HYBRID FLOW-SENSITIVE SECURITY MONITOR
FOR JAVASCRIPT

Bassam Sayed, Issa Traoré, and Amany Abdelhalim

May 2016
University of Victoria
Department of Electrical and Computer Engineering
Technical Report No. ECE-2016-05-19-1

UNIVERSITY OF VICTORIA
HYBRID FLOW-SENSITIVE SECURITY MONITOR FOR JAVASCRIPT

Bassam Sayed, Issa Traoré, and Amany Abdelhalim

University of Victoria
Department of Electrical and Computer Engineering
P.O. Box 1700,
STN CSC, Victoria BC,
V8W 2Y2, Canada
www.ece.uvic.ca

May 2016

Technical Report No. ECE-2016-05-19-1
# Contents

1 Introduction

2 Proposed JavaScript Security Type System
   2.1 Motivating Example
   2.2 Syntactic Conventions and Meta-notation
   2.3 Operational Semantics
   2.4 Extending The Semantics
   2.5 Flow-Sensitive Security Type System

3 Hybrid Flow-Sensitive Security Monitor For JavaScript
   3.1 VM Monitor
   3.2 Attack Model and Security Property

4 IF-Transpiler: Inlining of Hybrid Flow-Sensitive Security Monitor For JavaScript
   4.1 Transformation Stage
   4.2 Inlining Stage
   4.3 Soundness of the Inlined Security Monitor

5 Appendix A: Proof of the Soundness of the Hybrid Flow-Sensitive Security Monitor

6 Appendix B: Proof of the Observational Equivalence of the Inlined Monitor
## List of Figures

1. Flow-Sensitive vs. Flow-Insensitive model example. ................................ 4
2. Two examples of the flow-sensitivity attack that demonstrate how the `collect()` and `upgrade()` functions can be used. ........................................ 21
3. Implicit information flow using block structured control-flow in the left side and using non-block structured control-flow in the right side. ............... 22
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Syntactical conventions, meta-variables, and syntax for values.</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Syntax of Expressions</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Syntax of Statements.</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Statement Typing Part(A)</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Statement Typing Part(B)</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>Expression Typing.</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>Additional statements for monitored execution.</td>
<td>19</td>
</tr>
<tr>
<td>8</td>
<td>Statements transitions events.</td>
<td>19</td>
</tr>
<tr>
<td>9</td>
<td>JavaScript operational semantics with events Part (A).</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>JavaScript operational semantics with events Part (B).</td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td>VM Monitor Transitions.</td>
<td>25</td>
</tr>
<tr>
<td>12</td>
<td>Collecting variables and functions from a statement S.</td>
<td>26</td>
</tr>
<tr>
<td>13</td>
<td>Transformation function $T(S)$.</td>
<td>29</td>
</tr>
<tr>
<td>14</td>
<td>Security level of expression $e$.</td>
<td>30</td>
</tr>
<tr>
<td>15</td>
<td>Inlining function $I(S)$ [Part A].</td>
<td>31</td>
</tr>
<tr>
<td>16</td>
<td>Inlining function $I(S)$ [Part B].</td>
<td>32</td>
</tr>
<tr>
<td>17</td>
<td>Inlining function $I(S)$ [Part C].</td>
<td>33</td>
</tr>
</tbody>
</table>
1 Introduction

Modern web applications are characterized by their interactivity, reactivity, and service composition, which make the user experience engaging and delightful. In order for the web applications to provide such engagement, they have to rely on third-party libraries that provide services and/or utility functions. These functions range from manipulating the webpage’s "DOM" (Document Object Model) to facilitating the communication back to the hosting server. The integration of these different libraries happens at the client-side while the page is being rendered to the end-user. Mostly, the integration happens by including JavaScript libraries from different sources. These sources could be the hosting server itself or third-party content distribution networks (CDN). Including libraries from third-party CDNs has two major advantages to web application developers. Firstly, the developer of the web application does not have to maintain several third-party libraries on her server and at the same time, the web application will have access to the latest version of the library without updating the web application itself. Secondly, in many cases, the library is already downloaded and cached by the browser as a result of visiting another web application that happens to use the same third-party library. In this case the browser will not re-download the library, instead it will load it directly from its local cache, which yields faster load time for the web application. However, the fact that these third-parties libraries may have unrestricted access to the content of the webpage poses the threat that they could leak private information to unauthorized channels.

Web browsers implement what is known as Same-Origin-Policy (SOP). The intent of SOP is to prevent embedded HTML documents originating from different Internet domains from accessing webpage content, even if the different HTML documents are rendered in the same page at the client-side, e.g., using iframes. This means that scripts that are included in the same document via script tags will have full access to the current document contents, and scripts loaded in other iframes will be completely isolated. In other words, either the scripts are sealed off or fully integrated with the webpage content, which makes the SOP protect resources belonging to the same origin rather than protecting resources belonging to the end-user. A good example of such problem is the well known client-side attacks, cross-site scripting (XSS) and cross-site request forgery (CSRF), both of which are still in the top 10 list of the OWASP project [5]. Clearly the question is, how do we strike the right balance between usability and security?

Generally Web-based attacks can be classified as server-side attacks or client-side attacks. Server-side attacks exploit vulnerabilities in server-side web application components causing harm to the organization hosting such servers. A significant amount of research has been performed on how to secure the servers hosting sensitive information. The focus on protecting the server-side and the emergence of Web 2.0 technologies made the attackers switch focus to the client-side. Instead of attacking directly the servers of an organization, the focus is now primarily on attacking clients inside such organization. In particular attacks targeting directly the end-user have increased in number and sophistication in recent years. In many cases the intent is not just to attack a specific organization but rather to attack the end-users themselves since end-users are not as heavily guarded as servers and represent in some way the weakest link.

Client-side attacks exploit the trust relationship between an end-user and a website. Generally, when an end-user using a web browser visits a website, he/she assumes that the
website poses no harm to his system and is not going to obtain information without his consent. However, in many circumstances this is not the case.

There are two categories of malicious websites. Firstly, a malicious website can be a legitimate website that has been attacked to host the malicious content, like the cross-side scripting worms Samy and Mikeyy, which attacked My Space and Twitter, respectively [17, 16]. Secondly, the visited website could host the malicious content intentionally to attack the end-users. Usually, the end-user is tricked to visit these malicious websites through some form of social engineering medium, such as an email or a message posted in a forum with a link to the malicious website.

Client-side attacks can be widely categorized as either browser-specific or browser-agnostic attacks. Browser-specific attacks are the attacks that target a specific type of browser on a specific platform. For instance, attacks that rely on Microsoft ActiveX components are only possible on Windows platform running Microsoft Internet Explorer. If the end-user uses other types of browsers (e.g. Mozilla Firefox), the attack will not be successful. On the other hand, browser-agnostic attacks do not depend on specific type of browser or platform. These types of attacks take advantage of the fact that all the browsers running on any type of platform (even mobile platforms like smart phones and tablets) have to support specific set of web standards such as HTML, JavaScript, CSS, etc. For instance, cross-site scripting attacks that steal end-users’s cookie do not rely on a specific type of browser or platform since all major browsers have to support cookie mechanism to function properly. Browser-agnostic attacks are the stealthiest and hardest to detect.

Client-side attacks exhibit some common characteristics that set them apart. These characteristics fall under three main categories, namely, the attack scenario, the browser architecture, and the scripting language, as discussed in the following.

- **Attack Scenario:** usually the attack scenario starts by the end-user visiting a website that contains the malicious script (whether this script was injected into a benign website or the website is malicious). When the end-user’s browser renders the website, the malicious script is executed within the security context of the current page allowing it to manipulate the current content of the web page or perform actions on the end-user behalf.

- **Browser Architecture:** any web browser has to support a specific set of web standards such as HTML, XML, JavaScript, CSS, and enforce at least SOP as a common policy.

- **Client-side Scripting Language:** client-side attacks rely heavily on JavaScript. JavaScript is the ”only” supported language when it comes to the client-side of the web. There are other scripting languages used by web-developers such as CoffeeScript [3] and ClojureScript [12] to program the client-side of the web applications. However, all of them get compiled down to JavaScript. Recently, Google proposed a new programming language to program the two sides of the web, the client-side and the server-side, named ”Dart” [8]. Dart is only supported by an experimental version of Google Chrome [6]. However, Dart SDK supports compiling client-side Dart code to JavaScript to enable running Dart code inside standard web browsers.

All modern web browsers implement a complete JavaScript engine (virtual machine) as a component of their architectures. JavaScript is a highly dynamic, weakly typed, object-
Hybrid Flow-Sensitive Security Monitor for JavaScript

based, asynchronous, and event-based scripting language. JavaScript implementation inside web browsers is granted complete access to all aspects of the current web page. JavaScript can access and modify the document object model (DOM) objects and their properties. It can register for events coming from the user interface (UI) or from other objects such as networking object (e.g., XMLHttpRequest); it can manipulate the cookies and modify the browser history, etc. The usage of JavaScript as a dynamic scripting language combined with the ambiguous same-origin-policy lead to the failure to enforce adequate information flow policy.

Based on the above characteristics, it is clear that the structure and mode of operation of the JavaScript language provide a fertile ground for conducting client-side attacks. The objective of the research presented in this technical report is to mitigate client-side web attacks by developing a framework for rigorously enforcing information flow policies in the underlying client JavaScript implementation.

In order to monitor the information flow in JavaScript code and simultaneously preserve program semantics, JavaScript’s operational semantics must rigorously (i.e., formally) be defined. The operational semantics is defined as rules that are applied when a specific expression or statement of the language is executed.

We start the formalization of our proposed approach by developing a flow-sensitive security type system. The security type system provides the static analysis information for the security monitor. Then we instrument the operational semantics of the JavaScript language to generate events that are visible to the security monitor at the runtime. The security monitor combines information from both the type system and the generated events to guide its own transitions. Whenever there is an illegal information flow, the monitor applies its enforcement policy. We build on the proof presented in [14] to show that our proposed approach implements a sound termination-insensitive non-interference security policy. Then we present the formalization of our inlining transpiler and prove the observational equivalence of the inlined monitor with respect to the hybrid flow-sensitive security monitor. Our transpiler is syntax directed, as such, we show precisely how the instrumentation happens for each type of JavaScript expression and statement.
// Global variables
var private = ?, public = false;
if(private) {
    public = true; // Direct information flow
} // Indirect information flow

// Public's security class is as high as the private's variable at this point
public = 10; // Assigning a literal to public variable yields // a low security class
// This output statement will be allowed by our model since // a low security class.
output(public);
// Output the value of the public variable on a low security
// output channel.

Figure 1: Flow-Sensitive vs. Flow-Insensitive model example.

2 Proposed JavaScript Security Type System

In this section we briefly introduce our meta-notation and syntactic conventions used in the rest of the dissertation. Our proposed notation has some similarities to the meta-notation and syntactic conventions proposed in [11].

We briefly present the operational semantics of the language and extend them to cover a hypothetical output statement \( \text{output}_\ell(e) \) that is not defined by the language and two relations that model the scope-chain and prototype-chain of the language. The addition of the hypothetical output statement is important for our correctness proof. We also present our proposed security type system for the language.

2.1 Motivating Example

Our proposed type system is flow-sensitive in the sense that the security classes of program data can "float" (upgrade or downgrade) based on the information flow. In contrast, in flow-insensitive models the security classes of the data elements do not float. By allowing the security types of program data to "float", our type system can accept more programs than other models that are flow-insensitive. For example, the code in Figure 1 will not be accepted by models that are not flow-sensitive, however, it will be accepted by our model. The reason this program will not be accepted by flow-insensitive models is the existence of flow of information from the private variable to the public variable. It is important to note here that the actual value of the private variable will not matter. As indicated in the example, there will be either, a direct information flow in the case where private value is true, or an indirect information flow in the case where the value of the private variable is false. However, in both cases there is an information flow from a higher security class variable to a lower security class variable (public) which should not be allowed, hence, the program will not be accepted by flow-insensitive models. On the contrary, our model will accept this program as the security class of the public variable is allowed to float (upgraded in this case). So at line number 7, the security class of the public variable is as high as
Hybrid Flow-Sensitive Security Monitor for JavaScript

\(\vec{x}\) \(\{x_1, x_2, \ldots\}\) % list of elements

\(t^*\) \(t_1 \ldots t_n\) % \(t^*\) in the nonempty case

\([t]\) \(t\) % \(t\) is optional, in case of ambiguity

\(t | s\) % the \([\) symbol is escaped by \(\[]\)

\(\text{Ident} = \ x \ | \ y \ | \ foo \ | \ bar \ | \ldots\) % Identifiers

\(b = \ \text{true} \ | \ \text{false}\) % Booleans

\(m = "foo" \ | \"bar" \ | \ldots\) % Strings

\(n^+ = \ \text{Infinity} \ | \ 0 \ | \ 1 \ | \ 2 \ | \ldots\) % Positive numbers

\(n = n^+ \ | \ -n^+ \ | \ \text{NaN}\) % Positive and negative numbers plus NaN

\(\text{Prim} = b \ | \ m \ | \ n \ | \ \text{null} \ | \ \text{undefined}\) % Primitives

\(\text{Ref} = r\) where \(r \in \text{dom}(H)\) % references to objects in heap memory

\(v_a ::= m \ | \ n \ | \ b \ | \ \text{null} \ | \ \text{undefined} \ | \ r\) % values

Table 1: Syntactical conventions, meta-variables, and syntax for values.

the private variable. However, at line number 10, the public variable is assigned a value of a literal (we assume that literals have a low security class), which means that its security class is updated once again, in this case it is downgraded. At line number 15, the output statement will be allowed since the last assignment to the public variable yielded a low security class.

