\rho) \Rightarrow \mathcal{L} = \mathcal{L} \sqcup \mathcal{L} = \star\right] \quad i' := \text{Succ}(\sigma', \mathcal{CFG}, \iota) \quad \rho' := \text{isIPD}(i', \rho, \sigma')
\]

\[\iota, \theta, \sigma, \rho \leadsto i', \theta, \sigma', \rho'\]

\(\text{call-put-result} \text{ copies the return value } \gamma \text{ to the } \mathcal{res} \text{ register. The label assigned to the value in } \mathcal{res} \text{ register is the join of the label of the return value and the head of the pc-stack. In order to avoid implicit leak of information, } \text{deferred no-sensitive-upgrade} \text{ is performed.}\]

\[\iota = \text{"call-eval func:r args:n"} \]
\[
\Gamma(\iota(\mathcal{func})) \neq \star \quad \text{\(i' = \text{opCallEval}(\Gamma(\rho), \sigma, \iota, \text{func}, \text{args})\)}
\]
\[
\text{call-eval: } \rho'' := \rho.\text{push}(\ell_f \sqcup \Gamma(\iota, \sigma, \text{func})) \quad \text{\(i' = \text{opCallEval}(\Gamma(\rho), \sigma, \iota, \text{func}, \text{args})\)}
\]
\[\iota, \theta, \sigma, \rho \leadsto i', \theta, \sigma', \rho'\]

\(\text{call-eval} \text{ calls a function with the string passed as an argument converted to a code block. If } \text{func} \text{ register contains the original global eval function, then it is performed in local scope, else it is similar to } \text{call}.\]

\[\iota = \text{"create-arguments dst:r"} \]
\[
(\theta', v) := \text{createArguments}(\theta, \sigma) \quad \mathcal{L} := \Gamma(\rho) \quad \mathcal{V} := \Upsilon(v) \quad (\Gamma(\iota(\mathcal{dst})) = \mathcal{V}) \Rightarrow \mathcal{L} := \mathcal{L} \circ \mathcal{L} := \star
\]
\[
\sigma' := \sigma' \left[\mathcal{U}(\iota(\mathcal{dst})) = \mathcal{V} \mid \gamma = \Gamma(\sigma(\mathcal{dst})) \sqcup \Gamma(\rho) \Rightarrow \mathcal{L} = \mathcal{L} \sqcup \mathcal{L} = \star\right] \quad i' := \text{Succ}(\sigma', \mathcal{CFG}, \iota) \quad \rho' := \text{isIPD}(i', \rho, \sigma')
\]
\[\iota, \theta, \sigma, \rho \leadsto i', \theta', \sigma', \rho'\]

\(\text{create-arguments} \text{ creates the arguments object and places its pointer in the local } \mathcal{dst} \text{ register after the deferred NSU check. The label of the arguments object is set to the context.}\]
The prototype chain pointer is also labelled with the context label.

new-func constructs a new function instance from function at funcIndex and the current scope chain and puts the result in dst after deferred NSU check.

create-activation creates the activation object for the current call-frame if it has not been already created and writes it to the dst after the deferred NSU check and pushes the object in the scope-chain. If the label of the head of the existing scope-chain is less than the context, then the label of the pushed node is set to *, else it is set to the context.

construct invokes register func as a constructor and is similar to call. For JavaScript functions, the this object being passed (the first argument in the list of arguments) is a new object. For host constructors, no this is passed.

create-this creates and allocates an object as this used for construction later in the function. The object is labelled the context and placed in dst after deferred NSU check. The prototype chain pointer is also labelled with the context label.

new-object constructs a new empty object instance and puts it in dst after deferred NSU check. The object is labelled with the context label and the prototype chain pointer is also labelled with the context.
\begin{align*}
\lambda & = \text{"get-by-id dst:r base:r prop:id vdstr:"} \\
& v := \text{getPropertyByld}(\langle \sigma(\text{base}), \text{prop, vdst} \rangle) \quad \mathcal{L} := \Gamma(v) \sqcup \Gamma(\lambda) \quad \mathcal{V} := T(v) \\
& (\Gamma(\langle \sigma(\text{dst}) \rangle) \geq \Gamma(\lambda)) \Rightarrow (\mathcal{L} := \mathcal{L}) \circ (\mathcal{L} := \ast) \\
& \text{get-by-id:} \quad \sigma' := \sigma \left[ \tau_{\langle \sigma(\text{dst}) \rangle} = V \right] \quad i' := \text{Succ}(\sigma'.CFG, i) \quad \rho' := \text{isIPD}(i', \rho, \sigma') \\
& i, \theta, \sigma, \rho \rightsquigarrow i', \theta', \sigma', \rho'
\end{align*}

\textit{get-by-id} gets the property named by the identifier \textit{prop} from the object in the \textit{base} register and puts it into the \textit{dst} register after the deferred NSU check. If the object does not contain the property, it looks up the prototype chain to determine if any of the proto objects contain the property. When traversing the prototype chain, the context is joined with the structure label of all the objects and the prototype chain pointer labels until the property is found or the end of the chain. It then joins the property label to the context. If the property is not found, it returns \textit{undefined}. The joined label of the context is the label of the property put in the \textit{dst} register.

If the property is an accessor property, it calls the getter function, sets the getter flag in the call-frame and updates the destination register field with the register where the value is to be inserted. It then transfers the control to the first instruction in the getter function.

\begin{align*}
\lambda & = \text{"put-by-id base:r prop:id value:r direct:b"} \\
& \Gamma(\langle \sigma(\text{value}) \rangle) \neq \ast \\
& (\text{direct} = \text{true} \Rightarrow \theta' := \text{putDirect}(\Gamma(\lambda), \sigma, \theta, \text{base, prop, value}) \circ \theta' := \text{putIndirect}(\Gamma(\lambda), \sigma, \theta, \text{base, prop, value})) \\
& i' := \text{Succ}(\sigma'.CFG, i) \quad \rho' := \text{isIPD}(i', \rho, \sigma') \\
& i, \theta, \sigma, \rho \rightsquigarrow i', \theta', \sigma', \rho'
\end{align*}

\textit{put-by-id} writes into the heap the property of an object. We check for \ast in the label of \textit{value} register. If it contains a \ast, the program aborts as this could potentially result in an implicit information flow. If not, it writes the property into the object. The basic functionality is to search for the property in the object and its prototype chain, and change it. If the property is not found, a new property for the current object with the property label as the context is created. Based on whether the property is in the object itself (or needs to be created in the object itself) or in the prototype chain of the object, it calls \textit{putDirect} and \textit{putIndirect}, respectively.

\begin{align*}
\lambda & = \text{"del-by-id dst:r base:r prop:id"} \\
& \Gamma(\langle \sigma(\text{base}) \rangle) \neq \ast \\
& (\langle \theta', v \rangle := \text{delByld}(\Gamma(\lambda), \sigma, \theta, \text{base, prop}) \circ \mathcal{L} := \Gamma(v) \sqcup \Gamma(\lambda) \\
& \mathcal{V} := T(v) \\
& (\Gamma(\langle \sigma(\text{dst}) \rangle) \geq \Gamma(\lambda)) \Rightarrow (\mathcal{L} := \mathcal{L}) \circ (\mathcal{L} := \ast) \\
& \sigma' := \sigma \left[ \tau_{\langle \sigma(\text{dst}) \rangle} = V \right] \quad i' := \text{Succ}(\sigma'.CFG, i) \quad \rho' := \text{isIPD}(i', \rho, \sigma') \\
& i, \theta, \sigma, \rho \rightsquigarrow i', \theta', \sigma', \rho'
\end{align*}

\textit{del-by-id} deletes the property specified by \textit{prop} in the object contained in \textit{base}. If the structure label of the object is less than the context, the deletion does not happen. If the property is not found, the property is deleted and Boolean value \textit{true} is written to \textit{dst}, else it writes \textit{false} to \textit{dst}. The label of the Boolean value is the structure label of the object joined with the property label.
getter-setter puts the accessor descriptor to the object in register base. It initially checks if the structure label of the object is greater or equal to the context. The property for which the accessor properties are added is given in the register prop. The property label of the accessor functions is set to the context. putGetterSetter calls putIndirect internally and sets the getter/setter property of the object with the specified value.

**getter-setter**

\[
\text{getter-setter:} \quad \pi', \theta', \sigma, \rho' \sim \pi', \theta', \sigma, \rho' \\
\]

\[
\begin{align*}
\pi &= \text{“put-getter-setter base: r prop:id getter: r setter: r”}, \\
\ast \notin \Gamma(\sigma(\text{getter})), \ast \notin \Gamma(\sigma(\text{setter})), \\
\theta' &:= \text{putGetterSetter}(\Gamma(\rho), \sigma, \text{base, prop, } \sigma(\text{getter}), \sigma(\text{setter})), \\
\pi' &:= \text{Succ}(\sigma.\text{CFG}, \iota), \rho' := \text{isIPD}(\pi', \rho) \\
\end{align*}
\]

**get-pnames**

creates a property name list for object in register base and puts it in dst, initializing i and size for iteration through the list, after the deferred NSU check. If base is undefined or null, it jumps to breakTarget. It is a branching instruction and pushes the label with join of all the property labels and the structure label of the object along with the IPD on the pc-stack. If the IPD of the instruction is SEN or the same as the top of the pc-stack, then we just join the label on top of the pc-stack with the context label determined above.