2.2 Syntactic Conventions and Meta-notation

Our proposed meta-variables and syntactical conventions are outlined in Table 1. The values outlined in Table 1 are standard values and closely reflect the values in the JavaScript language. All the objects defined in the JavaScript programs are allocated in the heap memory. We define the heap memory \(H\) to be a mapping between set of references \(\text{Ref}\) to set of objects \(\text{Obj}\), formally, \(H: \text{Ref} \mapsto \text{Obj}\). In JavaScript, objects are records of primitive values, references, or functions indexed by strings, formally, \(\text{Obj} : m \mapsto \text{Ref} \cup \text{Prim} \cup \text{Func}\), where \(\text{Func}\) is a set of parsed function literals. In JavaScript terminology, these strings are called properties \(^1\). Some properties are internal and cannot be accessed by the user. For the purpose of clarity, internal properties are preceded by "@" sign to distinguish them from user definable properties. We use \([p_1 \mapsto va_1, \ldots, p_n \mapsto va_n]\) notation for partial function mapping \(p_1\) to \(va_1\), ..., and \(p_n\) to \(va_n\), respectively. We use \(f(r)(p)\) notation to mean \((f(r))p\), which is, the application of the image of \(r\) by \(f\) (which is assumed to be a function), to \(p\).

2.3 Operational Semantics

In order to dynamically enforce the information flow control policy in a rigorous and precise way, we need the operational semantics of the language. We use small-step operational semantics to define the rules that are applied when a specific JavaScript statement is evaluated. The general form of an operational semantic rule is as follows:

\[
\text{computation} \\
(\text{current state}) \rightarrow (\text{end state})
\]

\(^1\)This is also true for Arrays and Functions since both are types of objects.
The semantic rules are read bottom to top, left to right. Given a JavaScript expression \( e \), it is pattern-matched to an expression evaluation semantic rule. Then the rule is applied performing the attached computation and transition to the end state. We start by the semantics of expressions since they are the basic building blocks of the operational semantics. Table 2 outlines the syntax for expressions. We use "\( \odot \)" to denote binary operator, "\( \odot \)" to denote unary operator and "\( \odot \)" to denote postfix operator.

The initial heap contains the native objects that implement the functions, constructors, and prototypes. It also contains the initial scope object and the global object represented by "@Global". The global object is the root of the scope chain, as such its "@Scope" property is set to null and the "@this" property points to itself "#Global".

### 2.4 Extending The Semantics

We add the following hypothetical output statement \( \text{output}_\ell(e) \) to the list of statements defined in table 3. This output statement can correspond to a send method call of an XMLHttpRequest object or any other output communication primitive provided by the JavaScript hosting environment (e.g. browser or NodeJS [9]). The meaning of output statement \( \text{output}_\ell(e) \) is: given the current heap memory \( H \) and scope object reference \( r \), evaluate expression \( e \) and output it’s value \( va \) on output channel with security level \( \ell \). This was referred to as observational effect in [14]. The following operational semantics outlines the labeled transition of the output rule (assuming the evaluation of expression \( e \) did not generate an exception):

\[
va = H(r)(e) \\
\langle H, r, \text{output}_\ell(e) \rangle \rightarrow_{\alpha(va)} \langle H, r, va \rangle
\]

The output event \( \alpha(va) \) is triggered only when stepping of \( \text{output}_\ell(e) \) statement happens, otherwise, no output events get generated and all the transitions are considered internal (later in the paper we define what is an internal event). The \( \text{output}_\ell(e) \) statement
Hybrid Flow-Sensitive Security Monitor for JavaScript

Table 3: Syntax of Statements.

triggers the output event \( o_\ell(va) \), where \( va \) is the value of the expression \( e \) in the current heap memory \( H \) and current scope object \( r \), and \( \ell \) is the security level of the output channel. We assume that every channel has a security level \( \ell \) associated with it and it does not change over time. We assume attackers with security level \( \ell \) can only observe outputs from channels having security level lower than or equal to \( \ell \) denoted \( \downarrow \ell \), ignoring covert channels outlined in [10].

We define the following two relations for the inspection of the scope and prototype chain of objects [15]. Definition 1 defines the scope inspection relation. If \( (H,r_0,x) \triangleright_{\text{Scope}} r_1 \), then semantically this means that \( r_1 \) is the closest scope object to \( r_0 \) in the scope-chain that defines a binding for variable \( x \) and all the objects in the scope-chain including \( r_0 \) and \( r_1 \) are in the range of the heap memory \( H \).

**Definition 1.** Scope-Chain Inspection Relation \( \triangleright_{\text{Scope}} \) is defined recursively as follows:

\[
\begin{align*}
\text{NULL} & \triangleright_{\text{Scope}} \text{null} \\
(H,\text{null},x) & \triangleright_{\text{Scope}} \text{null} \\
\text{BASE} & \quad \frac{x \in \text{dom}(H(r))}{(H,r,x) \triangleright_{\text{Scope}} r} \\
\text{LOOKUP} & \quad \frac{x \notin \text{dom}(H(r))}{(H,H(r)(@\text{scope}),x) \triangleright_{\text{Scope}} r'} \\
\end{align*}
\]

Definition 2 defines the prototype-chain inspection relation. JavaScript implements prototypical based object inheritance. Every object (including object literals) includes a \( \_\text{proto}_- \) property that references its prototype object. The \( \_\text{proto}_- \) property is used when looking-up properties in objects. For example, if property \( m \in \text{dom}(r_o) \), then the property lookup returns \( r_o(m) \), otherwise the \( \_\text{proto}_- \) property is used to find if the prototype object of \( r_o \) defines property \( m \), if not, its prototype is checked and so forth until \( \_\text{proto}_- \) property is null. This is known as prototype-chain. If \( (H,r,m) \triangleright_{\text{Proto}} r' \), then semantically this means that \( r' \) is the closest object to \( r \) in the prototype-chain that defines a binding for
property \( m \) and all the objects in the prototype-chain including \( r \) and \( r' \) are in the range of the heap memory \( H \).

**Definition 2.** Prototype-Chain Inspection Relation \( \triangleright_{\text{Proto}} \) is defined recursively as follows:

\[
\begin{align*}
\text{NULL} & \triangleright_{\text{Proto}} \text{null} \quad \text{BASE} \quad m \in \text{dom}(H(r)) \quad \langle H, r, m \rangle \triangleright_{\text{Proto}} r \\
\text{m} \notin \text{dom}(H(r)) \quad \langle H, H(r)(\mathit{proto}), m \rangle \triangleright_{\text{Proto}} r' \\
\end{align*}
\]

### 2.5 Flow-Sensitive Security Type System

Security Types \( \tau \) are drawn from set \( S \) of security levels (classes). The triple \( \langle S, \sqcup, \sqsubseteq \rangle \) is the universally bounded lattice \( \mathcal{L} \) of our flow-sensitive type system, where:

- \( S = \{ l_1, l_2, \ldots \} \): a set of security levels (e.g. \( S = \{ L, H \} \) where \( L \) is low and \( H \) is high).
- \( \sqcup \): a lattice join operator returning the least upper bound over two given levels.
- \( \sqsubseteq \): a partial order relation between security levels (e.g. \( L \sqsubseteq H \) and \( H \not\sqsubseteq L \)).

The typing rules of our flow-sensitive system are defined in Tables 4 and 5. For a statement \( S \), the typing judgements have the form:

\[
H, r, pc \vdash \mathcal{L} \quad \Gamma, \Sigma, \Lambda \quad \{ S \} \quad \Gamma', \Sigma', \Lambda'
\]

where \( \Gamma, \Gamma' : \mathcal{Ref} \rightarrow m \rightarrow \mathcal{L} \) are type environments that map properties of objects to security levels. \( \mathcal{Ref} \) is a set of references, \( m \) is a string (property or variable name), and \( \mathcal{L} \) is a security level defined in our security lattice. \( \Sigma : \mathcal{Ref} \rightarrow \mathcal{L} \) is a relation that maps objects to security levels and is similar to structure security in [7]. \( \Lambda : \mathcal{N} \rightarrow \langle \mathcal{L}, \mathit{Id} \rangle \) is a security typing stack that maps integer indices \( \mathcal{N} \) to pairs \( \langle \mathcal{L}, \mathit{Id} \rangle \), where \( \mathit{Id} = \{ \mathit{LOOP}, \mathit{IF}, \mathit{TRY}, \mathit{WITH}, \mathit{FUNC}, \mathit{LBL} \} \) is a set of string constants that correspond to some of the statements and expressions defined in the JavaScript language and are used to label the entries in the security typing stack \( \Lambda \).

The program counter \( pc \) is always pointing to the top most element of the security typing stack \( \Lambda \), hence, \( pc = \text{tos}(\Lambda) = \Lambda(|\Lambda|) = \langle \mathcal{L}, \mathit{Id} \rangle \), where \( \text{tos()} \) means top-of-stack function and \( |\Lambda| \) is the cardinality of the security typing stack (length-of-stack). This means that, the program counter \( pc \) gets updated whenever the security typing stack \( \Lambda \) gets updated, e.g. by pushing an element onto the \( \Lambda \) stack or by removing an element from the \( \Lambda \) stack. We use the dot notation to access the elements of the pair pointed by the \( pc \), for example \( pc.l = \mathcal{L} \) and \( pc.id = \mathit{Id} \).

The idea is that \( \Gamma, \Sigma, \) and \( \Lambda \) represent the security levels of all the variables, literals, and primitive values in the current security context that holds before the execution of statement \( S \) and \( \Gamma', \Sigma', \) and \( \Lambda' \) represent the security levels of all the variables, literals, and primitive values after the execution of statement \( S \) in the current security context. The \( pc \) is used to eliminate indirect information flows [4] and the typing stack \( \Lambda \) is used to track information flow across function invocations and through nested control-flow statements.

The structure security level of an object \( o \) is the least upper bound of all the security levels associated with the properties defined in the domain of that object and the current
Table 4: Statement Typing Part(A)

<table>
<thead>
<tr>
<th>T-Block</th>
<th>H, r, pc ⊢ Γ, Σ, Λ(S₁)Γ₁, Σ₁, Λ₁</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H, r, pc ⊢ Γ₁, Σ₁, Λ₁(S₂)Γ₂, Σ₂, Λ²</td>
</tr>
<tr>
<td></td>
<td>⊢</td>
</tr>
<tr>
<td>T-Assign-Var</td>
<td>Γ, Σ, Λ, H, r ⊢ E : τ</td>
</tr>
<tr>
<td>T-Assign-Obj-Prop</td>
<td>H, r, pc ⊢ Γ, Σ, Λ{ x = E }Γ₁ = Γ(r)[x → pc.ℓ ∪ τ], Σ, Λ</td>
</tr>
<tr>
<td>T-Assign-Array-Index</td>
<td>Γ, Σ, Λ, H, r ⊢ E₁ : τ₁, E₂ : τ₂, x : τₓ (H, r, E₁) → (H’, r, m)</td>
</tr>
<tr>
<td>T-If-Stmt</td>
<td>Γ, Σ, Λ, H, r ⊢ E : τ</td>
</tr>
<tr>
<td>T-Output</td>
<td>H, r, pc ⊢ Γ, Σ, Λ{ output(E) }Γ, Σ, Λ</td>
</tr>
<tr>
<td>T-While</td>
<td>H, r, pc ⊢ Γ, Σ, Λ{ while(E) S }Γₓ, Σₓ, Λ</td>
</tr>
</tbody>
</table>

Do-while same as While rule

T-For-in | H, r, pc ⊢ Γ, Σ, Λ{ for(E₁ in E₂) S }Γₓ, Σₓ, Λ |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H, r, pc ⊢ Γ, Σ, Λ{ for(E = E₁ in E₂) S }Γₓ, Σₓ, Λ</td>
</tr>
<tr>
<td>T-For-var-in</td>
<td>H, r, pc ⊢ Γ, Σ, Λ{ for(x = E₁ in E₂) S }Γₓ, Σₓ, Λ</td>
</tr>
<tr>
<td>T-Assign-Obj-Prop</td>
<td>H, r, pc ⊢ Γ, Σ, Λ{ x = E(E) }Γ(r)[x → pc.ℓ ∪ τₓ], Σ, Λ</td>
</tr>
<tr>
<td>T-Func-Cal</td>
<td>H, r, pc ⊢ Γ, Σ, Λ{ x = E(E) }Γ(r)[x → pc.ℓ ∪ τₓ], Σ, Λ</td>
</tr>
</tbody>
</table>
Table 5: Statement Typing Part(B)

<table>
<thead>
<tr>
<th>Statement Type</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-Catch</td>
<td>$\frac{H, r, pc' \vdash \Gamma, \Sigma, \Lambda' {S} \Gamma', \Sigma', \Lambda, pc = \text{pop} (\Lambda')}{H, r, pc \vdash \Gamma, \Sigma, \Lambda {\text{catch} (x) (S)} \Gamma', \Sigma', \Lambda}$</td>
</tr>
<tr>
<td>T-Method-Call</td>
<td>$\frac{H, r, pc \vdash \Gamma, \Sigma, \Lambda' {\text{Method-Call} (\tilde{E}) } \Delta (\tilde{E}) (E'[\tilde{E}]) \Gamma (r) [x \mapsto \text{pc} \cup \tau_{\text{ext}}], \Sigma, \Lambda}{H, r, pc' \vdash \Gamma, \Sigma, \Lambda (x = E [E'[\tilde{E}]] (E') \tilde{r}) \text{pc} \mapsto \text{pc} \cup \tau_{\text{ext}}, \Sigma, \Lambda}$</td>
</tr>
<tr>
<td>T-Finally</td>
<td>$\frac{H, r, pc' \vdash \Gamma, \Sigma, \Lambda' {\text{finally} (S)} \Gamma', \Sigma', \Lambda}{H, r, pc \vdash \Gamma, \Sigma, \Lambda {\text{finally} (S)} \Gamma', \Sigma', \Lambda}$</td>
</tr>
<tr>
<td>T-Return</td>
<td>$\frac{pc' = \text{pop} (\Lambda') \Lambda, pc = \text{pop} (\Lambda')}{pc' \vdash \Gamma, \Sigma, \Lambda {\text{return} E} \Gamma', \Sigma', \Lambda''}$</td>
</tr>
<tr>
<td>T-Skip</td>
<td>$\frac{H, r, pc \vdash \Gamma, \Sigma, \Lambda {: } \Gamma', \Sigma, \Lambda}{pc' = \langle \text{pc} , \ell , _ \rangle}$</td>
</tr>
<tr>
<td>T-Cont-lbl, T-Brk-lbl</td>
<td>$\frac{pc' = \langle \text{pc} , \ell , _ \rangle, \ while (pc.id \neq \text{LBL}) { \Lambda', _ = \text{pop} (\Lambda') }}{H, r, pc \vdash \Gamma, \Sigma, \Lambda {\text{continue}</td>
</tr>
<tr>
<td>T-Continue, T-Break</td>
<td>$\frac{pc' = \langle \text{pc} , \ell , _ \rangle, \ while (pc.id \neq \text{LOOP}) { \Lambda', _ = \text{pop} (\Lambda') }}{H, r, pc \vdash \Gamma, \Sigma, \Lambda {\text{continue}</td>
</tr>
<tr>
<td>T-Throw</td>
<td>$\frac{H, r, pc \vdash \Gamma, \Sigma, \Lambda {\text{throw} E} \Gamma', \Sigma', \Lambda''}{H, r, pc' \vdash \Gamma, \Sigma, \Lambda' (pc' \ell \cup \tau _ _ _) \text{pc} \mapsto \text{pc} \cup \tau_{\text{ext}}}$</td>
</tr>
<tr>
<td>T-With</td>
<td>$\frac{H, r, pc' \vdash \Gamma, \Sigma, \Lambda' {\text{with} (E) (S)} \Gamma', \Sigma', \Lambda}{H, r, pc \vdash \Gamma, \Sigma, \Lambda {\text{with}(E) (S)} \Gamma', \Sigma', \Lambda} \text{pc} = \text{pop} (\Lambda')$</td>
</tr>
<tr>
<td>T-Try</td>
<td>$\frac{H, r, pc' \vdash \Gamma, \Sigma, \Lambda' {\text{try} (S)} \Gamma', \Sigma', \Lambda}{H, r, pc \vdash \Gamma, \Sigma, \Lambda {\text{try} (S)} \Gamma', \Sigma', \Lambda} \text{pc} = \text{pop} (\Lambda', n)$</td>
</tr>
</tbody>
</table>

Sub-Typing

$\begin{align*}
pc_1 & \vdash \Lambda, \Gamma_1(S) \Lambda, \Gamma'_1 \\
pc_2 & \vdash \Lambda, \Gamma_2(S) \Lambda, \Gamma'_2
\end{align*}$
Definition 3. New Object Literal Relation $\triangleright_{\text{ObjectLit}}$ is defined as follows:

\[
\begin{align*}
    \overline{pm} : \overline{\epsilon} = \{pn_1 : e_1, pn_2 : e_2 \ldots pn_n : e_n\}, \quad n = |\overline{pm} : \overline{\epsilon}|,
    \\
    \langle H, r, e_1 \rangle \rightarrow \langle H_1 = H(r)[pn_1 \mapsto va_1]\rangle,
    \\
    \vdots
    \\
    \langle H_{n-1}, r, e_n \rangle \rightarrow \langle H_n = H_{n-1}(r)[pn_n \mapsto va_n]\rangle,
    \\
    \tau_1 = H, r, pc \vdash \Gamma, \Sigma, \Lambda[e_1] \Gamma_1 = \Gamma(r)[pn_1 \mapsto \tau_1], \Sigma_1, \Lambda, \quad r_o \notin \text{dom}(H_n),
    \\
    H' = H_n \left[ r_o \mapsto \begin{cases} 
        \text{null} & \\
        \langle \text{proto}_o \mapsto \epsilon, \quad pm_i \mapsto va_i \rangle
    \end{cases} \right],
    \\
    \Gamma' = \Gamma_n \left[ r_o \mapsto \begin{cases} 
        \text{null} & \\
        \langle \text{proto}_o \mapsto \epsilon, \quad pm_i \mapsto pc \cup \tau_i \rangle
    \end{cases} \right],
    \\
    \Sigma' = \Sigma_n \left[ r_o \mapsto \bigcup_{i=1,2,n} \Gamma'(r_o)(pm_i) \right]
\end{align*}
\]

\[\langle \Gamma, \Sigma, \Lambda, pc, H, r, (\overline{pm} : \overline{\epsilon}) \rangle \triangleright_{\text{ObjectLit}} \langle \Gamma', \Sigma', \Lambda, pc, H', r \rangle\]

The structure security level of an array literal "[\overline{\epsilon}]" is the least upper bound of all the elements in the domain of that array as illustrated in Definition 4. Definition 4 defines relation $\triangleright_{\text{NewArray}}$ which models storing a new array literal "[\overline{\epsilon}]" in memory and tracks its security typing.

Definition 4. New Array Literal Relation $\triangleright_{\text{ArrayLit}}$ is defined as follows:

\[
\begin{align*}
    \overline{\epsilon} = \{e_0, e_1 \ldots e_{n-1}\}, \quad n = |\overline{\epsilon}|,
    \\
    \langle H, r, e_0 \rangle \rightarrow \langle H_0 = H(r)[0 \mapsto va_0]\rangle,
    \\
    \vdots
    \\
    \langle H_{n-2}, r, e_{n-1} \rangle \rightarrow \langle H_{n-1} = H_{n-2}(r)[n-1 \mapsto va_{n-1}]\rangle,
    \\
    \tau_0 = H, r, pc \vdash \Gamma, \Sigma, \Lambda[e_0] \Gamma_0 = \Gamma(r)[0 \mapsto \tau_0], \Sigma, \Lambda, \quad r_o \notin \text{dom}(H_{n-1}),
    \\
    \Gamma_{n-2}, \Sigma, \Lambda[e_{n-1}] \Gamma_{n-1} = \Gamma_{n-2}(r)[n-1 \mapsto \tau_{n-1}], \Sigma, \Lambda,
    \\
    H' = H_{n-1} \left[ r_o \mapsto \begin{cases} 
        \text{null} & \\
        \langle \text{proto}_o \mapsto \epsilon \rangle
    \end{cases} \right],
    \\
    \Gamma' = \Gamma_{n-1} \left[ r_o \mapsto \begin{cases} 
        \text{null} & \\
        \langle \text{proto}_o \mapsto \epsilon \rangle
    \end{cases} \right],
    \\
    \Sigma' = \Sigma \left[ r_o \mapsto pc \cup \bigcup_{i=0,1,2,n-1} \Gamma'(r_o)(i) \right]
\end{align*}
\]

\[\langle \Gamma, \Sigma, \Lambda, pc, H, r, [\overline{\epsilon}] \rangle \triangleright_{\text{ArrayLit}} \langle \Gamma', \Sigma', \Lambda, pc, H', r \rangle\]

Definition 5 defines relation $\triangleright_{\text{FuncLit}}$ which models storing a new function literal "function(\overline{x})\{P\}" in memory and tracks its security typing.

Definition 5. New Function Literal Relation $\triangleright_{\text{FuncLit}}$ is defined as follows:
The $\rhd_{\text{FuncLit}}$ relation stores the function body $P$ in an internal property $\@\text{code}$ and stores the formal parameters (if any) in $\@\text{fparam}$ internal property. When the function is invoked, the formal parameters get binded to the result of the evaluation of the function arguments. The function scope is stored in $\@\text{fscope}$ internal property which points to where the function is defined. Every function object stores a $\text{prototype}$ property which points to the prototype object of the function. Any properties added to the function prototype object will be accessible from all the objects instantiated from this function using the `new` operator. This is how the prototypical-based inheritance is implemented in the JavaScript language.

Definition 6 defines relation $\rhd_{\text{FuncCall}}$ which models the invocation of a function with arguments "$e(e)$" and tracks its security typing.

In any function invocation the first step is to evaluate the $e$ expression in $e(e)$ to a function reference $r_f$. The function reference is used to retrieve the formal parameters of the function being invoked. Every expression $e_i$ in the expression vector $\vec{e}$ is evaluated to a pure value $v_{a_i}$ (recall from Table 1 that pure values are either primitive values or references) and get binded to the corresponding formal parameter $x_i$. The relation tracks the security typing in the same manner where each formal parameter $x_i$ is binded to a security type $\tau_i$ in the typed environment $\Gamma$. A fresh reference $r'$ is created to point to the newly instantiated scope object in memory and also used to track the security typing in $\Gamma$ and the structure security typing $\Sigma$. A new security context entry is pushed into the security context stack $\Lambda$ taking the least upper bound of the security levels of the current security context $pc$ and the scope where this function was defined. The $\@\text{this}$ internal property of the new scope object is assigned the $\#\text{Global}$ value since the function is invoked on the global scope.
Definition 6. Function Call Relation $\triangleright_{\text{FuncCall}}$ is defined as follows:

$$\langle \Gamma, \Sigma, \Lambda, pc, H, r, e \rangle \triangleright_{\text{FuncCall}} \langle \Gamma_0, \Sigma_0, \Lambda, pc, H_0, r_f \rangle$$

$\vec{x} = H(r_f)(\@fparam)$, $\vec{x} = \{x_1, x_2, ..., x_n\}$

$P = H(r_f)(\@code)$, $r_f^\gamma = H(r_f)(\@fscope)$, $\vec{e} = \{e_1, e_2, ..., e_n\}$

$\langle H_0, r, e_1 \rangle \rightarrow \langle H_1 = H_0(r)[x_1 \mapsto va_1], \vec{x} = \{x_1, x_2, ..., x_n\} \rangle$

$\vdots$

$\langle H_{n-1}, r, e_n \rangle \rightarrow \langle H_n = H_{n-1}(r)[x_n \mapsto va_n], r \rangle$, $\tau_0 = H_0, r, pc \vdash \Gamma, \Sigma, \Lambda\{e_1\} \Gamma_0 = \Gamma(r)[x_1 \mapsto \tau_1], \Sigma, \Lambda,$

$\vdots$

$\tau_n = H_{n-1}, r, pc \vdash \Gamma_{n-1}, \Sigma, \Lambda\{e_n\} \Gamma_n = \Gamma_{n-1}(r)[x_n \mapsto \tau_n], \Sigma, \Lambda,$

$r' \not\in \text{dom}(H_n)$, $\Lambda', pc' = \text{push}(\langle pc.\ell \cup \Gamma(r_f)(\@fscope) \cup \Sigma(r_f), \text{FUNC} \rangle)$

$H' = H_n \left[ \begin{array}{l} r' \mapsto \langle \text{scope} \mapsto r_f^\gamma, \\
\quad \text{this} \mapsto \#\text{Global}, \\
\quad x_1 \mapsto va_1, ..., x_n \mapsto va_n \rangle \end{array} \right],$

$\Gamma' = \Gamma_n \left[ \begin{array}{l} r' \mapsto \langle \text{scope} \mapsto pc', \\
\quad \text{this} \mapsto \Gamma(\#\text{Global}), \\
\quad x_1 \mapsto \tau_1, ..., x_n \mapsto \tau_n \rangle \end{array} \right],$

$\Sigma' = \Sigma[r' \mapsto pc']$

$\langle \Gamma', \Sigma', N', pc', H', r', va \rangle \rightarrow \langle \Gamma, \Sigma, N', pc', H, r, va \rangle$

Definition 7 defines relation $\triangleright_{\text{MethodCall}}$ which models the calling of a property method with arguments "$e[e'](\vec{e})$" and tracks its security typing. When functions are assigned as properties to objects they are called property methods rather than functions. The main difference between function calls and property method calls is what the hidden property "$\@\text{this}$" is pointing to in the new scope object. As shown previously in Definition 6, the $\@\text{this}$ is assigned the value $\#\text{Global}$ which means it is pointing to the global object, however when the function is invoked as a property method the $\@\text{this}$ points to the object which contains the invoked method as a property. The other difference is the value of what get pushed on the security context stack; in Definition 6 it was the least upper bound of the current security context $pc$ and the function scope; in the case of method invocation, it adds the object’s (structure) security level $\Sigma(r_m)$ as well.
Definition 7. Method Call Relation $\triangleright_{MethodCall}$ is defined as follows:

$\langle \Gamma, \Sigma, \Lambda, pc, H, r, e \rangle \rightarrow \langle \Gamma_0, \Sigma_0, \Lambda, pc, H_0, r_0 \rangle, r_0 \neq \text{null}$

$\langle \Gamma_0, \Sigma_0, \Lambda, pc, H_0, r, e' \rangle \rightarrow \langle \Gamma_1, \Sigma_1, \Lambda, pc, H_1, m \rangle$

$\langle H_1, r_0, m \rangle \triangleright_{Proto} r_m, \ r_f = H_1(r_m)(m)$

$\tilde{x} = H_1(r_f)(\text{@fparam}), \tilde{x} = \{x_1, x_2, \ldots, x_n\}$

$P = H_1(r_f)(\text{@code}), \ r_f = H_1(r)(\text{@scope}), \ e = \{e_1, e_2, \ldots, e_n\}$

$\langle H_1, r, e_1 \rangle \rightarrow \langle H_2 = H_1(r)[x_1 \mapsto \text{va}_1], \ r_0 \rangle$

$\ldots$

$\langle H_n, r, e_n \rangle \rightarrow \langle H_{n+1} = H_n(r)[x_n \mapsto \text{va}_n] \rangle$

$\tau_0 = H_1, r, pc \vdash \Gamma_1, \Sigma, \Lambda\{e_1\} \Gamma_2 = \Gamma_1(r)[x_1 \mapsto \tau_1], \Sigma, \Lambda$

$\ldots$

$\langle \hat{\Gamma}, \hat{\Sigma}, \Lambda', \hat{pc}, \hat{H}, \text{va} \rangle$