**get-pnames**

\[
\text{get-pnames:} \quad \iota, \theta, \sigma, \rho \sim \iota', \theta', \sigma', \rho' \\
\]

\[
\begin{align*}
\iota &= \text{“get-pnames dst: r base: i r size: r breakTarget: offset”} \\
\Gamma(\sigma(\text{base})) &\neq \ast \quad (v_1, v_2, v_3, \iota') := \text{getPropNames}(\sigma, \iota, \text{base, i, size, breakTarget}) \\
\mathcal{L}_n &:= \Gamma(\sigma(\text{base})) \cup \Gamma(\rho) \quad \forall i := \mathcal{L}(v_n), n := 1, 2, 3 \\
(\Gamma(\sigma(\text{dst})) \geq \Gamma(\rho)) &\Rightarrow (\mathcal{L}_1 := \mathcal{L}_1) \circ (\mathcal{L}_1 := \ast) \\
(\Gamma(\sigma(\text{size})) \geq \Gamma(\rho)) &\Rightarrow (\mathcal{L}_2 := \mathcal{L}_2) \circ (\mathcal{L}_2 := \ast) \\
(\Gamma(\sigma(\text{size})) \geq \Gamma(\rho)) &\Rightarrow (\mathcal{L}_3 := \mathcal{L}_3) \circ (\mathcal{L}_3 := \ast) \\
\sigma'' &:= \sigma''[\mathcal{T}(\mathcal{L}(\sigma(\text{size})) := \mathcal{V}_3)] \\
\rho'' &:= \rho.\text{push}(\mathcal{L}, \text{IPD}(\iota), \text{CF}(\iota), \text{false}) \\
\end{align*}
\]

**next-pname**

copies the next name from the property name list created by get-pnames in iter to dst after deferred NSU check, and jumps to target. If there are no names left, it continues with the next instruction. Although, it behaves as a branching instruction, the context pertaining to this opcode is already pushed in get-pnames. Also, the IPD corresponding to this instruction, is the same as the one determined by get-pnames. Thus, we do not push on the pc-stack in this instruction.
resolve searches for the property in the scope chain and writes it into dst register, if found. The label of the property written in dst is a join of the context label, all the nodes (structure label of the object contained in it) traversed in the scope chain and the label associated with the pointers in the chain until the node (object) where the property is found. If the initial label of the value contained in dst was lower than the context label, then the label of the value in dst is joined with *. In case the property is not found, the instruction throws an exception (similar to throw, as described later).

\[ \ell = \text{"resolve-skip dst:r prop:id skip:n"} \]
\[ v := \text{resolveInScWithSkip}(\Gamma(\ell), \sigma, \text{sc}, \text{prop}) \]
\[ L := \Gamma(v) \sqcup \Gamma(\ell) \quad V := T(v) \quad (\Gamma(\sigma(\text{dst})) \geq \Gamma(\ell)) \Rightarrow (L := L) \circ (L := *) \]
\[ resolve-skip:\]
\[ \ell, \theta, \sigma, \rho \rightsquigarrow \ell', \theta, \sigma', \rho' \]

resolve-skip looks up the property named by prop in the scope chain similar to resolve, but it skips the top skip levels and writes the result to register dst. If the property is not found, it also raises an exception and behaves similarly to resolve.

\[ \ell = \text{"resolve-global dst:r prop:id"} \]
\[ v := \text{resolveGlobal}(\Gamma(\ell), \sigma, \text{prop}) \]
\[ L := \Gamma(v) \sqcup \Gamma(\ell) \quad V := T(v) \quad (\Gamma(\sigma(\text{dst})) \geq \Gamma(\ell)) \Rightarrow (L := L) \circ (L := *) \]
\[ resolve-global:\]
\[ \ell, \theta, \sigma, \rho \rightsquigarrow \ell', \theta, \sigma', \rho' \]

resolve-global looks up the property named by prop in the global object. If the structure of the global object matches the one passed here, it looks into the global object. Else, it falls back to perform a full resolve.

\[ \ell = \text{"resolve-base dst:r prop:id isStrict:bool"} \]
\[ v := \text{resolveBase}(\Gamma(\ell), \sigma, \text{sc}, \text{prop}, \text{isStrict}) \]
\[ L := \Gamma(v) \sqcup \Gamma(\ell) \quad V := T(v) \quad (\Gamma(\sigma(\text{dst})) \geq \Gamma(\ell)) \Rightarrow (L := L) \circ (L := *) \]
\[ resolve-base:\]
\[ \ell, \theta, \sigma, \rho \rightsquigarrow \ell', \theta, \sigma', \rho' \]

resolve-base looks up the property named by prop in the scope chain similar to resolve but writes the object to register dst. If the property is not found and isStrict is false, the global object is stored in dst.
\( \ell = \text{"resolve-with-base bDst:r pDst:r prop:id"} \)

\[
(bdst, pdst) \coloneqq \text{resolveBaseAndProperty}(\Gamma(\ell'), \sigma, \text{baseDst}, \text{propDst}, \text{prop})
\]

\[
L1 \coloneqq \Gamma(\sigma(bdst)) \sqcup \Gamma(\ell') \quad V1 \coloneqq \mathcal{T}(\sigma(bdst))
\]

\[
L2 \coloneqq \Gamma(\sigma(pdst)) \sqcup \Gamma(\ell') \quad V2 \coloneqq \mathcal{T}(\sigma(pdst))
\]

\[
(\Gamma(\sigma(bdst)) \geq \Gamma(\ell')) \Rightarrow (L1 \coloneqq L1) \circ (L2 \coloneqq *)
\]

\[
(\Gamma(\sigma(pDst)) \geq \Gamma(\ell')) \Rightarrow (L2 \coloneqq L2) \circ (L2 \coloneqq *)
\]

\[
\sigma'' \coloneqq \sigma[\mathcal{T}(\Gamma(\sigma(bdst))) \sqcup V1] \quad \sigma' \coloneqq \sigma[\mathcal{T}(\Gamma(\sigma(pDst))) \sqcup V2]
\]

\[
i' \coloneqq \text{Succ}(\sigma', \text{CFG}, i) \quad \rho' \coloneqq \text{isIPD}(i', \rho, \sigma')
\]

\( \text{resolve-with-base} \) looks up the property named by \( \text{prop} \) in the scope chain similar to \( \text{resolve-base} \) and writes the object to register \( \text{bDst} \). It also, writes the property to \( \text{pDst} \). If the property is not found it raises an exception like resolve.

\( \ell = \text{"get-scoped-var dst:r index:n skip:n"} \)

\[
v \coloneqq \text{getScopedVar}(\Gamma(\ell'), \sigma, \theta, \text{index}, \text{skip}) \quad L \coloneqq \Gamma(v) \sqcup \Gamma(\ell') \quad V \coloneqq \mathcal{T}(v)
\]

\[
(\Gamma(\sigma(\text{dst})) \geq \Gamma(\ell')) \Rightarrow (L \coloneqq L) \circ (L \coloneqq *)
\]

\[
\sigma' \coloneqq \sigma[\mathcal{T}(\Gamma(\sigma(\text{dst}))) \sqcup L] \quad i' \coloneqq \text{Succ}(\sigma', \text{CFG}, i) \quad \rho' \coloneqq \text{isIPD}(i', \rho, \sigma')
\]

\( \text{get-scoped-var} \) loads the contents of the \( \text{index} \) local from the scope chain skipping \( \text{skip} \) nodes and places it in \( \text{dst} \), after deferred NSU. The label of the value in \( \text{dst} \) includes the join of the current context along with all the structure label of objects in the skipped nodes.

\( \ell = \text{"put-scoped-var index:n skip:n value:r"} \)

\[
\Gamma(\sigma(\text{value})) \neq * \quad \sigma' \coloneqq \text{putScopedVar}(\Gamma(\ell'), \sigma, \theta, \text{index}, \text{skip}, \text{value})
\]

\[
i' \coloneqq \text{Succ}(\sigma', \text{CFG}, i) \quad \rho' \coloneqq \text{isIPD}(i', \rho, \sigma')
\]

\( \text{put-scoped-var} \) puts the contents of the \( \text{value} \) in the \( \text{index} \) local in the scope chain skipping \( \text{skip} \) nodes. The label of the value includes the join of the current context along with the structure label of all the objects in the skipped nodes.

\( \ell = \text{"push-scope scope:r"} \)

\[
\sigma' \coloneqq \text{pushScope}(\Gamma(\ell'), \sigma, \text{scope}) \quad i' \coloneqq \text{Succ}(\sigma', \text{CFG}, i) \quad \rho' \coloneqq \text{isIPD}(i', \rho, \sigma')
\]

\( \text{push-scope} \) converts \( \text{scope} \) to object and pushes it onto the top of the current scope chain. The contents of the register \( \text{scope} \) are replaced by the created object. The scope chain pointer label is set to the context.

\( \ell = \text{"pop-scope"} \)

\[
\sigma' \coloneqq \text{popScope}(\Gamma(\ell'), \sigma) \quad i' \coloneqq \text{Succ}(\sigma', \text{CFG}, i) \quad \rho' \coloneqq \text{isIPD}(i', \rho, \sigma')
\]

\( \text{pop-scope} \) removes the top item from the current scope chain if the scope chain pointer label is greater than or equal to the context.

\( \ell = \text{"jmp-scope count:n target:n"} \)

\[
\sigma' \coloneqq \text{jmpScope}(\Gamma(\ell'), \sigma, \text{count}) \quad i' \coloneqq \text{Succ}(\sigma', \text{CFG}, i) \quad \rho' \coloneqq \text{isIPD}(i', \rho, \sigma')
\]

\( \text{jmp-scope} \)
jmp-scope removes the top count items from the current scope chain if the scope chain pointer label is greater than or equal to the context. It then jumps to offset specified by target.

\[
\epsilon = \text{"throw ex:"} \quad \text{excValue} := T(!\sigma(ex))
\]

\[
\text{throw: } \quad (\sigma', \epsilon') := \text{throwException}(\sigma, \epsilon) \quad \rho' := \text{isIPD}(\epsilon', \rho, \sigma')
\]

\[
\epsilon, \theta, \sigma, \rho \rightsquigarrow \epsilon', \theta, \sigma', \rho'
\]

throw throws an exception and points to the exception handler as the next instruction to be executed, if any. The exception handler might be in the same function or in an earlier function. If it is not present, the program terminates. If it has an exception handler, it has an edge to the synthetic exit node. Apart from this, throwException pops the call-frames from the call-stack until it reaches the call-frame containing the exception handler. It writes the exception value to a local interpreter variable (excValue), which is then read by catch.

\[
\epsilon = \text{"catch ex:"}
\]

\[
\mathcal{L} := \Gamma(!\text{excValue}) \sqcup \Gamma(!\rho) \quad (\Gamma(!\sigma(ex)) \geq \Gamma(!\rho)) \Rightarrow (\mathcal{L} := \mathcal{L}) \circ (\mathcal{L} := \ast)
\]

\[
\text{excValue} := \text{empty}
\]

\[
\epsilon', \theta', \sigma', \rho' := \text{catch}(!\sigma'.CFG, \epsilon)
\]

\[
\epsilon, \theta, \sigma, \rho \rightsquigarrow \epsilon', \theta, \sigma', \rho'
\]

catch catches the exception thrown by an instruction whose handler corresponds to the catch block. It reads the exception value from excValue and writes into the register ex. If the label of the register is less than the context, a \(\ast\) is joined with the label. It then makes the excValue empty and proceeds to execute the first instruction in the catch block.

8.4 Proofs and Results

The fields in a frame of the pc-stack are denoted by the following symbols: \(\!\rho.ipd\) represents the IPD field in the top frame of the pc-stack, \(\Gamma(!\rho)\) returns the label field in the top frame of the pc-stack, and \(\!\rho.C\) returns the call-frame field in the top frame of the pc-stack.

In the definitions and proofs that follow, we assume that the level of attacker is \(L\) in the three-element lattice presented earlier, i.e., in the equivalence relation \(\sim^\beta\), \(\ell = L\). The level of the attacker is omitted for clarity purposes from definitions and proofs.