Definition 8 defines relation $\triangleright_{ConstrFunc}$ which models the instantiation of an object using a constructor function with arguments "new $\text{e}(\bar{e})$" and tracks its security typing.
Definition 8. Constructor Function Call Relation $\triangleright_{\text{ConstrFunc}}$ is defined as follows:

$$
\begin{align*}
\langle \Gamma, \Sigma, \Lambda, pc, H, r, e \rangle &\rightarrow \langle \Gamma_0, \Sigma_0, \Lambda, pc, H_0, rf \rangle \\
\vec{x} &= H(rf)(\vec{f}_{\text{param}}), \ \vec{x} = \{x_1, x_2, ..., x_n\} \\
P &= H(rf)(\vec{f}_{\text{scope}}), \ r_f' = H(rf)(\vec{f}_{\text{scope}}), \ e = \{e_1, e_2, ..., e_n\} \\
\langle H_0, r, e_1 \rangle &\rightarrow \langle H_1 = H_0(r)[x_1 \mapsto va_1] \rangle, \\
&\vdots \\
\langle H_{n-1}, r, e_n \rangle &\rightarrow \langle H_n = H_{n-1}(r)[x_n \mapsto va_n], r \rangle, \\
\tau_0 &= H_0, r, pc \vdash \Gamma, \Sigma, \Lambda(e_1) \Gamma_0 = \Gamma(r)[x_1 \mapsto \tau_1], \Sigma, \Lambda, \\
&\vdots \\
\tau_n &= H_{n-1}, r, pc \vdash \\
\Gamma_{n-1}, \Sigma, \Lambda(e_n) \Gamma_n &= \Gamma_{n-1}(r)[x_n \mapsto \tau_n], \Sigma, \Lambda, \\
\tau_r, \tau_o \notin \text{dom}(H_n), \\
\Lambda', pc' &= \text{push}(\langle pc, \ell \cong \Gamma_n(rf)(\vec{f}_{\text{scope}}) \cup \Sigma(rf), \text{FUNC} \rangle), \\
H' &= \Gamma_{n-1} \left[ \begin{array}{c} \\
\ell \cong \Gamma_n(rf)(\vec{f}_{\text{scope}}) \cup \Sigma(rf), \text{FUNC} \\
\end{array} \right], \\
\Gamma' &= \Gamma_{n-1} \left[ \begin{array}{c} \\
\ell \cong \Gamma_n(rf)(\vec{f}_{\text{scope}}) \cup \Sigma(rf), \text{FUNC} \\
\end{array} \right], \\
\Sigma' &= \Sigma[r \mapsto pc'], \\
\langle \Gamma', \Sigma', \Lambda', pc', H', r', P \rangle &\rightarrow \langle \Gamma, \Sigma, \Lambda, pc', H', r, va \rangle \\
\tau &= pc'.\ell, \Lambda, pc = \text{pop}(\Lambda'), \\
\hat{\nu} &= \begin{cases} \\
va, \text{isObj}(va) = \text{true} & \tau_0 = \begin{cases} \\
\Sigma(va) \cup \tau, \text{isObj}(va) = \text{true} & \\
\Sigma(r_o), \text{otherwise} & \\
\Gamma'' = \Gamma(r)(\vec{f} \mapsto \langle \text{proto} \mapsto \tau_o \cup \Gamma(r)(\text{prototype}), ... \rangle) \\
\Sigma'' = \Sigma[\hat{\nu} \mapsto \tau_0], \ H'' = H(r)(\vec{f} \mapsto \langle \text{proto} \mapsto H(rf)(\text{prototype}), ... \rangle) \\
\end{cases} \\
\end{cases} \\
\langle \Gamma, \Sigma, \Lambda, pc, H, r, \text{new e}(\hat{\nu}) \rangle &\triangleright_{\text{ConstrFunc}} \langle \Gamma'', \Sigma'', \Lambda, pc, H'', \hat{\nu} \rangle
\end{align*}
$$

Constructor functions are used in JavaScript to instantiate objects and to implement inheritance. When a function is invoked with the \texttt{new} operator it is called a constructor function and the result of the evaluation of the \texttt{new e(\hat{\nu})} expression is a reference to a newly created object in memory. If the function’s return value \footnote{Every function returns a value in JavaScript even functions with no explicit return statements, in which case the return value is \texttt{undefined}.} \texttt{2} is a reference to an object, this reference takes precedence over the one created by the runtime and becomes the return value of the \texttt{new e(\hat{\nu})} expression. The $\triangleright_{\text{ConstrFunc}}$ relation accounts for this fact by checking the type of the return value $va$ of the constructor function using the \texttt{isObj()} function (recall from Table 1 that $va$ is either a primitive type or a reference to an object). The \texttt{isObj()} function returns \texttt{true} when the $va$ is pointing to any type that is not a primitive type, formally, \texttt{isObj(va) = true, iff va \notin \{n, m, b, null, undefined\}}.

Definition 9 defines relation $\triangleright_{\text{ConMeth}}$ which models the instantiation of an object using a constructor method with arguments "$\texttt{new e(e^{1})}(\hat{\nu})$" and tracks its security typing. The relation $\triangleright_{\text{ConMeth}}$ defined below operates similarly to relation $\triangleright_{\text{ConstrFunc}}$ with minor differences. The main difference is the structure security level $\Sigma(r_m)$ of the object that owns the constructor method $rf$ is accounted for when taking the least upper bound of the current security context $pc$ and the scope where the function was defined $\Gamma(rf)(\vec{f}_{\text{scope}})$.

In the typing rules we write $\Gamma, \Sigma, \Lambda, H, r \vdash \Sigma : \tau$ to mean that expression $\Sigma$ has type $\tau$ assuming type environment $\Gamma$, structure security $\Sigma$, security context stack $\Lambda$, heap memory $H$, and $r$ is a reference to the current scope object. The type $\tau$ is defined as a recursive type...
relation as outlined in Table 6. To simplify the typing rules, we do not distinguish between recursive types and their un-foldings following the *equi-recursive* approach outlined in [13].

**Definition 9.** Constructor Method Call Relation \(\triangleright_{\text{ConMeth}}\) is defined as follows:

\[
\begin{align*}
\langle \Gamma, \Sigma, \Lambda, pc, H, r, e \rangle &\triangleright (\Gamma_0, \Sigma_0, \Lambda, pc, H_0, r_0), r_o \neq \text{null} \\
(\Gamma_0, \Sigma_0, \Lambda, pc, H_0, r, e^\ast) &\triangleright (\Gamma_1, \Sigma_1, \Lambda, pc, H_1, m) \\
(H_1, r_o, m) &\triangleright_{\text{proto}} r_o, r_f = H_1(r_m)(m) \\
\vec{x} = H_1(r_f)(\overline{fparam}), \vec{x} = \{x_1, x_2, \ldots x_n\} &
\end{align*}
\]

\[
\begin{align*}
P = H_1(r_f)(\overline{fcode}), r_f = H_1(r_f)(\overline{fscope}), \vec{e} = \{e_1, e_2, \ldots e_n\} \\
(H_1, r, e_1) &\triangleright (H_2 = H_1(r)[x_1 \mapsto va_1]), ...
\end{align*}
\]

\[
\begin{align*}
(H_n, r, e_n) &\triangleright (H_{n+1} = H_n(r)[x_n \mapsto va_n]), \\
\tau_0 = H_1, r, pc \vdash \Gamma_1, \Sigma_1, \Lambda\{e_1\} \Gamma_2 = \Gamma_1(r)[x_1 \mapsto \tau_1], \Sigma_1, \Lambda, \\
\tau_n = H_n, pc \vdash \Gamma_n, \Sigma_1, \Lambda\{e_n\} \Gamma_{n+1} = \Gamma_n(r)[x_n \mapsto \tau_n], \Sigma_1, \Lambda, \\
r', r_o \not\in \text{dom}(H_{n+1}), \\
\Lambda', pc' = \text{push}(pc.f \cup \Gamma_{n+1}(r_f)(\overline{fscope}) \cup \Sigma_1(r_m) \cup \Sigma_1(r_n).\text{FUNC}), \\
H' = H_{n+1} \begin{bmatrix} r' & \text{@scope} \mapsto r_f, & \text{@this} \mapsto r_o, \\
& x_1 \mapsto va_1, \ldots, x_n \mapsto va_n & \end{bmatrix}, \\
\Gamma' = \Gamma_{n+1} \begin{bmatrix} r' & \text{@scope} \mapsto pc', & \text{@this} \mapsto pc', \\
& x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n & \end{bmatrix}, \\
\Sigma' = \Sigma_1[r' \mapsto pc'] \\
\langle \Gamma', \Sigma', \Lambda', pc', H', r', P \rangle &\triangleright (\Gamma, \Sigma, \Lambda, pc, H, r, \vec{e}[\sigma]) \triangleright_{\text{ConMeth}} \langle \Gamma''', \Sigma'', \Lambda, pc, H''', \dot{\sigma} \rangle
\end{align*}
\]

The previously defined relations and the typing rules in Tables 4 and 5 use two auxiliary functions, namely, push() and pop(). Both functions operate on the security context stack \(\Lambda\). The \(\Lambda', pc' = \text{push}(\Lambda, arg_1, arg_2, ...)\) function pushes one or more arguments onto the \(\Lambda\) stack returning a reference to the modified stack, consequently the program counter \(pc\) is going to be modified since it always points to the top-most element of the stack. The \(\Lambda, x = \text{pop}(\Lambda')\) function pops the top-most element from the security context stack \(\Lambda\) returning a reference to the modified stack and the value of the \(n^{th}\) element pointed to by \(x\). When the value returned from the \text{pop()} function is not needed we use the "." symbol to mean the value is discarded.

Based on the rules in Tables 6, 4, and 5 the type system satisfies a straightforward confidentiality condition. An attacker observing outputs from channel with type \(\tau\) should only be able to observe inputs with types \(\subseteq \tau\). More precisely, given a derivation \(\vdash \Lambda, \Sigma, \Gamma \{\text{output}_{E}(E)\} \Lambda, \Sigma, \Gamma\), the final value of an expression \(E\) with final type \(\vdash E : \tau\) will at most depend on initial values of variables \(y : \tau_1\), literals \(\text{lit} : \tau_2\), and primitive values \(pv : \tau_3\) with initial types \(\tau_1 \sqcup \tau_2 \sqcup \tau_3 \subseteq \tau\), where \(\sqcup\) is the least upper bound operator.
Table 6: Expression Typing.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Σ, Γ, Λ, pc, H, r ⊢ E : τ</td>
<td>( \Gamma(r) \bowtie \text{prop} \rightarrow \mathcal{L} )</td>
</tr>
<tr>
<td>( e^0_{\text{po}} )</td>
<td>Σ, Γ, Λ, H, r ⊢ e : τ</td>
</tr>
<tr>
<td>( e^0_{\text{inc} e_2} )</td>
<td>Σ, Γ, Λ, H, r ⊢ e : τ</td>
</tr>
<tr>
<td>( e([\text{e}']) )</td>
<td>pc'.\ell → L, iff , ( \langle \ldots, pc, \mathbf{e}(\mathbf{s}) \rangle \bowtie \text{FuncCall} \langle \ldots, pc', va \rangle )</td>
</tr>
<tr>
<td>( e'([\text{e}']) )</td>
<td>pc'.\ell → L, iff , ( \langle \ldots, pc, e(\ldots) \text{e}(\ldots) \rangle \bowtie \text{MethodCall} \langle \ldots, pc', va \rangle )</td>
</tr>
<tr>
<td>( \text{new } e'([\text{e}']) )</td>
<td>Σ''(\text{e}) → L, iff , ( \langle \ldots, \Sigma, \text{new } e(\ldots) \rangle \bowtie \text{ProtCall} \langle \ldots, \Sigma'', v \rangle )</td>
</tr>
<tr>
<td>( \text{function } ([\text{e}']) ([\text{P}]) )</td>
<td>Σ'(\text{r}) → L, iff , ( \langle \ldots, \Sigma, \ldots \rangle \bowtie \text{FuncLit} \langle \ldots, \Sigma', \text{f} \rangle )</td>
</tr>
<tr>
<td>( x[e] )</td>
<td>( \Gamma(r') (m) \cup \Sigma(r_0) \rightarrow L, \text{iff} ) ( \langle \ldots, H, e \rangle \rightarrow \langle \ldots, H', m \rangle ) ( &amp; \langle H', r, x \rangle \bowtie \text{Proto} \langle \ldots, H(r') \rangle )</td>
</tr>
<tr>
<td>( x[e] )</td>
<td>( \Sigma(r_0) \rightarrow L, \text{iff} ) ( \langle H, r, x \rangle \bowtie \text{Proto} \langle \ldots, H', m \rangle ) ( &amp; \langle H', r_0, m \rangle \bowtie \text{Proto} \langle \ldots, \rangle ) ( &amp; r' = \text{null} )</td>
</tr>
</tbody>
</table>
3 Hybrid Flow-Sensitive Security Monitor For JavaScript

In this section, we present a sound flow-sensitive monitor by combining static analysis of the type system introduced in the previous section and the dynamic flow-sensitive (monitoring) techniques. The static analysis of the type system is augmented to take into account the control-flow branches that have not been taken in the current run.

3.1 VM Monitor

In order for any security monitor to observe a given virtual machine (VM), the VM (the running program) should generate events that are visible to the VM monitor. In this regard, we augment the list of statements outlined in table 3 with the statements outlined in Table 7. The new statements (commands) are similar to the ones defined in [2]:

- □: statement with no transition; it is used to serve as a terminal configuration in the semantics.
- ⋊: generates ”join-of-branch” event $j$ that makes the VM monitor synchronize with joint points in the control-flow of the running program.
- ✓: generates ”step” event $t$. The ✓ statement never appears in the source code, hence, the VM monitor does not need to handle its $t$ event. It is used in the inlining of the VM monitor to prove some correctness properties about the inlined version of the VM monitor.

Tables 9 and 10 present the JavaScript statement transitions and the associated events that get generated and allow the monitor to observe the behaviour of the VM. The events are outlined in Table 8. All the events are internal and can only be observed by the VM monitor (e.g. can’t be observed by an attacker) with the exception of the $o_e(e, va)$ event which generates an observable output event $w$ as previously discussed in subsection 2.4. The transition events outlined in Table 8 consist of variable assignment event $a_v(x, e)$, property assignment event $a_p(x.m, e)$, array index (or object property if $x$ is an object not an array) assignment event $a_i(x[m], e)$, compensate event $k(lbl)$, branching event $b(e, S)$ and its variant $b(\vec{e}, S)$, the ”skip” event $sk$, and the join-of-branch event $j$.