**Definition 1 (Partial bijection)** A partial bijection \(\beta\) is a binary relation on heap locations satisfying the following properties: (1) if \((a, b) \in \beta\) and \((a, b') \in \beta\), then \(b = b'\), and (2) if \((a, b) \in \beta\) and \((a', b) \in \beta\), then \(a = a'\).

Using partial bijections, we define equivalence of values, labeled values and objects.

**Definition 2 (Value equivalence)** Two values \(r_1\) and \(r_2\) are equivalent up to \(\beta\), written \(r_1 \sim^\beta r_2\) if either (1) \(r_1 = a, r_2 = b\) and \((a, b) \in \beta\), or (2) \(r_1 = r_1 = v\) where \(v\) is some primitive value.
Definition 3 (Labeled value equivalence) Two labeled values $v_1 = r_1^\ell_1$ and $v_2 = r_2^\ell_2$ are equivalent up to $\beta$, written $v_1 \sim^\beta v_2$ if one of the following holds: (1) $\ell_1 = \ast$ or $\ell_2 = \ast$, or (2) $\ell_1 = \ell_2 = H$, or (3) $\ell_1 = \ell_2 = L$ and $r_1 \sim^\beta r_2$.

The first clause of the above definition is standard for the permissive-upgrade check. It equates a partially leaked value to every other labeled value.

Objects are formally denoted as $N = (\{p_i \mapsto \{v_i, flags_i\}\})_{i=0}^n$, ____proto____ $\mapsto \ast \ell_s, \ell_s)$. Here $p_i$s correspond to the property name, $v_i$s are their respective values and $flags_i$ represent the writable, enumerable and configurable flags as described in the PropertyDescriptor structure in the cpp model above. As the current model does not allow modification of the $flags$, they are always set to true. Thus, we do not need to account for the $flags_i$ in the equivalence definition below. ____proto____ represents a labelled pointer to the object’s prototype.

Definition 4 (Object equivalence) For ordinary objects $N = (\{p_i \mapsto \{v_i, flags_i\}\})_{i=0}^n$, ____proto____ $\mapsto \ast \ell_s, \ell_s$ and $N' = (\{p'_i \mapsto \{v'_i, flags'_i\}\})_{i=0}^n$, ____proto____ $\mapsto \ast \ell'_s, \ell'_s$, we say $N \sim^\beta N'$ iff either $\ell_s = \ell'_s = H$ or the following hold: (1) $\ell_s = \ell'_s = L$, (2) $[p_0, \ldots, p_n] = [p'_0, \ldots, p'_n]$ (in particular, $n = m$), (3) $\forall i. v_i \sim^\beta v'_i$, and (4) $a \sim^\beta a'$. For function objects $F = (N, f, \Sigma)$ and $F' = (N', f', \Sigma')$, we say $F \sim^\beta F'$ iff either $N, \ell_s = N, \ell'_s = H$ or $N \sim^\beta N'$, $f = \beta f'$ and $\Sigma \sim^\beta \Sigma'$.

The equality $f = \beta f'$ of nodes $f, f'$ in CFGs means that the portions of the CFGs reachable from $f, f'$ are equal modulo renaming of operands to bytecodes under $\beta$. Equivalence $\Sigma \sim^\beta \Sigma'$ of scope chains is defined below. Because we do not allow $\ast$ to flow into heaps, we do not need corresponding clauses in the definition of object equivalence.

Definition 5 (Heap equivalence) For two heaps $\theta_1, \theta_2$, we say that $\theta_1 \sim^\beta \theta_2$ iff $\forall (a, b) \in \beta$, $\theta_1(a) \sim^\beta \theta_2(b)$.

Unlike objects, we allow $\ast$ to permeate scope chains, so our definition of scope chain equivalence must account for it. Scope chains are denoted as $\Sigma$. A scope-chain node contains a label $\ell$ along with an object $S$ (either JSActivation or JSObject) represented as $(S, \ell)$.

Definition 6 (Scope chain equivalence) For two scope chain nodes $S, S'$, we say that $S \sim^\beta S'$ if one of the following holds: (1) $S = O, S' = O'$ and $O \sim^\beta O'$, or (2) $S = v_0: \ldots: v_n, S' = v'_0: \ldots: v'_n$ and $\forall i. v_i \sim^\beta v'_i$.

Equivalence of two scope chains $\Sigma, \Sigma'$ is defined by the following rules. (1) nil $\sim^\beta$ nil (2) (nil $\sim^\beta (S, \ell)$) if $\ell = H$ or $\ell = \ast$ (3) ((S, $\ell$) $\sim^\beta$ nil) if $\ell = H$ or $\ell = \ast$ and (4) ((S, $\ell$), $\Sigma$) $\sim^\beta$ ((S', $\ell'$), $\Sigma'$) if one of the following holds: (a) $\ell = \ast$ or $\ell' = \ast$, (b) $\ell = \ell' = H$, or (c) $\ell = \ell' = L, S \sim^\beta S'$ and $\Sigma \sim^\beta \Sigma'$.

Definition 7 (Call-frame equivalence) For two call frames $\mu_1, \mu_2$, we say $\mu_1 \sim^\beta \mu_2$ iff (1) $\#\text{Registers}(\mu_1) = \#\text{Registers}(\mu_2)$, (2) $\forall i. \mu_1.\text{Registers}[i] \sim^\beta \mu_2.\text{Registers}[i]$, (3) $\mu_1.\text{CFG} = \beta \mu_2.\text{CFG}$, (4) $\mu_1.\text{Scopechain} \sim^\beta \mu_2.\text{Scopechain}$.
Definition 10 (State equivalence) σ \sim \langle σ and Definition 9 (Call-stack equivalence) are equal, except for the call-frame µ.

Note that a register is simply a labeled value in our semantics, so clause (2) above is well-defined.

Definition 8 (pc-stack equivalence) For two pc-stacks ρ₁, ρ₂, we say ρ₁ \sim ρ₂ iff the corresponding nodes of ρ₁ and ρ₂ having label L are equal, except for the call-frame (C) field.

In proofs that follow, two pc-stack nodes are equal if their respective fields are equal, except for the call-frame (C) field.

Definition 9 (Call-stack equivalence) Given ρ₁ \sim ρ₂, suppose:

1. e₁ is the lowest H-labelled node in ρ₁
2. e₂ is the lowest H-labelled node in ρ₂
3. μ₁ is the node of σ₁ pointed to by e₁
4. μ₂ is the node of σ₂ pointed to by e₂
5. σ₁' is prefix of σ₁ up to and including μ₁ or
   if Γ(!ρ₁) = L or ρ₁ is empty, σ₁' = σ₁
6. σ₂' is prefix of σ₂ up to and including μ₂ or
   if Γ(!ρ₂) = L or ρ₂ is empty, σ₂' = σ₂

then σ₁ \sim^β_{ρ₁, ρ₂} σ₂, iff (1) |σ₁'| = |σ₂'|, and (2) ∀i ≤ |σ₁'|, (σ₁'[i] \sim^β σ₂'[i]).

Definition 10 (State equivalence) Two states s₁ = ⟨ι₁, θ₁, σ₁, ρ₁⟩ and s₂ = ⟨ι₂, θ₂, σ₂, ρ₂⟩ are equivalent, written as s₁ \sim s₂, iff ι₁ = ι₂, ρ₁ \sim ρ₂, θ₁ \sim θ₂, and σ₁ \sim^β_{ρ₁, ρ₂} σ₂.

Lemma 1 (Confinement Lemma) If ⟨ι, θ, σ, ρ⟩ \sim ⟨ι', θ', σ', ρ'⟩ and Γ(!ρ) = H, then ρ \sim ρ', σ \sim^β_{ρ, ρ'} σ' and θ \sim^β θ' where β = {(a, a) | a ∈ θ}

Proof. As Γ(!ρ) = H, the L labelled nodes in the pc-stack will remain unchanged. Branching instructions pushing a new node would have label H due to monotonicity of pc-stack. Even if ι' is the IPD corresponding to the !ρ.ipd, it would only pop the H labelled node. Thus, the L labelled nodes will remain unchanged. Hence, ρ \sim ρ'.

We assume that the !ρ is the first node labelled H in the context stack. For, other higher labelled nodes above the first node labelled H in the pc-stack, the call-frames corresponding to the nodes having L label in the pc-stack remain the same. Hence, σ \sim^β_{ρ, ρ'} σ'.

By case analysis on the instruction type:

1. prim:
   (a) If Γ(!σ(dst)) ≥ Γ(!ρ), then Γ(!σ(dst)) = H.
       By premise of prim, Γ(!σ'(dst)) = H. By Definition 3 !σ(dst) \sim^β !σ'(dst).
10. \( \text{call} \)

11. \( \text{call-put-result} \)

12. \( \text{construct} \)

13. \( \text{jfalse} \)

14. \( \text{enter} \)

15. \( \text{create-activation} \)

16. \( \text{end} \)

17. \( \text{enter} \)

18. \( \text{call} \)

19. \( \text{create-activation} \)

20. \( \text{call-eval} \)

21. \( \text{loop-if-less} \)

22. \( \text{typeof} \)

23. \( \text{instanceof} \)

24. \( \text{ret} \)

25. \( \text{end} \)

26. \( \text{call} \)

27. \( \text{call-eval} \)

28. \( \text{typeof} \)

29. \( \text{instanceof} \)

30. \( \text{ret} \)

31. \( \text{call} \)

32. \( \text{create-arguments} \)

33. \( \text{new-func} \)

34. \( \text{create-activation} \)

35. \( \text{construct} \)

(b) If \( \Gamma(!\sigma(dst)) < \Gamma(!\rho) \), then \( \Gamma(!\sigma'(dst)) \) will contain a \( \star \) and by Definition 3, \( \sim_{\rho,\rho'} \). Only \( dst \) changes in the call-frame, so by Definition 7, \( \sim_{\rho,\rho'} \). Also, other call-frames remain unchanged. By Definition 9, \( \sim_{\rho,\rho'} \). \( \theta = \theta' \), thus, \( \theta \sim_{\rho,\rho'} \theta' \).

2. \( \text{mov} \): Similar to prim.

3. \( \text{jfalse} \): \( \sigma = \sigma' \) and \( \theta = \theta' \), so, \( \sim_{\rho,\rho'} \theta' \) and \( \theta \sim_{\rho,\rho'} \theta' \).

4. \( \text{loop-if-less} \): Similar to jfalse.

5. \( \text{typeof} \): Similar to prim.

6. \( \text{instanceof} \): Similar to prim.

7. \( \text{ret} \): If \( !\sigma.getter = \text{false} \) then only \( !\sigma \) is popped, the call-frames until \( (!\rho.C) \) are unchanged. When \( !\sigma.getter = \text{true} \), then it sets \( !\sigma'(!\sigma.dReg) \) with \( !\sigma(res) \).

Now, let \( \sigma_1 \) is the prefix of \( \sigma \) such that \( !\rho.C = !\rho!\sigma_1 \). If \( !\sigma' \neq !\sigma_1 \) then changes in \( !\sigma' \) does not effect the callframe equivalence and if \( !\sigma' \in !\sigma_1 \) then \( \Gamma(!\sigma'(!\sigma.dReg)) = \star \) (when \( \Gamma(!\sigma_1(!\sigma.dReg)) \) \( = L \) or \( \Gamma(!\sigma_1(!\sigma.dReg)) \) \( = \star \)) and \( \Gamma(!\sigma'(!\sigma.dReg)) \) \( = H \) (when \( \Gamma(!\sigma_1(!\sigma.dReg)) \) \( = H \)), each of the cases give \( !\sigma_1(!\sigma.dReg) \) \( \sim_{\rho,\rho'} !\sigma'_1(!\sigma.dReg) \) from Definition 3. So, \( \sim_{\rho,\rho'} \theta' \).

8. \( \text{end} \): The confinement lemma does not apply.

9. \( \text{call} \): If it pushes on top of pc-stack, \( !\rho.C \) is the lowest \( H \)-labelled node in \( \rho' \). If it joins the label with \( !\rho \), the \( L \) labelled nodes remain unchanged and the \( !\rho.C = !\rho'.C \). All the call-frames until \( !\rho.C \) remain unchanged. So, by Definition 9, \( \sim_{\rho,\rho'} \theta' \).

10. \( \text{call-eval} \): If it is a user-defined eval, it is similar to call.

In strict mode, it pushes a node on scope-chain with label \( H \) if \( \Gamma(!\sigma',\Sigma) = H \), else labels it \( \star \). In non-strict mode, it does not push a node on the scope-chain. \( !\sigma \) remains equivalent with corresponding call-frame in \( \sigma' \) by Definition 5. As other \( L \) call-frames are unchanged, by Definition 9, \( \sim_{\rho,\rho'} \theta' \).

11. \( \text{call-put-result} \): Similar to prim.

12. \( \text{call-eval} \): If it is a user-defined eval, it is similar to call.

13. \( \text{create-arguments} \): Over the initial \( \beta \), by Definition 5, \( \theta \sim_{\rho,\rho'} \theta' \). If the argument object is created at \( x \), then \( \beta = (x, x) \cup \beta \) after the step is taken.

14. \( \text{new-func} \): Over the initial \( \beta \), by Definition 5, \( \theta \sim_{\rho,\rho'} \theta' \). If the function object is created at \( x \), then \( \beta = (x, x) \cup \beta \) after the step is taken.

15. \( \text{create-activation} \): Over the initial \( \beta \), by Definition 5, \( \theta \sim_{\rho,\rho'} \theta' \). If the argument objects is created at \( x \), then \( \beta = (x, x) \cup \beta \) after the step is taken.

It puts the object in \( dst \) with label \( H \) or \( \star \), depending on \( dst \) value’s initial label. Also, pushes a node containing the object in the scope chain with a \( \star \), if \( \Gamma(!\sigma,\Sigma) = L \lor \star \), or with label \( H \), if \( \Gamma(!\sigma,\Sigma) = H \) or \( (!\sigma,\Sigma) = \text{nil} \). Thus, \( !\sigma,\Sigma \sim_{\rho,\rho'} !\sigma',\Sigma \) by Definition 6. By Definition 7, \( !\sigma \sim_{\rho,\rho'} !\sigma' \). Other call-frames are unchanged, so \( \sim_{\rho,\rho'} \theta' \).

16. \( \text{construct} \): Similar to call.
17. create-this: Similar to create-arguments.
18. new-object: Over the initial $\beta$, by Definition 5 \( \theta \sim^\beta \theta' \). If the new object is created at $x$, then $\beta = (x, x) \cup \beta$ after the step is taken.
\[ \sigma \sim_{\rho, \rho'}^\beta \sigma' \] (Similar to prim).
19. get-by-id: Similar to mov when the property is a data property. If the property is an accessor property then getter is invoked and if the invocation of getter pushes an entry on top of pc-stack, $!p.C$ remains the lowest $H$-labelled node in $\rho'$. If it joins the label with $lp$, the L labelled nodes remain unchanged and the $!p.C = !p'.C$. All the call-frames until $!p.C$ remain unchanged. So, by Definition 9, $\sigma \sim_{\rho, \rho'}^\beta \sigma'$. \[ \theta = \theta', \text{ so } \theta \sim^\beta \theta' \] .
20. put-by-id: Sets the property of the object base object to the value with label $H$ if the structure label of the object $\ell_s = H$. Thus, the object remains low-equivalent by Definition 4. Thus, $\theta \sim^\beta \theta'$ by Definition 5. Also, $\sigma = \sigma'$, so, $\sigma \sim_{\rho, \rho'}^\beta \sigma'$.
21. del-by-id: Deletes the property if structure label of object, $\ell_s = H$. Thus, the object remains low-equivalent by Definition 4. By Definition 5, $\theta \sim^\beta \theta'$.
22. setter-setter: Sets accessor property of the object base object with $\Gamma$(getter) and $\Gamma$(setter) and label $H$ if the structure label of the object $\ell_s = H$. Thus, the object remains low-equivalent by Definition 4. Thus, $\theta \sim^\beta \theta'$ by Definition 5. Also, $\sigma = \sigma'$, so, $\sigma \sim_{\rho, \rho'}^\beta \sigma'$.
23. get-pnames: Similar to mov and jfalse.
24. next-pname: Similar to mov.
25. resolve: If the property exists, it is similar to mov. If it does not, it is similar to throw.
26. resolve-skip: Similar to resolve.
27. resolve-global: Similar to resolve.
28. resolve-base: Similar to resolve.
29. resolve-with-base: Similar to resolve.
30. get-scoped-var: Similar to mov.
31. put-scoped-var: Writes the value in the indexth register in skipth node. If $\Gamma(\sigma(index)) = H$, then, $\Gamma(\sigma'(index)) = H$. Else if $\Gamma(\sigma(index)) = L$, then, $\Gamma(\sigma'(index)) = \ast$. Other call-frames are unchanged. Thus, $\sigma \sim_{\rho, \rho'}^\beta \sigma'$ by Definition 7 and 9.
\[ \theta = \theta', \text{ so } \theta \sim^\beta \theta' \]
32. push-scope: Pushes node on scope-chain with label $H$ if $\Gamma(!\sigma.\Sigma) = H$ or $(!\sigma.\Sigma) = nil$. Else, assigns a $\ast$ as the label. Thus, $!\sigma.\Sigma \sim^\beta !\sigma'.\Sigma$. Registers remain unchanged. By Definition 7, $!\sigma \sim^\beta !\sigma'$. Other call-frames are unchanged, so by Definition 9, $\sigma \sim_{\rho, \rho'}^\beta \sigma'$. $\theta = \theta'$, so $\theta \sim^\beta \theta'$.
33. pop-scope: Pops the node from the scope-chain if $\Gamma(!\sigma.\Sigma) = H \lor \ast$. Registers remain unchanged. By Definition 7, $!\sigma \sim^\beta !\sigma'$. Other call-frames are unchanged, so by Definition 9, $\sigma \sim_{\rho, \rho'}^\beta \sigma'$. $\theta = \theta'$, so $\theta \sim^\beta \theta'$.
34. jmp-scope: Similar to pop-scope.
35. throw: Pops the call-frames until the handler is reached, i.e., until $(!p.C)$. The property of IPD ensures that $!\sigma' = (!p.C)$. Either $!p'.ipd = !p.ipd$ or $\nu' = !p.ipd$. Thus, $!p.C$ is $!\sigma'$. This call-frame and the ones below remain
Corollary 2 If \( \langle \ell_0, \theta_0, \sigma_0, \rho_0 \rangle \sim^n \langle \ell_n, \theta_n, \sigma_n, \rho_n \rangle \) and \( \forall (0 \leq i \leq n). \Gamma(\ell_i') = H \), then \( \rho_0 \sim \rho_n \), and \( \sigma_0 \sim^{\beta} \rho_0, \rho_n \sigma_n \).

Proof. To prove: \( \rho_0 \sim \rho_n \).

Proof by induction on \( n \).

Basis: \( \rho_0 \sim \rho_0 \)

IH: \( \rho_0 \sim \rho_{n-1} \)

From Definition \( L \) labelled nodes of \( \rho_0 \) and \( \rho_{n-1} \) are equal. From Lemma \( L \rho_{n-1} \sim \rho_n \) so, \( L \) labelled nodes of \( \rho_{n-1} \) and \( \rho_n \) are equal. Thus, \( L \) labelled nodes of \( \rho_0 \) and \( \rho_n \) are equal and by Definition \( \rho_0 \sim \rho_n \).

To prove: \( \sigma_0 \sim^{\beta} \rho_0, \rho_n \sigma_n \).

Basis: \( \sigma_0 \sim^{\beta} \rho_0, \rho_0 \sigma_0 \)

IH: \( \sigma_0 \sim^{\beta} \rho_0, \rho_n \sigma_{n-1} \).

From Lemma \( \sigma_{n-1} \sim^{\beta} \rho_{n-1}, \rho_n \sigma_n \). As \( \forall (0 \leq i \leq n). \Gamma(\ell_i) = H \), the lowest \( H \)-labelled node is the same (pc-stack grows monotonically) in \( \rho_0, \rho_{n-1}, \rho_n \). Let the call-frames pointed to by lowest \( H \)-labelled node be \( C_0, C_{n-1}, C_n \) with call-stack size until the call-frames \( k \) (from Definition \( \theta \) size of the prefix is same and by transitivity of equality it is the same for all the three cases).

\( \forall \mu_0 \in \sigma_0, \mu_{n-1} \in \sigma_{n-1}, \mu_n \in \sigma_n \) until \( C_0, C_{n-1}, C_n \) respectively with sizes \( k \), the following conditions hold:

1. \( \forall (1 \leq i \leq k).((\mu_0[i], \#\text{Registers}) = (\mu_{n-1}[i], \#\text{Registers})) \) and \( \forall (1 \leq i \leq k).((\mu_{n-1}[i], \#\text{Registers}) = (\mu_n[i], \#\text{Registers})) \).

Thus, \( \forall (1 \leq i \leq k).((\mu_0[i], \#\text{Registers}) = (\mu_n[i], \#\text{Registers})) \).

2. As the number of registers is the same, given by \( r \),

\( \forall (1 \leq i \leq k).\forall r((\mu_0[r].\text{Registers}[r]) \sim^\beta (\mu_{n-1}[r].\text{Registers}[r])[r]) \) and

\( \forall (1 \leq i \leq k).\forall r((\mu_{n-1}[r].\text{Registers}[r]) \sim^\beta (\mu_n[r].\text{Registers}[r])[r]) \).