The VM monitor transitions combine information from the flow-sensitive type system and the generated events from the running program (VM) and have the following form:

$$\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r \rangle \rangle \xrightarrow{\alpha} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc \rangle, \langle H', r \rangle \rangle$$

The VM monitor runs alongside the running program (VM) in a lockstep fashion. At each statement transition of the running program, the VM monitor inspects the generated event $\alpha$, updates its state, and generates $\gamma$ which is either an observable output event $w$ or nothing.
s ::= s \quad \text{% previously presented in Table 3} \\
| \text{output}_\ell(e) \quad \text{% output statement} \\
| \text{☐} \quad \text{(strict)} \quad \text{✓} \quad \text{% additional statements} \\

Table 7: Additional statements for monitored execution.

| Statements events \( \beta \) | ::= | \alpha \mid t \\
|---|---|---|---|---|
| Statements events \( \alpha \) | ::= | \epsilon \mid o_\ell(e,va) \\
|---|---|---|---|---|
| Statements events \( \epsilon \) | ::= | a_v(x,e) \mid a_p(x,m,e) \\
|---|---|---|---|---|
| Statements events \( a_i(x[m],e) \mid k(lbl) \\
|---|---|---|---|---|
| Statements events \( b(e,S) \mid b(\vec{e},S) \\
|---|---|---|---|---|
| Statements events \( sk \mid j \\

Observations \( w \ ::= o_\ell(va) \\

Table 8: Statements transitions events.

All the symbols in the VM monitor configuration have been introduced before with the exception of the \( \Delta \) symbol, which is a stack of pairs \( \langle \vec{x}, \ell \rangle \). The \( \vec{x} \) is a list of variables and \( \ell \) is a security level. The control-flow stack \( \Delta \) is used to capture the security level \( \ell \) and variables \( \vec{x} \) of a control-flow branching point. The \( \vec{x} \) list corresponds to the variables and functions that could have been updated or executed in the branch that has not been taken in the current run, and \( \ell \) is the security level of the conditional expression of the control-flow statement.

**Definition 10. (Monitored Transitions)** The transition relation \( \gamma \rightarrow \) on monitored configuration of the running program is defined as follows:

\[
\langle H, r, s \rangle \xrightarrow{\alpha} \langle H', r, s' \rangle, \\
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r \rangle \rangle \xrightarrow{\alpha} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc \rangle, \langle H, r \rangle \rangle \\
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, s \rangle \rangle \xrightarrow{\gamma} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc \rangle, \langle H', r, s' \rangle \rangle
\]

Similar to [2] we write:

\[
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, s \rangle \rangle \xrightarrow{w} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc \rangle, \langle H', r, s' \rangle \rangle
\]

to mean

\[
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, s \rangle \rangle \xrightarrow{\ast} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc \rangle, \langle H', r, s' \rangle \rangle
\]

and we write

\[
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, s \rangle \rangle \xrightarrow{\vec{w}} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc \rangle, \langle H', r, s' \rangle \rangle
\]

to mean the transitive closure and the \( \vec{w} \) for the concatenation of the labeled transitions.

Table 11 presents the VM monitor transitions. The labeled transitions of the VM monitor make use of the following functions:
• $\vec{x} = \text{collect}(S)$: takes a statement $S$ as input and returns a list of variables $\vec{x}$ as output. The $\vec{x}$ list contains all the variables that appear in the left hand side (lhs) of the assignment statements in $S$ and all the functions that could be called in $S$. The definition of the collect function is outlined in Table 12. The collect function uses a helper function $\text{colFunc}(e)$ that collects function calls in expression $e$. The collect function is used in rule M-Branch outlined in Table 11.

• $\Gamma', \Sigma' = \text{upgrade}(\vec{x}, \ell, \Gamma, \Sigma, H, r)$: upgrades variables in list $\vec{x}$ to the security level $\ell$. The upgrade function is used in rule M-Join outlined in Table 11. The upgrade function is defined as follows:

\[
\begin{align*}
\text{proc upgrade}(\vec{x}, \ell, \Gamma, \Sigma, H, r) & \equiv \\
& \langle \vec{x} = \{x_1, x_2, \ldots, x_n\}, \langle H, r, x_i \triangleright \text{scope } r_x \rangle \rangle; \\
& \text{begin} \\
& \quad \text{for } (i := 1 \text{ to } n \text{ step } 1) \text{ do} \\
& \quad \quad \text{if } (\text{isFunc}(x_i) = \text{true}) \\
& \quad \quad \quad \quad \Gamma(r_x)[x_i \mapsto [\text{@f} \text{scope } \triangleright \Gamma(r_x)({@f} \text{scope}) \sqcup \ell]]; \\
& \quad \quad \quad \quad \Sigma(r_x)[x_i \mapsto \Sigma(r_x)(x_i) \sqcup \ell]; \\
& \quad \quad \text{else if } (\text{isObj}(x_i) = \text{true}) \\
& \quad \quad \quad \quad \Sigma(r_x)[x_i \mapsto \Sigma(r_x)(x_i) \sqcup \ell]; \\
& \quad \quad \text{else} \\
& \quad \quad \quad \quad \Gamma(r_x)[x_i \mapsto \Gamma(r_x)(x_i) \sqcup \ell]; \\
& \quad \fi \\
& \quad \fi \\
& \text{end} \\
& \text{end.}
\end{align*}
\]

Every M-Join transition performed by the VM monitor corresponds to a M-Branch transition, hence, there is one upgrade() function call that corresponds to one collect() function call. Consider the example in Figure 2 which is an adaptation of the flow-sensitivity attack example in [14] . The VM monitor will branch on the first If-Statement, therefore, it will collect the variables or functions that could possibly be modified or invoked in the untaken branch. In the case when the higher security level variable secret is false, the else branch of the If-Statement should be executed, hence, collecting the lower security level variable flag (or, in case of the code in the right hand side, function f()) from the then branch of the If-Statement. This corresponds to rule If-Else in Figure 9. When the VM (running program) executes the else branch and generates the join event as per rule End in Figure 10, the VM monitor will upgrade the variables (and the functions’ ”@scope” property) that have been collected previously, hence, upgrading the security class of variable flag (or function f() ”@scope” property) to be as high as the secret variable. In the case of the code in the right hand side, when the f() function is executed the pc.ℓ will be as high as the secret variable per Definition 6, hence, the assignment of the flag variable inside the function body will upgrade the security level of flag to be as high as the secret variable.
Figure 2: Two examples of the flow-sensitivity attack that demonstrate how the `collect()` and `upgrade()` functions can be used.

The same scenario will happen with the second If-Statement in Figure 2, resulting in the upgrade of the security level of variable `low` to be as high as `flag`, which implicitly means it will be as high as the `secret` variable, hence, preventing information leakage.

- $\Lambda' = \text{comp}_{\Lambda}(\text{lbl}, \ell)$: compensates for the existence of JavaScript statements that can change the control-flow in a non-block structured way, such as `return` and `throw` statements. Our hybrid monitor compensates for the existence of these types of statements by updating the entries of the security context stack $\Lambda$ with security level $\ell$. The updating starts from the inner most context, until the context above the one labeled `lbl`. The $\text{comp}_{\Lambda}(\text{lbl}, \ell)$ function is defined as follows:

\[
\text{proc comp}_{\Lambda}(\text{lbl}, \ell) \equiv \\
\begin{align*}
\text{var } i & := |\Lambda|; \\
\text{begin } & \\
& \text{while } (i > 1 \land \Lambda(i).id \neq \text{lbl}) \text{ do} \\
& \quad i := i - 1; \\
& \quad \Lambda(i).\ell := \Lambda(i).\ell \sqcup \ell; \\
& \text{end} \\
& i := i - 1; \\
& \Lambda(i).\ell := \Lambda(i).\ell \sqcup \ell; \\
\text{end.}
\end{align*}
\]

Non-block structured control-flow is especially challenging, for example, a function can have a return statement located arbitrarily in the function body. If an attacker combines
Figure 3: Implicit information flow using block structured control-flow in the left side and using non-block structured control-flow in the right side.

return statement (or throw statement) with an If-Statement in a function, she can make use of the conditional return (or conditional throw) to deduce the value of the conditional expression of the If-Statement. Consider the following example: the code in Figure 3 deduces the value of a higher security level variable by conditionally updating the value of a lower level variable inside a function. The function updates the lower security level variable in a way that an implicit flow is happening from the higher security level variable \texttt{high} to the lower security level variable \texttt{low}. In Figure 3, two examples are given achieving same result of implicit flow. The first example on the left side uses return statement and the one on the right side uses throw statement.

The VM monitor supports three different policies, namely, OutputFailStop, OutputSuppress, and OutputDefault, when it comes to how the VM monitor deals with insecure outputs. The OutputFailStop policy means that the monitor either will allow the output command to execute or diverge (fail stop), this is outlined in rule M-OutputFailStop in Table 11. The OutputSuppress policy means that the monitor either will allow the output command to execute or completely suppress the output, this is outlined in rule M-OutputSuppress in Table 11. The OutputDefault policy means that the monitor either will allow the output command to execute or will output a fixed literal value \textit{D}, this is outlined in rule M-OutputDefault in Table 11.

3.2 Attack Model and Security Property

Our attack model is similar to [14] and [2]. The attacker can only supply the program that is subject to monitoring and can provide the non-secret inputs that are modelled as the initial values of low security level variables. The attacker cannot observe the memory directly in any way, and cannot control or influence the behaviour of the running program. The attacker cannot observe the internal events generated by the running program. The attacker can only observe outputs that are equal or lower to her security level \(\ell\). We assume the attacker cannot change her security level during the execution of the running program and she cannot observe power consumption, time, or any other kind of covert channels.

We believe that this attack model is equivalent to the kind of attacks that target the client-side of the web, for example, cross-site scripting (XSS) attacks. A cross-site scripting

\footnote{It is important to note here that, if the thrown exception is not handled by a try-catch statement, the exception will keep propagating until it reaches the global scope and the running program will terminate.}
Table 9: JavaScript operational semantics with events Part (A).

<table>
<thead>
<tr>
<th>Event</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip</td>
<td>$\langle H, r, s \rangle \xrightarrow{\delta_{\text{Skip}}} \langle H, \square \rangle$</td>
</tr>
<tr>
<td>Assign-Var</td>
<td>$(H, r, x = e) \xrightarrow{a_{\text{x=x}}(x, e)} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>Assign-Obj-Prop</td>
<td>$(H, r, x = e) \xrightarrow{a_{\text{x=y}}(x, y, e)} \langle H'[r_x][y \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>Assign-Obj-Prop</td>
<td>$(H, r, [x = e_2]) \xrightarrow{a_{\text{[x=m,e_1,e_2]}}(x, m, e_1, e_2)} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>Assign-Array-Index</td>
<td>$(H, r, x[e_1] = e_2) \xrightarrow{a_{\text{[x=m,e_1,e_2]}}(x, m, e_1, e_2)} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>If-Then</td>
<td>$(H, r, c) \xrightarrow{\text{if}(e) \text{ S else } S_2} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>If-Else</td>
<td>$(H, r, c) \xrightarrow{\text{if}(e) \text{ S else } S_2} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>While-Loop</td>
<td>$(H, r, while(e) \text{ S}) \xrightarrow{b_{\text{while,skip}}(e, \text{skip})} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>While-Skip</td>
<td>$(H, r, while(e) \text{ S}) \xrightarrow{b_{\text{while,skip}}(e, \text{skip})} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>For-Loop</td>
<td>$(H, r, for(e_1; e_2; e_3) \text{ S}) \xrightarrow{b_{\text{for,skip}}(e_2, \text{skip})} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>For-Skip</td>
<td>$(H, r, for(e_1; e_2; e_3) \text{ S}) \xrightarrow{b_{\text{for,skip}}(e_2, \text{skip})} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>For-var-in-Loop</td>
<td>$(H, r, for(x = e_1 \text{ in } e_2) \text{ S}) \xrightarrow{b_{\text{for,skip}}(e_2, \text{skip})} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>For-var-in-Skip</td>
<td>$(H, r, for(x = e_1 \text{ in } e_2) \text{ S}) \xrightarrow{b_{\text{for,skip}}(e_2, \text{skip})} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>For-in-Loop</td>
<td>$(H, r, for(x \text{ in } e) \text{ S}) \xrightarrow{b_{\text{for,skip}}(e, \text{skip})} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
<tr>
<td>For-in-Skip</td>
<td>$(H, r, for(x \text{ in } e) \text{ S}) \xrightarrow{b_{\text{for,skip}}(e, \text{skip})} \langle H'[r_x][r \mapsto \text{val}], \square \rangle$</td>
</tr>
</tbody>
</table>

$S' \neq \square$
Table 10: JavaScript operational semantics with events Part (B).