Let \( v_0^n, v_{n-1}^n \) and \( v_n^n \) represents the values in the registers for \( \sigma_0, \sigma_{n-1} \) and \( \sigma_n \) respectively. Then from Definition \( \theta \)

(a) \( \ell_0 = \ell_{n-1} = H \): In this case \( \ell_n = H \) and \( v_0^n \sim^\beta v_{n-1}^n \) (from Lemma \( \theta \) and Definition \( \theta \)).

(b) \( \ell_0 = \ell_{n-1} = L \) and \( v_0 = v_{n-1} \): In this case either:

i. \( \ell_n = * \)

ii. \( \ell_n = L \) and \( v_n = v_n \): In this case, the value remains unchanged.

Thus, from Definition \( \theta \)

\( v_0^n \sim^\beta v_{n-1}^n \).

(c) \( \ell_0 = * \vee \ell_{n-1} = * \): Now the following cases arise:

i. \( \ell_0 = * : v_0^n \sim^\beta v_{n-1}^n \).

ii. \( \ell_{n-1} = * \): By Lemma \( \theta \) \( \ell_n = * \). Thus, \( v_0^n \sim^\beta v_{n-1}^n \).

3. \( \forall (1 \leq i \leq k).((\mu_0[i].\text{CFG}) = (\mu_{n-1}[i].\text{CFG})) \) and \( \forall (1 \leq i \leq k).((\mu_{n-1}[i].\text{CFG}) = (\mu_n[i].\text{CFG})) \).

Thus, \( \forall (1 \leq i \leq k).((\mu_0[i].\text{CFG}) = (\mu_n[i].\text{CFG})) \).
4. \( \forall (1 \leq i \leq k).((\mu_0[i], \Sigma) \sim^\beta (\mu_{n-1}[i], \Sigma)) \) and \( \forall (1 \leq i \leq k).((\mu_{n-1}[i], \Sigma) \sim^\beta (\mu_n[i], \Sigma)) \).

From Definition 6

(a) If \( \text{nil}_0 \) and \( \text{nil}_{n-1} \) be the two scope chains, then due to confinement (Lemma 1) \( \mu_n[i], \Sigma = \text{nil} \) or \( \mu_n[i], \Sigma = (S, \ell_n) \), where \( \ell_n = H \). In either case \( \forall (1 \leq i \leq k).((\mu_0[i], \Sigma) \sim^\beta (\mu_n[i], \Sigma)) \) from Definition 6.

(b) If \( ((S_0, \ell_0) : \Sigma_0), ((S_{n-1}, \ell_{n-1}) : \Sigma_{n-1}) \) and \( ((S_n, \ell_n) : \Sigma_n) \) be the three scope-chains, then for \((S_0, \ell_0) : \Sigma_0 \) and \((S_{n-1}, \ell_{n-1}) : \Sigma_{n-1}) \) one of the following holds:

i. \( \ell_0 \neq * \lor \ell_{n-1} = * \): Due to confinement (Lemma 1) and Definition 6
   \( \ell_n = * \).

ii. \( \ell_0 = \ell_{n-1} = H \): Due to confinement (Lemma 1) and Definition 6
   \( \ell_n = H \).

iii. \( \ell_0 = \ell_{n-1} = L \land S_0 \sim^\beta S_{n-1} \land \Sigma_0 \sim^\beta \Sigma_{n-1} \): Due to confinement (Lemma 1) either one should hold:
   A. \( \ell_n = * \): By Definition 6
   B. \( \ell_n = L \land S_{n-1} \sim^\beta S_n \land \Sigma_{n-1} \sim^\beta \Sigma_n \): No additions to the scope chain.

Thus, \( \forall (1 \leq i \leq k).((\mu_0[i], \Sigma) \sim^\beta (\mu_n[i], \Sigma)) \) from Definition 6.

5. \( \forall (1 \leq i \leq k).((\mu_0[i], \Sigma) \sim^\beta (\mu_n[i], \Sigma)) \) from Definition 6.

Then, \( \forall (1 \leq i \leq k).((\mu_0[i], \Sigma) \sim^\beta (\mu_n[i], \Sigma)) \) and \( \forall (1 \leq i \leq k).((\mu_{n-1}[i], \Sigma) \sim^\beta (\mu_n[i], \Sigma)) \) from Definition 6.

6. \( \forall (1 \leq i \leq k).((\mu_0[i], \Sigma) \sim^\beta (\mu_n[i], \Sigma)) \) from Definition 6.

From Definition 7 and Definition 9 \( \sigma_0 \sim^\beta \rho \sigma_n \).

**Corollary 3** If \( \langle \iota_0, \theta_0, \sigma_0, \rho_n \rangle \sim^\ast \langle \iota_n, \theta_n, \sigma_n, \rho_n \rangle \) and \( \forall (0 \leq i \leq n). \Gamma(!\rho_i) = H \), then \( \theta_0 \sim^\beta \theta_n \).

**Proof.** By induction on \( n \).

**Basis.** \( \theta_0 \sim^\beta \theta_0 \) by Definition 5

**IH.** \( \theta_0 \sim^\beta \theta_{n-1} \).
Thus, if \( \forall (a, b) \in \beta. (\theta_0(a) \sim^\beta \theta_{n-1}(b)) \). From Lemma 1, \( \theta_{n-1} \sim^\beta \theta_n \). Thus, \( \forall (b, c) \in \beta. (\theta_{n-1}(b) \sim^\beta \theta_n(c)) \).

As \((a, b) \in \beta \) and \((b, c) \in \beta \), we have \((a, c) \in \beta \) because \( \beta \) is an identity bijection. Thus, if \( \forall (a, c) \in \beta. (\theta_0(a) \sim^\beta \theta_n(c)) \), then \( \theta_0 \sim^\beta \theta_n \). If \( \theta_0(a) \) and \( \theta_{n-1}(b) \) contain an ordinary object, then for their respective structure labels \( \ell_s \) and \( \ell_s' \), either:

- \( \ell_s = \ell'_s = H \): If \( \ell_s' = H \), then \( \ell''_s = H \) by Definition 4 where \( \ell''_s \) is the structure label of the object in \( \theta_n(c) \). Thus, \( \theta_0(a) \sim^\beta \theta_n(c) \).
- \( \ell_s = \ell'_s = L \): \( [p_0, \ldots, p_n] = [p'_0, \ldots, p'_m] \) \((n = m) \), \( \forall i. v_i \sim^\beta v'_i \), and \( a^{e\nu} \sim^\beta a'^{e\nu} \) for respective properties in \( \theta_0(a) \) and \( \theta_{n-1}(b) \).
- \( \ell_s = \ell'_s = L \): \( [p_0, \ldots, p_n] = [p'_0, \ldots, p'_m] \) \((n = m) \). If \( \forall i. v_i \sim^\beta v'_i \) and \( \forall i. v'_i \sim^\beta v''_i \), then either \( \ell_i = \ell'_i = \ell''_i = H \) or \( \ell_i = \ell'_i = \ell''_i = L \) and \( r_i = r'_i = r''_i = n \).

Also, as \( a^{e\nu} \sim^\beta a'^{e\nu} \) and \( a'^{e\nu} \sim^\beta a''^{e\nu} \), we have \( a^{e\nu} \sim^\beta a''^{e\nu} \). Thus, by Definition 4, \( \theta_0(a) \sim^\beta \theta_n(c) \).

If \( \theta_0(a) \) and \( \theta_{n-1}(b) \) contain a function object, then for their respective structure labels \( \ell_s \) and \( \ell'_s \), either:

- \( \ell_s = \ell'_s = H \): If \( \ell'_s = H \), then \( \ell''_s = H \) by Definition 4 where \( \ell''_s \) is the structure label of the function object in \( \theta_n(c) \). Thus, \( \theta_0(a) \sim^\beta \theta_n(c) \).
- \( \ell_s = \ell'_s = L \): \( \ell''_s = L \) is the structure label of the function object in \( \theta_n(c) \). Thus, \( N \sim^\beta N'' \) from the above result for objects. The CFGs \( f =^\beta f' =^\beta f'' \) and the scope chains \( \Sigma \sim^\beta \Sigma'' \) by Corollary 2. Thus, \( \theta_0(a) \sim^\beta \theta_n(c) \).

Thus, \( \theta_0 \sim^\beta \theta_n \).

**Lemma 2 (Supporting Lemma 1)** Suppose

\[
\langle \iota, \theta_1, \sigma_1, \rho_1 \rangle \sim \langle \iota', \theta'_1, \sigma'_1, \rho'_1 \rangle,
\langle \iota, \theta_2, \sigma_2, \rho_2 \rangle \sim \langle \iota', \theta'_2, \sigma'_2, \rho'_2 \rangle,
\rho_1 \sim \rho_2, \Gamma(!\rho_1) = \Gamma(!\rho_2) = L, \Gamma(!\rho'_1) = \Gamma(!\rho'_2) \text{ and } (\sigma_1 \sim^\beta_{\rho_1, \rho_2} \sigma_2) \land (\theta_1 \sim^\beta \theta_2)
\]
then \( \rho'_1 \sim \rho'_2 \), and \( \exists \beta' : ((\beta' \supseteq \beta) \land (\sigma'_1 \sim^\beta_{\rho'_1, \rho'_2} \sigma'_2) \land (\theta'_1 \sim^\beta \theta'_2)) \).

**Proof.** Every instruction executes \( isIPD \) at the end of the operation. If \( \iota'_i \) is the IPD corresponding to the \(!\rho_i, ipd \), then it pops the first node on the pc-stack. As \( \rho_1 \sim \rho_2 \) and \( \Gamma(!\rho_1) = \Gamma(!\rho_2) \), \( \iota'_i \) would either pop in both the runs or in none. Thus, \( \rho'_i sim \rho'_2 \). For instructions that push (branch), we explain in respective instructions.

Proof by case analysis on the instruction type:

1. **prim:** No new object is created, so \( \beta' = \beta \).

As \( \sigma_1 \sim^\beta_{\rho_1, \rho_2} \sigma_2 \), so \( !\sigma_1 \sim^\beta !\sigma_2 \) and \( !\sigma_1(src_i) = !\sigma_2(src_i) \) for \( i = 1, 2 \). Case analysis on the definition of \( \sim^\beta \) for \( src_i \).
   - If \( (\Gamma(!\sigma_1(src_1)) = \star \lor \Gamma(!\sigma_1(src_2)) = \star \lor \Gamma(!\sigma_2(src_1)) = \star \lor \Gamma(!\sigma_2(src_2)) = \star) \), then \( \Gamma(!\sigma_1(dst)) = \star \lor \Gamma(!\sigma_1(dst)) = \star \). Hence, \( !\sigma_1(dst) \sim^\beta !\sigma_2(dst) \) by Definition 3.
If \( \Gamma(!\sigma_1(src_1)) = \Gamma(!\sigma_1(src_2)) = H \) and \( \Gamma(!\sigma_2(src_1)) = \Gamma(!\sigma_2(src_2)) = H \), \( \Gamma(!\sigma_1(dst)) = \Gamma(!\sigma_2(dst)) = H \). So, \( !\sigma_1(dst) \sim^\beta !\sigma_2(dst) \) by Definition 3.

If \( !\sigma_1(src_1) = !\sigma_2(src_1) \land \Gamma(!\sigma_1(src_2)) = H \land \Gamma(!\sigma_2(src_2)) = H \), then \( \Gamma(!\sigma_1(dst)) = \Gamma(!\sigma_2(dst)) = H \). So, \( !\sigma_1(dst) \sim^\beta !\sigma_2(dst) \) by Definition 3.

Symmetrical reasoning for \( !\sigma_1(src_2) = !\sigma_2(src_2) \land \Gamma(!\sigma_1(src_1)) = H \land \Gamma(!\sigma_2(src_1)) = H \).

\( !\sigma_1(dst) = !\sigma_2(dst) \). So, \( !\sigma_1(dst) \sim^\beta !\sigma_2(dst) \) by Definition 3.

Only \( dst \) changes in both the call-stacks. Thus, by Definition 7. \( !\sigma'_1 \sim^\beta !\sigma'_2 \). Other call-frames in \( \sigma'_1 \) and \( \sigma'_2 \) are unchanged. By Definition 9, \( \sigma'_1 \sim^\beta \rho'_1, \rho'_2 \sigma'_2 \).

\( \theta_1 = \theta'_1 \) and \( \theta_2 = \theta'_2 \), so, \( \theta'_1 \sim^\beta \theta'_2 \).

2. \text{mov}: Similar reasoning as \text{prim} with single source.

3. \text{jfalse}: No new object is created, so \( \beta' = \beta \).

- \( !\sigma_1(cond) = !\sigma_2(cond) \land \Gamma(!\sigma_1(cond)) = \Gamma(!\sigma_2(cond)) = L \): \( L \) is the label to be pushed on \( \rho \).
- \( \Gamma(!\sigma_1(cond)) = \Gamma(!\sigma_2(cond)) = H \): \( H \) is the label to be pushed on \( \rho \).

The IPD of \( i \) would be the same as we have same CFG in both the cases. If the IPD is SEN, then we join the label of \( !\rho_1 \) with the label obtained above, which is the same in both the runs. Thus, \( \Gamma(!\rho_1) = \Gamma(!\rho_2) \). Because \( \rho_1 \sim \rho_2 \), \( \rho'_1 \sim \rho'_2 \).

If the IPD is not SEN, then it is some other node in the same call-frame. Thus the \text{ipd} field is also the same. The \text{H} field is \text{false} in both the cases. Thus, the pushed node is the same in both the cases and hence, \( \rho'_1 \sim \rho'_2 \).

As, \( \Gamma(!\rho'_1) = \Gamma(!\rho'_2) \), either \( \ell'_1 = \ell'_2 = IPD(i) \) or \( \ell'_1 \) and \( \ell'_2 \) may or may not be equal.

\( \sigma'_1 = \sigma_1 \sim^\beta \rho'_1, \rho'_2 \sigma_2 = \sigma'_2 \). \( \theta'_1 = \theta_1 \sim^\beta \theta'_2 \).

4. \text{loop-if-less}: Similar reasoning as \text{jfalse}.

5. \text{type-of}: Similar to \text{mov}.

6. \text{instance-of}: No new object is created, so \( \beta' = \beta \).

The label of the value in the \( dst \) is the label of the context joined with the label of all the prototype chain pointers traversed. As \( !\sigma_1(value) \sim^\beta !\sigma_2(value) \), where \( \ell_s \) and \( \ell'_s \) are the structure labels of objects pointed to by \( !\sigma_1(value) \) and \( !\sigma_2(value) \) respectively, then by Definition 4.

- If \( \ell_s = \ell'_s = H \), then \( \Gamma(!\sigma'_1(dst)) = H \) and \( \Gamma(!\sigma'_2(dst)) = H \). So, \( !\sigma'_1(dst) \sim^\beta !\sigma'_2(dst) \) from Definition 3.

- If \( \ell_s = \ell'_s = L \), then the objects have similar properties and prototype chains. If it is not an instance and none of traversed prototype chain and objects are \( H \), then \( \Gamma(!\sigma'_1(dst)) = \Gamma(!\sigma'_2(dst)) = L \) and \text{false}. Else if it is present it has \text{true}. So, \( !\sigma'_1(dst) \sim^\beta !\sigma'_2(dst) \) from Definition 3. If any one of traversed prototype chain and objects are \( H \), then \( \Gamma(!\sigma'_1(dst)) = \Gamma(!\sigma'_2(dst)) = H \). So, \( !\sigma'_1(dst) \sim^\beta !\sigma'_2(dst) \) from Definition 3.

Only \( dst \) changes in the top call-frame of both the call-stacks. Thus, by Definition 7, \( !\sigma'_1 \sim^\beta !\sigma'_2 \). Other call-frames are unchanged and by Definition 9.
\[
\sigma_1' \sim^{\beta'} \rho_1', \rho_2' \sigma_2'.
\]
\[
\theta_1 = \theta_1' \text{ and } \theta_2 = \theta_2', \text{ so, } \theta_1' \sim^{\beta'} \theta_2'.
\]
7. enter: No new object is created, so \(\beta' = \beta\).
\[
\sigma_1' = \sigma_1 \sim^{\beta'} \rho_1', \rho_2' \sigma_2 = \sigma_2'. \theta_1 = \theta_1' \sim^{\beta'} \theta_2 = \theta_2'.
\]
8. ret: No new object is created, so \(\beta' = \beta\).

Since \(\sigma_1 \sim^{\beta} \sigma_2\) so only two cases arise for the getter flag.

- \(!\sigma_1.getter = \!\sigma_2.getter = false\): \(\sigma_1'\) is same as \(\sigma_1\) with \(!\sigma_1\) popped. Similarly, \(\sigma_2'\) is same as \(\sigma_2\) with \(!\sigma_2\) popped. As other call-frames are unchanged by Definition 9, \(\sigma_1' \sim^{\beta'} \rho_1', \rho_2' \sigma_2'\).

- \(!\sigma_1.getter = \!\sigma_2.getter = true\): only register which changes is \(\sigma_1'(\!\sigma_1.dReg \text{ and } \sigma_2'(\!\sigma_2.dReg))\). Now, if \(\Gamma(\sigma_1'(\!\sigma_1.dReg)) = \Gamma(\sigma_2'(\!\sigma_2.dReg)) = H\) then \(!\sigma_1'(\!\sigma_1.dReg) \sim^{\beta} \sigma_2'(\!\sigma_1.dReg)\) from definition 8. And if \(\Gamma(\sigma_1'(\!\sigma_1.dReg)) = \Gamma(\sigma_2'(\!\sigma_2.dReg)) = L\) then \(!\sigma_1'(\!\sigma_1.dReg) = \!\sigma_1(\!\sigma_1.dReg) \sim^{\beta} \sigma_2'(\!\sigma_2.dReg)\) = \(!\sigma_2'(\!\sigma_2.dReg)\) and \(!\sigma_1(\!\sigma_1.dReg)\).

\[
\theta_1' = \theta_1 \sim^{\beta'} \theta_2 = \theta_2'.
\]
9. end: No \(\sigma_1'\) and \(\theta_1'\).
10. call: No new object is created, so \(\beta' = \beta\).

Pushes the same node on both \(\rho\)'s (similar to \(jfalse\)). The only difference is the \(H\) field. As the CFGs are same, if it has an associated exception handler, we set the \(H\) field to \(true\) in both the runs. Else, it is \(false\). Thus, \(!\rho_1' = !\rho_2'\) is the node pushed on \(\rho\) and hence, \(\rho_1' \sim \rho_2'\).

As \(!\sigma_1(\!func) \sim^{\beta} \!\sigma_2(\!func)\), if:

- \((\Gamma(!\rho_1') = H)\): As call-frames until \(!\sigma_1\) and \(!\sigma_2\) remain unchanged, which correspond to the \(C\) field in the lowest \(H\)-labelled node and \(\sigma_1 \sim^{\beta} \rho_1, \rho_2 \sigma_2\), by Definition 9, \(\sigma_1' \sim^{\beta'} \rho_1', \rho_2' \sigma_2'\).

- \((\Gamma(!\rho_1') = L)\): Registers created in the new call-frame contain \(undefined\) with label \(L\) and, as \(\theta_1 \sim^{\beta} \theta_2\) so the function objects \(N \sim^{\beta} N'\) implying \(!\sigma_1'.CFG = !\sigma_2'.CFG\) and \(!\sigma_1'.\Sigma \sim^{\beta} \sigma_2'.\Sigma\), also return addresses are the same and the callee is the same. So \(!\sigma_1' \sim^{\beta} \sigma_2'\). Other call-frames are unchanged so, \(\sigma_1' \sim^{\beta'} \rho_1', \rho_2' \sigma_2'\).

\[
\theta_1' = \theta_1 \sim^{\beta'} \theta_2 = \theta_2'.
\]
11. call-put-result: Similar to move.
12. call-eval: \(\rho_1' \sim \rho_2'\): Similar to op-call.

In strict mode, it pushes a node on the scope-chain with label \(L\). The pushed nodes are low-equivalent. Thus, \(!\sigma_1'.\Sigma \sim^{\beta} \sigma_2'.\Sigma\) by Definition 6. In non-strict mode, it does not push anything and is similar to call. Thus, \(\sigma_1' \sim^{\beta'} \rho_1', \rho_2' \sigma_2'\) and \(\theta_1' = \theta_1 \sim^{\beta'} \theta_2 = \theta_2'.
\]
13. create-arguments: Let the argument object be created at \(x\) and \(y\) in \(\theta_1\) and \(\theta_2\), then \(\beta' = \beta \cup (x, y)\). \(\Gamma(\sigma_1'(dist)) = \Gamma(\sigma_2'(dist)) = L\) and as \(!\sigma_1 \sim^{\beta} !\sigma_2\), the objects are low-equivalent. Thus, \(!\sigma_1' \sim^{\beta'} \sigma_2'\) by Definition 7 and \(\sigma_1' \sim^{\beta'} \rho_1', \rho_2' \sigma_2'\) by Definition 9. Also, \(\theta_1' \sim^{\beta'} \theta_2'\) by Definition 5 as the objects are low-equivalent.
14. new-func: Let the function object be created at \( x \) and \( y \) in \( \theta_1 \) and \( \theta_2 \), then \( \beta' = \beta \cup (x, y) \). Function objects are low-equivalent as \( !\sigma_1, \Sigma \sim!\sigma_2, \Sigma \) and \( !\sigma_1(\text{func}) \sim!\sigma_2(\text{func}) \). \( \sigma'_1(\text{dest}) \sim!\sigma'_2(\text{dest}) \) by Definition 3. Thus, \( !\sigma'_1 \sim!\sigma'_2 \) by Definition 7 and \( \sigma'_1 \sim!_{\rho'_1, \rho'_2} \sigma'_2 \) by Definition 9. Also, \( \theta'_1 \sim!\theta'_2 \) by Definition 5 as the objects are low-equivalent.