For-in-Skip $\langle H, r, e \rangle \rightarrow \langle H', r, o \rangle$, isObj$(r_o) = false$

$\langle H, r, for(x in e) S \rangle \xrightarrow{b(e,S)} \langle H', x \rangle$

Func-Call $\langle H, r, e(\mathfrak{a}) \rangle \xrightarrow{\text{FuncCall} \ (H, va), \ (H, r, x) \xrightarrow{\text{Scope} r_x}} \langle H, r, x=e(\mathfrak{a}) \rangle$

Meth-Call $\langle H, r, e[e'](\mathfrak{a}) \rangle \xrightarrow{\text{MethCall} \ (H, va), \ (H, r, x) \xrightarrow{\text{Scope} r_x}} \langle H, r, x=e[e'](\mathfrak{a}) \rangle$

Return $\langle H, r, e \rangle \rightarrow \langle H', r, va \rangle$

Label-Stmt $\langle H, r, lbl:S \rangle \xrightarrow{k(func)} \langle H, S \rangle$

Continue-lbl $\langle H, r, lbl; \rangle \xrightarrow{k(loop)} \langle H, \square \rangle$

Break-lbl $\langle H, r, break lbl; \rangle \xrightarrow{k(loop)} \langle H, \square \rangle$

Throw $\langle H, r, e \rangle \rightarrow \langle H', r, va \rangle$

Try $\langle H, r, try\{ \} \rangle \xrightarrow{k\text{or func}} \langle H', r' \rangle$

Catch $\langle H, r, catch(x)\{ \} \rangle \xrightarrow{k\text{or func}} \langle H, r, S \rangle$

Finally $\langle H, r, finally\{ \} \rangle \xrightarrow{k} \langle H, r, S \rangle$

Output $\langle H, r, output, e \rangle \xrightarrow{\alpha(e, va)} \langle H', \square \rangle$

End $\langle H, r, x \rangle \xrightarrow{j} \langle H, \square \rangle$
Table 11: VM Monitor Transitions.

<table>
<thead>
<tr>
<th>Transition Type</th>
<th>Transition Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-Skip</td>
<td>$\langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r) \rangle \xrightarrow{\downarrow} \langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r) \rangle$</td>
</tr>
<tr>
<td>M-Assign-Var</td>
<td>$\langle \Gamma, \Sigma, \Lambda, pc, H, r \vdash e : \tau, \ (H, r, x) \triangleright_{\text{Scope}} r_x \rangle \xrightarrow{a_{\nu(x,e)}} \langle \Gamma', \Sigma, \Lambda, \Delta, pc, (H, r) \rangle$</td>
</tr>
<tr>
<td>M-Assign-Obj-Prop</td>
<td>$\langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r) \rangle \xrightarrow{a_{\nu(x,e)}} \langle \Gamma', \Sigma, \Lambda, \Delta, pc, (H, r) \rangle$</td>
</tr>
<tr>
<td>M-Assign-Obj-Prop</td>
<td>$\langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r) \rangle \xrightarrow{a_{\nu(x,y,e)}} \langle \Gamma', \Sigma, \Lambda, \Delta, pc, (H, r) \rangle$</td>
</tr>
<tr>
<td>M-Assign-Array-Index</td>
<td>$\langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r) \rangle \xrightarrow{a_{\nu(x,m),e_1,e_2}} \langle \Gamma', \Sigma, \Lambda, \Delta, pc, (H, r) \rangle$</td>
</tr>
<tr>
<td>M-Branch</td>
<td>$\langle \Gamma, \Sigma, \Lambda, pc, H, r \vdash e : \tau, \ (H, r, x) \triangleright_{\text{Scope}} r_x \rangle \xrightarrow{b(e,S)} \langle \Gamma, \Sigma, \Lambda, \Delta', pc, (H, r) \rangle$</td>
</tr>
<tr>
<td>M-Branch</td>
<td>$\langle \Gamma, \Sigma, \Lambda, pc, H, r \vdash e : \tau, \ (H, r, x) \triangleright_{\text{Scope}} r_x \rangle \xrightarrow{b(e,S)} \langle \Gamma, \Sigma, \Lambda, \Delta', pc, (H, r) \rangle$</td>
</tr>
<tr>
<td>M-Join</td>
<td>$\langle \Gamma, \Sigma, \Lambda, \Delta', pc, (H, r) \rangle \xrightarrow{\downarrow} \langle \Gamma', \Sigma', \Lambda, \Delta', pc, (H, r) \rangle$</td>
</tr>
<tr>
<td>M-Compensate</td>
<td>$\langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r) \rangle \xrightarrow{k(h)} \langle \Gamma, \Sigma, \Lambda, \Delta', pc, (H, r) \rangle$</td>
</tr>
<tr>
<td>M-OutputFailStop</td>
<td>$\langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r) \rangle \xrightarrow{\alpha(v,a)} \langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r) \rangle$</td>
</tr>
<tr>
<td>M-OutputSuppress</td>
<td>$\langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r) \rangle \xrightarrow{\alpha(v,a)} \langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r) \rangle$</td>
</tr>
<tr>
<td>M-OutputDefault</td>
<td>$\langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r) \rangle \xrightarrow{\alpha(D)} \langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r) \rangle$</td>
</tr>
</tbody>
</table>
Table 12: Collecting variables and functions from a statement $S$.

<table>
<thead>
<tr>
<th>$\text{collect}(S)$</th>
<th>$\text{colFunc}(e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${S^*}$</td>
<td>${f}$</td>
</tr>
<tr>
<td>if($e$) $S_1$ else $S_2$</td>
<td>${g}$</td>
</tr>
<tr>
<td>$x = e$</td>
<td>${x.f}$</td>
</tr>
<tr>
<td>$x.y = e$</td>
<td>${x[g]}$</td>
</tr>
<tr>
<td>$x[m] = e$</td>
<td>${x.f}$</td>
</tr>
<tr>
<td>while($e$) $S$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>for($e_1$; $e_2$; $e_3$) $S$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>for($x$ in $e$) $S$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>return $e$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>throw $e$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>with($e$){$S$}</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>output($e$)</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>otherwise</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

(XSS) attack involves the injection of malicious JavaScript code in a benign website, that when visited by the end-user (e.g. using a web browser) it leaks sensitive information to a server that is controlled by the attacker. The attacker has no control over the running script inside the victim’s web browser nor she can intercept the communication between the victim and the benign website.

The desired security property in this case is termination insensitive non-interference (TINI) [14], [2]. Non-interference means that an attacker who can supply $\ell$-level inputs and can observe $\ell$-level outputs cannot observe outputs whose security levels are higher than $\ell$. Termination insensitive means that leaks due to progress or lack-of-progress at each step are ignored, that is because the attacker can not learn the secret in polynomial time in the size of the secret [1]. Definition 12 provides a formal definition of the TINI property. This definition relies on the notion of equivalence between heap memories that is formally expressed in Definition 11.

Definition 11. (Equivalence relation $\sim_{\Gamma, \Sigma}^\ell$) Given two heap memories $H_1$ and $H_2$ with the same domain, typing environment $\Gamma$, and a structure security environment $\Sigma$, we define that the two heaps are level-$\ell$ equivalent denoted $H_1 \sim_{\Gamma, \Sigma}^\ell H_2$, if for all the variables $x \in \text{dom}(H_1)$ and $\text{dom}(H_2)$, $\Gamma(x) \subseteq \ell$ and objects $o \in \text{dom}(H_1)$ and $\text{dom}(H_2)$, $\Sigma(o) \subseteq \ell$.

Definition 12. (TINI) A statement $S$ satisfies TINI if the following holds for an attacker at any security level $\ell$. For any initial heap memories $H_1$ and $H_2$ that are level-$\ell$ equivalent $H_1 \sim_{\Gamma, \Sigma}^\ell H_2$ if

$$
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H_1, r, s \rangle \rangle \xrightarrow{w} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc \rangle, \langle H'_1, r, s' \rangle \rangle
$$
then \( \exists H'_2 \) and \( \vec{w}' \) such that

\[
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H_2, r, s \rangle \rangle \xrightarrow{\vec{w}} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc \rangle, \langle H'_2, r, s' \rangle \rangle
\]

and either

1. \( \vec{w} \) and \( \vec{w}' \) have same sequence of \( \ell \)-visible outputs events (outputs on channels of level \( \sqsubseteq \ell \))

2. or \( \vec{w}' \) is a prefix of \( \vec{w} \) and monitor configuration

\[
\langle \Gamma', \Sigma', \Lambda', \Delta', \langle H'_2, r, s' \rangle \rangle
\]

is in \( \ell' \)-level state where \( \ell' \not\sqsubseteq \ell \) and outputs in level \( \ell' \) are suppressed due to the monitor.

We establish the soundness of our proposed monitor in the following.

**Theorem 1. (Soundness)** The proposed VM Monitor is secure with respect to TINI.

**proof:** is by induction on \( \rightarrow^* \) of the VM monitor. The full proof is presented in Appendix A. It is important to note that our VM monitor supports multi-level security lattice when compared to the VM monitor presented in [14] and our TINI definition is similar to the batch-job TINI defined in [1] since the attackers are not allowed to observe intermediate results of computation through outputs. ■
4 IF-Transpiler: Inlining of Hybrid Flow-Sensitive Security Monitor For JavaScript

In this section we present the IF-Transpiler. As we mentioned earlier, our approach operates as a source-to-source compiler, in which, the input is a JavaScript code and the output is an instrumented version of the code, where the security monitor is inlined. The IF-Transpiler consists of two main stages: transformation stage, and inlining stage. The transformation stage serves as a preparation stage for the inlining stage. formally, \( IF-\text{Transpiler}(S) = (I \circ T)(S) = \hat{S} \), which means the composition of the inlining function \( I: S' \mapsto \hat{S} \) with transformation function \( T: S \mapsto S' \) on JavaScript Statement \( S \).

4.1 Transformation Stage

The transformation stage performs a set of transformations on the JavaScript statements to facilitate the inlining process. The set of transformations are syntactical changes to the JavaScript statements that yield semantically equivalent statements. For example, the following If-Statement "if(e) \( S \)" has a single statement \( S \) in the then-clause and has no else-clause. The transformation function will transform it to "if(e){\( S \})\ else\{skip;\}"", which is a semantically equivalent statement to the original although the curly brackets and the empty else-clause have been added. The additional changes are important because the inlining stage assumes that every If-Statement has an else-clause and uses curly brackets so that it can inline (insert) the monitoring statements and still produce semantically equivalent code. The transformation function \( T: S \mapsto S' \) is defined in Table 13 and uses the string concatenation operator "+" to concatenate statements and a helper function \( \text{Hoist}: S \mapsto S' \) to hoist and transform some of JavaScript expressions and statements that are challenging. For instance, the ternary conditional expression "\( x = e_1 ? e_2 : e_3; \)" can not be instrumented in a straightforward way. It is not possible to insert the monitoring statements and still produce semantically equivalent code, as such, the \( \text{Hoist} \) function transforms it into an equivalent If-Statement "if(\( e_1 \)){\( x = e_2; \})\ else\{x = e_3; \}". Another example is the de-anonymization of anonymous functions. Anonymous functions in JavaScript have the following syntax "(\( function(x)\{S\})\)(a)", which defines both an anonymous function and invokes it by passing the variable "\( a \)". We have to hoist the function definition to separate it from the function invocation. By doing that, it is possible to insert additional statements to track the information flow, this is outlined in Table 13 when the case is an anonymous function invocation.

4.2 Inlining Stage

The inlining stage operates in a similar fashion to the transformation stage. The input is a JavaScript statement \( S' \) and the output is an instrumented version \( \hat{S} \), formally, \( I: S' \mapsto \hat{S} \). The \( S' \) is expected to be a statement that has been converted through the transformation stage. In addition, the inlining function assumes the existence of the following global variables \( \Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in} \). These global variables correspond to \( \Gamma, \Sigma, \Lambda, \Delta, pc \) symbols that constitute the VM monitor configuration except that they are the inlined version, hence, the \( in \) subscript. The only difference is that \( \Gamma_{in} = \Gamma \uplus \Sigma \) where, \( \uplus \) means a disjoint union of \( \Gamma \) and \( \Sigma \). Therefore, the inlining process will have the form \( I_{in}: \Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in} \vdash I(S) = \hat{S} \). The inlining function make use of \( \text{sec_fv}(e) \) function defined in Table 14 to inline the security
Table 13: Transformation function $T(S)$.}

\[
\begin{align*}
T(S) &::= \\
x &= e; & \Rightarrow \text{\texttt{Hoist}}(x = e; ) \\
x.y &= e; & \Rightarrow \text{\texttt{Hoist}}(x.y = e; ) \\
e_1.x &= e_2; & \Rightarrow \text{\texttt{Hoist}}($t_1.x = e_2; ) + \text{\texttt{Hoist}}($t_2.x = e_2; ) + \text{\texttt{Hoist}}($t_1[$t_2] = e_3; ) \\
e_1[e_2] &= e_3; & \Rightarrow \text{\texttt{Hoist}}($t_1 = e_1; ) + \text{\texttt{Hoist}}($t_2 = e_2; ) + \text{\texttt{Hoist}}($t_1[$t_2] = e_3; ) \\
(fuction(x){S})((e)); & \Rightarrow \text{\texttt{Hoist}}($tmp = (function(x){S})(e); ) \\
(e); & \Rightarrow \text{\texttt{Hoist}}($tmp = e; ) + ($tmp); \\
\{S\} &\Rightarrow \text{\texttt{Hoist}}($tmp = e; ) + (T(S_1) + \ldots + T(S_n)); \\
if(e) S &\Rightarrow \text{\texttt{Hoist}}($tmp = e; ) + \text{\texttt{if}}($tmp \{T(S)\}) \text{\texttt{else}} \{\}
\]

case S : 

\[
\begin{align*}
\text{\texttt{Hoist}}(S) &::= \\
x &= y[e] & \Rightarrow_{N} \text{\texttt{Hoist}}($tmp = e; ) + x = y[$tmp]; \\
x &= e_1.e_2 & \Rightarrow_{N} \text{\texttt{Hoist}}($t_1 = e_1; ) + \text{\texttt{Hoist}}(x = $t_1.e_2; ) \\
x &= e_1[e_2] & \Rightarrow_{N} \text{\texttt{Hoist}}($t_1 = e_1; ) + \text{\texttt{Hoist}}($t_2 = e_2; ) + x = $t_1[$t_2]; \\
x &= e_1[e_2](e) & \Rightarrow_{N} \text{\texttt{Hoist}}($t_1 = e_1; ) + \text{\texttt{Hoist}}($t_2 = e_2; ) + \text{\texttt{Hoist}}(x = $t_1[$t_2](e); ) \\
x &= e.y & \Rightarrow_{N} \text{\texttt{Hoist}}($tmp = e; ) + x = $tmp.y; \\
x &= e_1; e_2 : e_3 & \Rightarrow_{N} \text{\texttt{Hoist}}($tmp = e; ) + x = e_1; e_2 : e_3 \\
x &= y.f(e = \{e_1,..,e_n\}) & \Rightarrow_{N} \text{\texttt{Hoist}}($t_1 = e_1; ) + \ldots + \text{\texttt{Hoist}}($t_n = e_n; ) + x = y.f($t_1,...,$t_n); \\
x &= y[f](e = \{e_1,..,e_n\}) & \Rightarrow_{N} \text{\texttt{Hoist}}($t_1 = e_1; ) + \ldots + \text{\texttt{Hoist}}($t_n = e_n; ) + x = y[f]($t_1,...,$t_n); \\
x &= f(e = \{e_1,..,e_n\}) & \Rightarrow_{N} \text{\texttt{Hoist}}($t_1 = e_1; ) + \ldots + \text{\texttt{Hoist}}($t_n = e_n; ) + x = f($t_1,...,$t_n); \\
x &= new y.f(e = \{e_1,..,e_n\}) & \Rightarrow_{N} \text{\texttt{Hoist}}($t_1 = e_1; ) + \ldots + \text{\texttt{Hoist}}($t_n = e_n; ) + x = new y.f($t_1,...,$t_n); \\
x &= new y[f](e = \{e_1,..,e_n\}) & \Rightarrow_{N} \text{\texttt{Hoist}}($t_1 = e_1; ) + \ldots + \text{\texttt{Hoist}}($t_n = e_n; ) + x = new y[f]($t_1,...,$t_n); \\
x &= new f(e = \{e_1,..,e_n\}) & \Rightarrow_{N} \text{\texttt{Hoist}}($t_1 = e_1; ) + \ldots + \text{\texttt{Hoist}}($t_n = e_n; ) + x = new f($t_1,...,$t_n); \\
x &= [e_1,..,e_n] & \Rightarrow_{N} \text{\texttt{Hoist}}($t_1 = e_1; ) + \ldots + \text{\texttt{Hoist}}($t_n = e_n; ) + x = [e_1,..,e_n]; \\
\{pm_1 : e_1,..,pn : e_n\} & \Rightarrow_{N} \text{\texttt{Hoist}}($t_1 = e_1; ) + \ldots + \text{\texttt{Hoist}}($t_n = e_n; ) + x = \{pm_1 : $t_1,...,pn : $t_n\}; \\
(fuction(x){S})(e); & \Rightarrow_{N} \text{\texttt{Hoist}}($tmp = e; ) + x = $tmp; \\
\text{\texttt{var}} x = e_1 \text{\texttt{in}} e_2 & \Rightarrow_{N} \text{\texttt{Hoist}}($tmp = e_2; ) \Rightarrow_{N} \text{\texttt{Hoist}}(x = e; ) + x = \text{\texttt{Hoist}}(x = e; ) \\
\text{\texttt{otherwise}} & \Rightarrow_{N} S \\
\text{\texttt{try}}(S) & \Rightarrow_{N} \text{\texttt{Hoist}}($tmp = e; ) + \text{\texttt{try}}(T(S)) ;
\end{align*}
\]

\[
\begin{align*}
\text{\texttt{catch}}(x){S} & \Rightarrow_{N} \text{\texttt{catch}}(x)(T(S)) \\
finally(S) & \Rightarrow_{N} \text{\texttt{finally}}(T(S)) \\
output_e(e) & \Rightarrow_{N} \text{\texttt{Hoist}}($tmp = e; ) + \text{\texttt{output}}($tmp) \\
\text{\texttt{return}} e; & \Rightarrow_{N} \text{\texttt{Hoist}}($tmp = e; ) + \text{\texttt{return}} $tmp; \\
\text{\texttt{throw}} e; & \Rightarrow_{N} \text{\texttt{Hoist}}($tmp = e; ) + \text{\texttt{throw}} $tmp; \\
\text{\texttt{id} : S} & \Rightarrow_{N} \text{\texttt{id} : \{T(S)\}} \\
\text{\texttt{Hoist}}(S) &::= \\
\text{\texttt{if}}(e) S & \Rightarrow_{N} \text{\texttt{Hoist}}($tmp = e; ) + \text{\texttt{if}}($tmp \{T(S)\}) \text{\texttt{else}} \{\}
\end{align*}
\]
Table 14: Security level of expression $e$.

$sec_{lvl}(e) ::= \text{case}:
\begin{align*}
&x \Rightarrow \text{scope($$$CS,$'$x')[$'$x'$]}; \\
&\text{this} \Rightarrow $$$CS.$this \\
&pv \Rightarrow \perp \\
&e_1 \triangleright_{bin} e_2 \Rightarrow sec_{lvl}(e_1) \sqcup sec_{lvl}(e_2) \\
&\triangleright_{un} x \Rightarrow sec_{lvl}(x) \\
&x \triangleright_{po} \Rightarrow sec_{lvl}(x) \\
&y[z] \mid y.z \Rightarrow \text{prop('y', 'z', $$$CS)}
\end{align*}$

level of the expression $e$. As a convention we precede variables introduced by the inlining process with $\$ symbol to distinguish them from user defined variables; these include all the global variables $\Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in}$, hence, in the instrumented code they will be referred to as $\$\Gamma, \$\Lambda$ and so on. The inlining function makes use of a special symbol $\$\$CS$ which always points to the current scope object. The $\$\$CS$ symbol is a parse time variable and does not appear in the instrumented code; it will be replaced by the proper current scope object of the typing environment $\Gamma$. For example, if the current scope is the global scope the $\$\$CS$ symbol will be replaced by $\Gamma[\text{'Global'}]$, and if the current scope is a function foo that is invoked in the global scope then the $\$\$CS$ symbol will be replaced by $\Gamma[\text{'Global'}][\text{'foo'}]$ etc. The inlining function also makes use of a $\$\text{scope}(\$\$CS, \text{var})$ function and a $\$\text{proto}(\text{obj, prop, \$\$CS})$ function. The two functions correspond to the $\triangleright_{\text{Scope}}$ and $\triangleright_{\text{Proto}}$ relations defined previously in subseciotn 2.4.

The $\text{scope($$$CS, \text{var})}$ function takes two arguments, the first is the current scope object $\$\$CS$ and the second argument is the variable var to locate. The $\text{proto(\text{obj, prop, \$\$CS})}$ function takes three arguments, the first is the base object, the second is the property we are trying to locate on the base object or its prototypes, and the third argument is the current scope object. The inlining function is outlined in Tables 15 to 17. It is important to note that the inlining function introduces the $\checkmark$ and $\square$ statements, which are only used for our proof of correctness and do not appear in the instrumented code.

### 4.3 Soundness of the Inlined Security Monitor

Similar to [2], we prove the soundness of the instrumented programs by showing that they are observationally equivalent to programs monitored by the VM monitor. The assumption is that two parallel runs of the monitored and the instrumented programs, starting with same user memory and compatible monitor states, produce the same output traces and end up with the same user memory and compatible monitor states. We start our formalization by defining monitor state coupling.

**Definition 13. (Monitor State Coupling)** ($mst$) Define $\langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle \cong (\Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in} = mst)$ if and only if $\Gamma_{in} = \Gamma \uplus \Sigma, \Lambda_{in} = \Lambda, \Delta_{in} = \Delta, \text{and } pc_{in} = pc$.

The instrumented program should start with configuration $\langle H_0 \uplus mst_0, r_0, \hat{S} \rangle$, where $H_0$ is the initial heap memory, $mst_0$ is the initial monitor state, and the initial scope

\footnote{There is a small detail regarding the scope lookup function, which we chose to omit for the sake of simplicity: if the variable being looked-up in the scope chain is on the left-hand-side (lhs) of an assignment expression and it does not exist, it will be added to the global scope. However, if it is on the right-hand-side it is considered as runtime error.}
Table 15: Inlining function $I(S)$ [Part A].

$$
\begin{align*}
\Gamma, \Lambda, \Delta, pc \vdash t(S') &\quad::=\quad \text{case } S' : \\
&\quad\Rightarrow\quad \text{Skip} : \✓ \\
&\quad\Rightarrow\quad \text{function } foo(\bar{x})\{t(S)\} : \✓ \quad+ $$S'S'[foo'] = \\
\quad\Rightarrow\quad \text{function } foo(\bar{x})\{t(S)\} : \✓ \quad+ $$S'S'[foo'] = \\
\quad\Rightarrow\quad \text{obj} = \{ pm_1 : e_1, ..., pm_n : e_n \} : \✓ \\
\quad\Rightarrow\quad \text{obj} = \{ pm_1 : e_1, ..., pm_n : e_n \} : \✓ \quad+ $$S'S'[obj] = \\
\quad\Rightarrow\quad x = [e_0, e_1, ..., e_{n-1}] : \✓ \\
\quad\Rightarrow\quad x = [e_0, e_1, ..., e_{n-1}] : \✓ \quad+ $$S'S'[x'] = \\
\quad\Rightarrow\quad x = f(\bar{e}) : \✓ \\
\quad\Rightarrow\quad x = f(\bar{e}) : \✓ \quad+ $$rf.scoped = $$S'CS; \quad+ $$rf.scoped = $$S'CS; \\
\quad\Rightarrow\quad x = f(\bar{e}) : \✓ \quad+ $$rf.scoped = $$S'CS; \quad+ $$rf.scoped = $$S'CS; \\
\quad\Rightarrow\quad x = y.f(\bar{e}) : \✓ \\
\quad\Rightarrow\quad x = y.f(\bar{e}) : \✓ \quad+ $$rf.scoped = $$S'CS; \quad+ $$rf.scoped = $$S'CS; \\
\quad\Rightarrow\quad x = y.f(\bar{e}) : \✓ \quad+ $$rf.scoped = $$S'CS; \quad+ $$rf.scoped = $$S'CS; \\
\quad\Rightarrow\quad x = e : \✓ \\
\quad\Rightarrow\quad x = e : \✓ \quad+ $$rf.scoped = $$S'CS; \quad+ $$rf.scoped = $$S'CS; \\
\end{align*}
$$
Table 16: Inlining function $I(S)$ [Part B]

$x.y = e$; \[\Rightarrow_i x.y = e; +\text{scope}($\$CS,'x$)'["x"]'["y"] = \text{sec}_\text{Jvl}(e); + (\text{isObj}(\text{prop}'(x',y',\$CS)?)\ \\
\text{prop}'(x',y',\$CS),\Sigma = \text{prop}'(x',y',\$CS),\Sigma \cup \text{pc.ℓ}+$ \\\n:\text{scope}($\$CS,'x$)'["x"]'["y"] = \text{prop}'(x',y',\$CS)\cup\text{pc.ℓ}; + \\\n\text{scope}($\$CS,'x$)'["x"];\Sigma = \text{sec}_\text{Jvl}(x,y) \cup \text{sec}_\text{Jvl}(x); + \checkmark \]

$x[y] = e$; \[\Rightarrow_i x[y] = e; +\text{scope}($\$CS,'x$)'["x"]'["y"] = \text{sec}_\text{Jvl}(e); + \\\n\text{tmp} = (\text{isObj}(\text{sec}_\text{Jvl}(y))?\text{sec}_\text{Jvl}(y),\Sigma : \text{sec}_\text{Jvl}(y); + (\text{isObj}(\text{prop}'(x',y,\$CS)?)\text{prop}'(x',y,\$CS),\Sigma = \text{prop}'(x',y,\$CS),\Sigma \cup \text{tmp} \cup \text{pc.ℓ}+$ \\\n:\text{scope}($\$CS,'x$)'["x"]'["y"] = \text{prop}'(x',y,\$CS)\cup\text{tmp} \cup \text{pc.ℓ}; + \\\n\text{scope}($\$CS,'x$)'["x"];\Sigma = \text{sec}_\text{Jvl}(x[y]) \cup \text{sec}_\text{Jvl}(x); + \checkmark \]

return; \[\Rightarrow_i \$\text{old}_\text{pc} = \text{pc}; +\text{while}(\text{pc.id} \not= \text{"FUNC"}) \Lambda.\text{pop}(); + \\\n\text{if}(\$\$\text{CS.InvokedAsConstr})\{ + \$\$\text{CS.this.}\Sigma = \$\$\text{CS.this.}\Sigma \cup \$\text{old}_\text{pc.ℓ}; + \Lambda[\Lambda.$\Lambda$.\text{len} - 1] = \{\ell: \$\$\text{CS.this.};\}; + \} \text{else} \{ \\\n\Lambda[\Lambda.$\Lambda$.\text{len} - 1] = \{\ell: \$\text{old}_\text{pc.ℓ}; \} + \text{return}; + + \text{RET \{"lbl' : 'FUNC"} \]

return x; \[\Rightarrow_i \$\text{old}_\text{pc} = \text{pc}; +\text{while}(\text{pc.id} \not= \text{"FUNC"}) \Lambda.\text{pop}(); + \\\n\$\text{rx} = \text{scope}($\$CS,'x$)'["x"]; + \\\n\text{if}(\text{isObj}(\$\text{rx}))\{ + \$\text{rx.}\Sigma = \$\text{rx.}\Sigma \cup \$\text{old}_\text{pc.ℓ}; + \Lambda[\Lambda.$\Lambda$.\text{len} - 1] = \{\ell: \$\text{rx};\}; + \} \text{else if}(\$\$\text{CS.InvokedAsConstr})\{ + \$\$\text{CS.this.}\Sigma = \$\$\text{CS.this.}\Sigma \cup \$\text{old}_\text{pc.ℓ}; + \Lambda[\Lambda.$\Lambda$.\text{len} - 1] = \{\ell: \$\$\text{CS.this.};\}; + \} \text{else} \{ \$\Lambda[\$\Lambda.$\Lambda$.\text{len} - 1] = \{\ell: \$\text{rx} \cup \$\text{old}_\text{pc.ℓ}; \} + \text{return x}; + + \text{RET \{"lbl' : 'FUNC"} \]

break; | continue; \[\Rightarrow_i \$\text{old}_\text{pc} = \text{pc}; +\text{while}(\text{pc.id} \not= \text{"LOOP"}) \Lambda.\text{pop}(); + \\\n\$\Lambda[\$\Lambda.$\Lambda$.\text{len} - 2].\ell = \$\Lambda[\$\Lambda.$\Lambda$.\text{len} - 2].\ell \cup \$\text{old}_\text{pc.ℓ}; + \text{break; | continue}; + + \text{RET \{"lbl' : 'LOOP"} \]

break 'label'; | continue 'label'; \[\Rightarrow_i \text{same as previous case} + + \checkmark \]

if(e) \{S_1\} else \{S_2\} \[\Rightarrow_i \$\text{x}_1 = \$\text{collect}(S_1); +\$\text{x}_2 = \$\text{collect}(S_2); + \\\n\Lambda.\text{push}((\ell: \text{pc.ℓ} \cup \text{sec}_\text{Jvl}(e), \text{id} : \text{"IF"}); + \\\n\text{if}(e) \{\$\text{upgrade}(\$\text{x}_2, \text{pc.ℓ}); + + + \checkmark + \text{var \$\text{should}_\text{comp} = I(S_1); \} \text{else} + \{\$\text{upgrade}(\$\text{x}_1, \text{pc.ℓ}); + + + \checkmark + \text{var \$\text{should}_\text{comp} = I(S_2); \} + + \text{if}($\text{should}_\text{comp})\{\$\text{comp}_A($\text{should}_\text{comp}.\text{lbl}, \text{pc.ℓ}); \} + + \Lambda.\text{pop}(); + + \checkmark}
Table 17: Inlining function $I(S)$ (Part C).