15. create-activation: Similar to create-arguments.

16. construct: Similar to call.

17. create-this: Similar to create-this.

18. new-object: Similar to create-arguments.

19. get-by-id: No new object is created, so \( \beta' = \beta \).

As \( !\sigma_1(\text{base}) \sim!\sigma_2(\text{base}) \), either the objects have the same properties or are labelled \( H \) because of Definition 3. In case of data property, either \( \Gamma(\sigma'_1(\text{dest})) = \Gamma(\sigma'_2(\text{dest})) = H \) or \( \Gamma(\sigma'_1(\text{dest})) = \Gamma(\sigma'_2(\text{dest})) = L \) and value of prop is the same. So, by Definition 3 \( !\sigma'_1(\text{dest}) \sim!\sigma'_2(\text{dest}) \).

In case of an accessor property, only dest changes in the top call-frame of both the call-stacks, and \( \sigma'_1(\text{dest}) \sim!\sigma'_2(\text{dest}) \) since \( \sigma_1 \sim!\sigma_2 \) and \( \theta_1 \sim!\theta_2 \). Thus, by Definition 7 \( !\sigma'_1 \sim!\sigma'_2 \). Other call-frames are unchanged and by Definition 9 \( \sigma'_1 \sim!_{\rho'_1, \rho'_2} \sigma'_2 \). For \( \rho'_1 \sim \rho'_2 \), reasoning is similar to call. \( \theta_1 = \theta'_1 \) and \( \theta_2 = \theta'_2 \), so, \( \theta'_1 \sim!\theta'_2 \).

20. put-by-id: No new object is created, so \( \beta' = \beta \).

\( \sigma'_1 = \sigma_1 \sim!_{\rho'_1, \rho'_2} \sigma_2 = \sigma'_2 \).

Because \( \sigma_1 \sim!_{\rho'_1, \rho'_2} \sigma_2 \), if value is labelled \( H \), then the properties created or modified will have label \( H \), and structure labels of the respective objects will become \( H \). Else if value is labelled \( L \), then the properties created or modified will have same value and label \( L \). Thus, the objects remain low-equivalent by Definition 4 and hence, by Definition 5 \( \theta'_1 \sim!\theta'_2 \).

21. del-by-id: No new object is created, so \( \beta' = \beta \).

If the deleted property is \( H \) or if the structure label of the object is \( H \), then \( \Gamma(\sigma'_1(\text{dest})) = \Gamma(\sigma'_2(\text{dest})) = H \). Else if is labelled \( L \), then \( \Gamma(\sigma'_1(\text{dest})) = \Gamma(\sigma'_2(\text{dest})) = L \) and value is true or false depending on whether the property is deleted or not. \( \sigma'_1 \sim!_{\rho'_1, \rho'_2} \sigma'_2 \) by Definition 7 and 9. If structure labels of the objects are \( L \), they have same properties by Definition 4. If not, they have structure label as \( H \). Thus, objects remain low-equivalent by Definition 4 and \( \theta'_1 \sim!\theta'_2 \) by Definition 5.

22. put-getter-setter: Reasoning similar to put-by-id.

23. get-pnames: No new object is created, so \( \beta' = \beta \).

As \( \sigma_1 \sim!_{\rho_1, \rho_2} \sigma_2 \), \( !\sigma_1(\text{base}) \sim!\sigma_2(\text{base}) \) and so are the objects (\( \text{obj}_1 \) and \( \text{obj}_2 \), \( \text{obj}_1 \sim!\text{obj}_2 \), as \( \theta_1 \sim!\theta_2 \). Thus, the structure label of the object is either \( H \) in both the runs or \( L \) and have the same properties with values (Definition 4). The IPD in both the cases is the same and so is the \( C \) field. The \( mH \) field is set to false. Thus, \( \rho'_1 \sim \rho'_2 \).

For \( \sigma'_1 \sim!_{\rho'_1, \rho'_2} \sigma'_2 \), it is similar to mov, but done for dst, i and size.

\( \theta'_1 = \theta_1 \sim!\theta'_2 = \theta'_2 \).

24. next-pname: Similar to move, but done for dst and base.
25. resolve: No new object is created, so $\beta' = \beta$.
   If property is found in a L object and the scope-chain node labels are also L, then the property value is the same as $\sigma_1 \sim^\beta!\sigma_2$. If it is in H object or any scope-chain node labels are H or have a *, then label of the property is H or *. Thus, $\sigma_1' (dst) \sim^\beta!\sigma_2' (dst)$ by Definition 3. Thus, $\sigma_1' \sim_{\rho_1', \rho_2'} \sigma_2'$. If property is not found in both runs, it is similar to throw. If property is not found in second run, then in the first run the property is in H context. So, the exception thrown is also H. Until, the call-frame of $!\rho_2'.ipd$, call-frames are unchanged, so $\sigma_1' \sim_{\rho_1', \rho_2'} \sigma_2'$. $\theta_1' = \theta_1 \sim^{\beta'} \theta_2 = \theta_2'$.
26. resolve-skip: Similar to resolve.
27. resolve-global: Similar to resolve.
28. resolve-base: Similar to resolve.
29. resolve-with-base: Similar to resolve.
30. get-scoped-var: No new object is created, so $\beta' = \beta$.
   Reads index register in object in skipth node in the scope-chain and writes into dst. As $(!\sigma_1, \Sigma \sim^\beta!\sigma_2, \Sigma)$, the value, if labelled L is the same, else is labelled H or * . By Definition 3 $\sigma_1' (dst) \sim^\beta!\sigma_2' (dst)$ and by Definition 7 $\sigma_1' \sim^\beta!\sigma_2'$. Thus, $\sigma_1' \sim_{\rho_1', \rho_2'} \sigma_2'$. $\theta_1' = \theta_1 \sim^{\beta'} \theta_2 = \theta_2'$. 
31. put-scoped-var: No new object is created, so $\beta' = \beta$.
   Writes into the scope chain node the same value, if “value” is labelled L. If it is labelled * in any of the runs, scope chains remain equivalent. If value is H, it checks the label of register and puts the value with label H or * . Thus, $(!\sigma_1', \Sigma \sim^\beta!\sigma_2', \Sigma)$ by Definition 6 and $(!\sigma_1' \sim^\beta!\sigma_2')$ by Definition 7. By Definition 4 $\sigma_1' \sim_{\rho_1', \rho_2'} \sigma_2'$. $\theta_1' = \theta_1 \sim^{\beta'} \theta_2 = \theta_2'$. 
32. push-scope: No new object is created, so $\beta' = \beta$.
   Pushes in the scope-chain a node containing the object in “scope” with node label L. As $(!\sigma_1, scope) \sim^\beta!\sigma_2 (scope))$ and $(!\sigma_1, \Sigma \sim^\beta!\sigma_2, \Sigma)$ by Definition 6. Registers and other call-frames remain the same, so, $\sigma_1' \sim_{\rho_1', \rho_2'} \sigma_2'$. $\theta_1' = \theta_1 \sim^{\beta'} \theta_2 = \theta_2'$. 
33. pop-scope: No new object is created, so $\beta' = \beta$.
   Pops a node from the scope chain if $\Gamma (!\sigma_1, \Sigma) = \Gamma (!\sigma_1, \Sigma) \neq (* \lor H)$. As $(!\sigma_1, \Sigma \sim^\beta!\sigma_2, \Sigma)$, so $(!\sigma_1', \Sigma \sim^\beta!\sigma_2', \Sigma)$. Other registers remain the same, so, $\sigma_1' \sim_{\rho_1', \rho_2'} \sigma_2'$. $\theta_1' = \theta_1 \sim^{\beta'} \theta_2 = \theta_2'$. 
34. jmp-scope: Similar to pop-scope.
35. throw: No new object is created, so $\beta' = \beta$.
   The property of IPD ensures that $!\sigma_1' = (!\rho_1, C)$ and $!\sigma_2' = (!\rho_2, C)$. The $!\sigma_1'$ and $!\sigma_2'$ and the ones below them remain unchanged. Thus, $\sigma_1' \sim_{\rho_1', \rho_2'} \sigma_2'$ by Definition 8 $\theta_1' = \theta_1 \sim^{\beta'} \theta_2 = \theta_2'$. 
36. catch: Similar to mov.

Lemma 3 (Supporting Lemma 2) Suppose
1. $\langle \iota_0, \theta_0', \sigma_0', \rho_0' \rangle \sim^{n-1} \langle \iota_1', \theta_1', \sigma_1', \rho_1' \rangle \sim^{n-1} \langle \iota_n', \theta_n', \sigma_n', \rho_n' \rangle$. 

2. \( \langle \iota_0, \theta''_0, \sigma''_0, \rho''_0 \rangle \sim \langle \iota''_1, \theta''_1, \sigma''_1, \rho''_1 \rangle \sim^{m-1} \langle \iota''_m, \theta''_m, \sigma''_m, \rho''_m \rangle \),
3. \( (\rho'_0 \sim \rho''_0), (\sigma'_0 \sim_{\beta} \rho'_0, \sigma''_0), (\theta'_0 \sim \theta''_0) \),
4. \( (\Gamma(\rho'_0) = \Gamma(\rho''_0) = L), (\Gamma(\rho'_m) = \Gamma(\rho''_m) = L), \)
5. \( \forall(0 < i < n). (\Gamma(\rho'_i) = H) \land \forall(0 < j < m). (\Gamma(\rho''_j) = H), \)
then \( (\iota'_n = \iota''_n), (\rho'_n \sim \rho''_n), (\sigma'_n \sim_{\beta} \rho'_n, \sigma''_n), \) and \( (\theta'_n \sim_{\beta} \theta''_n) \).

Proof. Starting with the same instruction and high context in both the runs, we might get two different instructions, \( \iota'_1 \) and \( \iota''_1 \). This is only possible if \( \iota \) was some branching instruction in the first place and this divergence happened in a high context. Now,

1. To prove \( \iota'_n = \iota''_m \):
   From the property of the IPDs we know that if \( \iota_0 \) pushes a \( H \) node on top of pc-stack which was originally \( L \), \( \text{IPD}(\iota_0) \) pops that node. Since we start from the same instruction \( \iota_0 \), \( \iota'_n = \iota''_m = \text{IPD}(\iota) \), where \( \Gamma(\rho) = L \).

2. To prove \( \rho'_n \sim \rho''_m \):
   - \( n > 1 \) and \( m > 1 \): \( \Gamma(\rho'_1) = \Gamma(\rho''_1) \), because \( \iota_0 \) pushes equal nodes and \( \iota'_1, \iota''_1 \) are not the IPDs. As \( \rho'_0 \sim \rho''_0 \) and \( \Gamma(\rho'_1) = \Gamma(\rho''_1) \), from Lemma 2 we get \( \rho'_1 \sim \rho''_1 \) and \( \rho'_1, \rho''_1 \) are not the IPDs. From Corollary 2 we get \( \rho'_n \sim \rho''_m \).
   - \( n = 1 \) and \( m > 1 \): If \( \Gamma(\rho'_1) \neq \Gamma(\rho''_1) \) and \( \iota'_1 = \text{IPD}(\iota_0) \), then \( \Gamma(\rho'_1) = L \).
   - \( n > 1 \) and \( m = 1 \): Symmetric case of the above.

3. To prove \( \sigma'_n \sim_{\beta} \rho'_n, \rho''_n \sigma''_m \):
   (a) \( n > 1 \) and \( m > 1 \): From Lemma 2 we get \( \sigma'_1 \sim_{\rho'_1, \rho''_1} \sigma''_1 \). From Corollary 2 we get \( \sigma'_n \sim_{\rho'_n, \rho''_n} \sigma''_m \).
   - We do case analysis on the different cases of Definitions 3 for \( v_1 \) and \( v_2 \), to show \( v_n \sim_{\beta} v_m \). As \( (\forall(1 \leq i < n). (\Gamma(\rho'_i) = H) \land \forall(1 \leq j < m). (\Gamma(\rho''_j) = H)) \):
     - If \( v_1 = v_2 \) or \( (\Gamma(\rho'_i) = H) \land (\Gamma(\rho''_j) = H) \):
       - If \( v_1 = v_2 \) and \( \Gamma(v_1) = \Gamma(v_2) = L \), then either \( v_n = v_m \) or \( (\ast = \Gamma(v_n)) \lor (\ast = \Gamma(v_m)) \). By Definition 3(1), \( v_n \sim_{\beta} v_m \).
• If $\Gamma(v_1) = \Gamma(v_2) = H$, then $\Gamma(v_n) = \Gamma(v_m) = H$. By Definition 3.2, $v_n \sim^\beta v_m$.
• If $* = \Gamma(v_1)$, then $* = \Gamma(v_n)$ and if $* = \Gamma(v_2)$, then $* = \Gamma(v_m)$.

By Definition 3.1, $v_n \sim^\beta v_m$.

- Let $S_1$ and $S_2$ be the scopechains in $\sigma_0'$ and $\sigma_0''$. And $S_n$ and $S_m$ represent the scopechains in $\sigma_n'$ and $\sigma_m''$, i.e., $S_n$ and $S_m$ are the respective scope-chains in the $n$th and $m$th step of the two runs and $\ell_n$ and $\ell_m$ are their node labels. For scope chain pointers the following cases arise:

  i. $S_1 = S_2 = \text{nil}$: In this case $S_n$ and $S_m$ either remain nil or its head will have a $H$ label, because of the rules of the instructions that modify the scope-chain.

  ii. $S_1 = (s_1, \ell_1) : \Sigma_1$ and $S_2 = (s_2, \ell_2) : \Sigma_2$:
  
  A. $\ell_1 = * \vee \ell_2 = *$: In this case $\ell_n$ and $\ell_m$ will be $*$. too.
  
  B. $\ell_1 = \ell_2 = H$: In this case $\ell_n$ and $\ell_m$ will be $H$ too.
  
  C. $\ell_1 = \ell_2 = L \wedge \Sigma_1 \sim^\beta \Sigma_2$: In this case $\ell_n = *$ and $\ell_m = *$ or scopechains remain unchanged.

(b) $n = 1$ and $m > 1$:

In case of $\text{if} false$ and $\text{loop-if-less}$, $\sigma_0' = \sigma_1'$ and $\sigma_0'' = \sigma_1''$. And in case of $\text{get-pnames}$, if $n = 1$ and $m \neq 1$, $\Upsilon(\sigma_0'(\text{base})) = \Upsilon(\sigma_0''(\text{base})) \neq \text{undefined}$. Because $\sigma_0' \sim^\beta \sigma_0''$, $\Upsilon(\sigma_0'(\text{base})) \sim^\beta \Upsilon(\sigma_0''(\text{base}))$. Hence, $\Gamma(\sigma_0'(\text{base})) = \Gamma(\sigma_0''(\text{base})) = H$. Thus, $\Gamma(\sigma_1'(\text{base})) = \Gamma(\sigma_1''(\text{base})) = H$ and similarly, $\Gamma(\sigma_1'(i)) = \Gamma(\sigma_1''(i)) = H$. Other registers remain unchanged and so do the other call-frames.

Thus, $\sigma_1' \sim_{\sigma_1', \sigma_1''} \sigma_1''$. From the case (a) above, we know that if $\sigma_1' \sim_{\sigma_1', \sigma_1''} \sigma_1''$, then $\sigma_n' \sim_{\sigma_n', \sigma_n''} \sigma_n''$.

(c) $n > 1$ and $m = 1$: Symmetric case of the above.

4. To prove $\theta_n' \sim^\beta \theta_m''$:

(a) $n > 1$ and $m > 1$: From Lemma 2 we get $\theta_1' \sim^\beta \theta_1''$. From Corollary 3 we get $\theta_1' \sim^\beta \theta_n' - 1$. And from Lemma 1 we have $\theta_n' - 1 \sim^\beta \theta_n'$. Assume $O_1$ is an object at $x$ in $\theta_1'$ and $O_2$ is an object at $y$ in $\theta_1''$, such that $(x, y) \in \beta$ and $O_n$ and $O_m$ are the respective objects in the $n$th and $m$th step of the two runs. We do case analysis on the different cases of Definitions 4 for $O_1$ and $O_2$, to show $O_n \sim^\beta O_m$.

- If $\Gamma(O_1) = \Gamma(O_2) = H$, then $\Gamma(O_n) = \Gamma(O_m) = H$. By Definition 4

  $O_n \sim^\beta O_m$.

- If $O_1 = O_2 \wedge \Gamma(O_1) = \Gamma(O_2) = L$, then $O_n = O_m \wedge \Gamma(O_n) = \Gamma(O_m) = L$. By Definition 4

  $O_n \sim^\beta O_m$.

Similarly, for function objects, the structure labels would remain $H$ if they were originally $H$ or will remain $L$ with the same CFGs and scope-chains.

(b) $n = 1$ and $m > 1$: In case of $\text{if} false$, $\text{loop-if-less}$, $\text{get-pnames}$, $\theta_0' = \theta_1'$ and $\theta_0'' = \theta_1''$. Thus, $\theta_1' \sim^\beta \theta_1''$. From the case (a) above, we know that if $\theta_1' \sim^\beta \theta_1''$, then $\theta_n' \sim^\beta \theta_m''$.

(c) $n > 1$ and $m = 1$: Symmetric case of the above.
**Definition 12 (Epoch-trace)** An epoch-trace $(\mathcal{E})$ over a trace $\mathcal{T} = s_1 :: s_2 :: \ldots :: s_n$ where $s_i = \langle \iota_i, \theta_i, \sigma_i, \rho_i \rangle$, is defined inductively as:

$$\mathcal{E}(\text{nil}) := \text{nil}$$

$$\mathcal{E}(s_i :: \mathcal{T}) := \begin{cases} 
\mathcal{E}(\mathcal{T}) & \text{if } \Gamma(!\rho_i) = L, \\
\mathcal{E}(\mathcal{T}) & \text{else if } \Gamma(!\rho_i) = H.
\end{cases}$$

**Theorem 1 (Termination-Insensitive Non-interference).**

Suppose $\mathcal{P}$ and $\mathcal{P}'$ are two program evaluations. Then for their respective epoch-traces given by:

$$\mathcal{E}(\mathcal{T}(\mathcal{P})) = s_1 :: s_2 :: \ldots :: s_n,$$

$$\mathcal{E}(\mathcal{T}(\mathcal{P}')) = s'_1 :: s'_2 :: \ldots :: s'_m,$$

if $s_1 \sim^\beta s'_1$ and $n \leq m$,

then

$$\exists \beta_n \supseteq \beta : s_n \sim^{\beta_n} s'_n.$$

**Proof.** Proof proceeds by induction on $n$.

**Basis:** $s_1 \sim^\beta s'_1$, by assumption.

**IH:** $s_k \sim^{\beta_k} s'_k$, where $\beta_k \supseteq \beta$.

To prove: $\exists \beta_{k+1} \supseteq \beta : s_{k+1} \sim^{\beta_{k+1}} s'_{k+1}$.

Let $s_k \sim_i s_{k+1}$ and $s_k \sim_{i'} s'_{k+1}$, then:

- $i = i' = 1$: From Lemma 2, $s_{k+1} \sim^{\beta_{k+1}} s'_{k+1}$ where $\beta_{k+1} \supseteq \beta$.
- $i > 1$ or $i' > 1$: From Lemma 3, $s_{k+1} \sim^{\beta_{k+1}} s'_{k+1}$ where $\beta_{k+1} = \beta$.

**Corollary 4** Suppose:

1. $\langle \iota_1, \theta_1, \sigma_1, \rho_1 \rangle \sim^\beta \langle \iota_2, \theta_2, \sigma_2, \rho_2 \rangle$
2. $\langle \iota_1, \theta_1, \sigma_1, \rho_1 \rangle \sim^* \langle \text{end}, \theta'_1, [], [] \rangle$
3. $\langle \iota_2, \theta_2, \sigma_2, \rho_2 \rangle \sim^* \langle \text{end}, \theta'_2, [], [] \rangle$

Then, $\exists \beta' \supseteq \beta$ such that $\theta'_1 \sim^{\beta'} \theta'_2$.

**Proof.** $\sigma_1, \sigma_2$ and $\rho_1, \rho_2$ are empty at the end of * steps. From the semantics, we know that in L context both runs would push and pop the same number of nodes. Thus, both take same number of steps in L context. Let $k$ be the number of states in L context. Then in Theorem 4 $n = m = k$. Thus, $s_k \sim^{\beta_k} s'_k$, where $s_k = \langle \text{end}, \theta'_1, [], [] \rangle$ and $s'_k = \langle \text{end}, \theta'_2, [], [] \rangle$. By Definition 10, $\theta'_1 \sim^{\beta'} \theta'_2$, where $\beta' = \beta_k$. 