$$\text{while}(e)\{S\} \Rightarrow_t \Delta.push(\{\ell : \text{sec}_p \cup \text{sec}_l(e), \text{lbl} : \text{LDOOP}'\}); +$$
$$\Delta.push(\{xs : \text{Collect}(S), \ell : \text{pc}_\ell\}); +$$
$$\text{while}(e)\{+$$
$$\text{true}; \text{var $should\_comp = I(S); } +$$
$$\text{tmp} = \Delta.pop(); +$$
$$\text{tmp}_x = \text{upgrade}(\text{tmp}_x, \text{tmp}_\ell); +$$
$$\text{if($should\_comp\} \{\text{Comp}($$ $should\_comp, \text{lbl}, \text{pc}_\ell\}; } + \Delta.pop(); + \text{true}$$

$$\text{for}(); e; \}\{S\} \Rightarrow_t \Delta.push(\{\ell : \text{sec}_p \cup \text{sec}_l(e), \text{lbl} : \text{LDOOP}'\}); +$$
$$\Delta.push(\{xs : \text{Collect}(S), \ell : \text{pc}_\ell\}); +$$
$$\text{for}(); e; \}\{+$$
$$\text{true}; \text{var $should\_comp = I(S); } +$$
$$\text{tmp} = \Delta.pop(); +$$
$$\text{tmp}_x = \text{upgrade}(\text{tmp}_x, \text{tmp}_\ell); +$$
$$\text{if($should\_comp\} \{\text{Comp}($$ $should\_comp, \text{lbl}, \text{pc}_\ell\}; } + \Delta.pop(); + \text{true}$$

$$S* \Rightarrow_t \text{var $tmp$, $ret$; } +$$
$$\text{tmp} = I(S_1); + \text{if($ret$) $ret = tmp$; } +$$
$$\text{RET} \text{ $ret$}$$

$$id : S \Rightarrow_t \Delta.push(\{\ell : \text{pc}_\ell, \text{lbl} : \text{id}'\}); + \text{id : $I(S)$} + \Delta.pop(); + \text{true}$$

$$\text{throw e; } \Rightarrow_t \text{old}_p = \text{pc}_p; +$$
$$\text{while($\text{pc}_id !== \text{\textquote{FUNCTION}} \&\& \text{pc}_id !== \text{\textquote{TRY}}\} \{\Delta.pop(); +$$
$$\Delta(\text{\textquote{Len} - 1}) = \{\ell : \text{old}_p \cup \text{sec}_l(e)\}; +$$
$$\text{throw e; } + \text{true}$$

$$\text{RET \{\text{Lbl} : \text{\textquote{FUNCTION}} | \text{\textquote{TRY}}\}$$

$$\text{with}(x)\{S\} \Rightarrow_t \Delta.push(\{\ell : \text{sec}_p \cup \text{sec}_l(x), \text{lbl} : \text{WITH}'\}); +$$
$$\text{with}(x)\{+$$
$$\text{var $ro$ = $scope(x$, $$CS')[x']}; +$$
$$\text{var $ro$.$this = $$CS$.this}; +$$
$$\text{var $ro$ = $scope$}; +$$
$$\text{I(S)} +$$
$$\text{CS = $ro$.scope}; +$$
$$\Delta.pop(); + \text{true}$$

$$\text{try\{S\} } \Rightarrow_t \text{try}\{\Delta.push(\{\ell : \text{pc}_\ell, \text{lbl} : \text{\textquote{TRY}}\}); + $$
$$I(S); + \Delta.pop(); } + \text{true}$$

$$\text{catch(x)\{S\} } \Rightarrow_t \text{catch(x)\{I(S); + \Delta.pop(); } + \text{true}$$

$$\text{finally\{S\} } \Rightarrow_t \text{finally\{I(S) + \text{true}$$

$$\text{else } \Rightarrow_t \text{case $Policy$ : }$$

'OutputFailStop' : $$\text{if($\text{sec}_l(e) \cup \text{sec}_l \ell \subset \ell$) $output_e(e); + \text{true}$$

'OutputSuppress' : $$\text{if($\text{sec}_l(e) \cup \text{sec}_l \ell \subset \ell$) $output_e(e); + \text{true}$$

'OutputDefault' : $$\text{if($\text{sec}_l(e) \cup \text{sec}_l \ell \subset \ell$) $output_e(e); + \text{true}$$

'Otherwise' : $$\Rightarrow_t S'$$
B. on the instrumented statement \( \hat{\text{every step performed by the VM monitor}} \), we go by cases on statement \( S \) coupled with the inlined monitor configuration \( o \) by filtering non-output events and changing every \( S \) is about the global scope, \( \Delta \) initially is empty, and \( pc \) is pointing to the top-most element of security context stack \( \Lambda \) which corresponds to the global context. In order to prove the soundness of the inlined monitor we have to define configuration coupling between the inlined monitor and the VM monitor as follows:

**Definition 14. (Configuration Coupling)** Define \( \langle \langle H_1, r_1, S \rangle, \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle \rangle \sim \langle \langle H_2 \uplus mst, r_2, \hat{S} \rangle \rangle \) if and only if \( H_1 = H_2, r_1 = r_2, \) and \( \exists \Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in} \) such that \( \Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in} \vdash \hat{S} = (T \circ \tau) \hat{S} \) and \( \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle \equiv (\langle \Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in} \rangle = mst) \).

Similar to [2], we define \( \overrightarrow{\beta} \) for transitions of statement configurations by

\[
\langle H \uplus mst, r, \hat{S} \rangle \overrightarrow{\beta} \langle H' \uplus mst', r, \hat{S}' \rangle
\]

to mean

\[
\langle H \uplus mst, r, \hat{S} \rangle \xrightarrow{\beta}^* \langle H' \uplus mst', r, \hat{S}' \rangle
\]

and we write \( \overrightarrow{\beta} \) to mean transitive closure concatenating transition labels.

It is important to note that the instrumented programs use the same semantics and generate the same internal events as defined in Tables 9 and 10. Among the possible events is \( o(e, va) \) which is an internal event and it is not supposed to be observable, hence, we define the following output relation that connects the traces of internal events with traces of monitored executions.

**Definition 15. (Output Relation)** We define \( \overrightarrow{\beta} |_{out} \overline{w} \) to hold if and only if \( \overline{w} \) is obtained by filtering non-output events and changing every \( o(e, va) \) to \( o(va) \).

**Theorem 2. (Observational Equivalence)** For all \( \hat{S}, \hat{S}', H, H', mst, \Gamma, \Sigma, \Lambda, \Delta, pc \) if \( \langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, S \rangle \rangle \sim \langle \langle H \uplus mst, r, \hat{S} \rangle \rangle \) then we have:

(a) For all \( S, \Gamma', \Sigma', \Lambda', \Delta', pc' \) if \( \langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, S \rangle \rangle \overrightarrow{\beta} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc' \rangle, \langle H', r, S' \rangle \rangle \)

then there is \( \hat{\beta} \) such that \( \overrightarrow{\beta} |_{out} \overline{w}, \langle H \uplus mst, r, \hat{S} \rangle \hat{\beta} \langle H' \uplus mst', r, \hat{S}' \rangle \),

and \( \langle \langle H', r, S' \rangle, \langle \Gamma', \Sigma', \Lambda', \Delta', pc' \rangle \rangle \sim \langle \langle H', r, S' \rangle, \hat{S}' \rangle \).

(b) For all \( \hat{S}, mst' \) if \( \langle H \uplus mst, r, \hat{S} \rangle \hat{\beta} \langle H' \uplus mst', r, \hat{S}' \rangle \) then there is \( \overline{w} \) such that \( \overrightarrow{\beta} |_{out} \overline{w}, \langle \langle \Gamma, \Sigma, \Lambda, pc \rangle, \langle H, r, S \rangle \rangle \overrightarrow{\beta} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc' \rangle, \langle H', r, S' \rangle \rangle \),

and \( \langle \langle H', r, S' \rangle, \langle \Gamma', \Sigma', \Lambda', \Delta', pc' \rangle \rangle \sim \langle H' \uplus mst', r, \hat{S}' \rangle \).

**proof:** here we only outline the idea of the proof, the full proof is presented in Appendix B.

Proof of clause (a) goes by the induction on the number of steps performed by the VM monitor starting from configuration \( \langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, S \rangle \rangle \) which is assumed to be coupled with the inlined monitor configuration \( \langle H \uplus mst, r, \hat{S} \rangle \) as stated in Theorem 2. For every step performed by the VM monitor, we go by cases on statement \( S \) and compare each step of the VM monitor with one or more steps performed by the inlined monitor on the instrumented statement \( \hat{S} \) finishing with a \( t \)-transition. In order to prove that
the resulting configuration is coupled \(\langle (\Gamma', \Sigma', \Lambda', \Delta', pc'), (H', r, S') \rangle \approx \langle H' \uplus \text{mst}', r, \hat{S}' \rangle\), we have to establish monitor state coupling: \(\langle H', r, S' \rangle \equiv \langle \text{mst}' (\Gamma'_m, \Lambda'_m, \Delta'_m, pc'_m) \rangle\). We sketch the case where \(S = \text{e} = e\); as an illustration. According to \(\text{IF-Transpiler}(\text{e}) \Rightarrow \hat{S} = (x = e; +$\text{rx}$ = $\text{scope}($$\text{CS}' x'$' \text{'}$'x'; +$\text{rx}$ = $\text{sec}$.jvl(e); +(isObj($$\text{rx}$'))$\text{rx}$.$\Sigma$ = $\text{rx}.$\Sigma$ $\uplus$ $\text{pc}.\ell$ : $\text{rx}$ = $\text{rx} \uplus$ $\text{pc}.\ell$; +\(\checkmark\)).

Now we show the steps of the VM monitor execution, assuming \(\tau\) to be the security type of expression \(e\), \(va\) is its value, and \(va\) is a primitive type (not an object just to make the example simple):

\[
\begin{align*}
\langle (\Gamma, \Sigma, \Lambda, \Delta, pc), (H, r, x = e) \rangle & \\
\xrightarrow{a_v(x, e)} & \\
\langle (\Gamma (r_x) [x \mapsto pc.\ell \uplus \tau], \Sigma, \Lambda, \Delta, pc), (H (r_x) [x \mapsto va], r, \Box) \rangle \text{ where } (H, r, x) \uplus \text{scope} \text{ rx} & \\
\end{align*}
\]

which defines \(H', \Gamma', S'\).

Next, we show the trace of the instrumented statement(s) until the first \(t\)-event:

\[
\begin{align*}
\langle H \uplus mst, r, "x = e; +$\text{rx}$ = $\text{scope}($$\text{CS}' x'$' \text{'}$'x'; +$\text{rx}$ = $\text{sec}$.jvl(e); +$\text{rx}$ = $\text{rx} \uplus$ $\text{pc}.\ell$; +\(\checkmark\)) \rangle & \\
\xrightarrow{\beta_1} & \\
\langle H (r_x) [x \mapsto va] \uplus mst, r, "$\text{rx}$ = $\text{scope}($$\text{CS}' x'$' \text{'}$'x'; +$\text{rx}$ = $\text{rx} \uplus$ $\text{pc}.\ell$; +\(\checkmark\)) \rangle & \\
\xrightarrow{\beta_2} & \\
\langle H (r_x) [x \mapsto va] \uplus mst, r, "$\text{rx}$ = $\text{sec}$.jvl(e); +$\text{rx}$ = $\text{rx} \uplus$ $\text{pc}.\ell$; +\(\checkmark\)) \rangle & \\
\xrightarrow{\beta_3} & \\
\langle H (r_x) [x \mapsto va] \uplus mst \equiv (\Gamma (r_x) [x \mapsto \tau], \Lambda, \Delta, pc), r, "$\text{rx}$ = $\text{sec}$.jvl(e); +$\text{rx}$ = $\text{rx} \uplus$ $\text{pc}.\ell$; +\(\checkmark\)) \rangle & \\
\xrightarrow{\beta_4} & \\
\langle H (r_x) [x \mapsto va] \equiv mst \equiv (\Gamma (r_x) [x \mapsto \tau \uplus$ $\text{pc}.\ell], \Lambda, \Delta, pc), r, "\checkmark\) \rangle & \\
\xrightarrow{t} & \\
\langle H (r_x) [x \mapsto va] \equiv mst \equiv (\Gamma (r_x) [x \mapsto \tau \uplus$ $\text{pc}.\ell], \Lambda, \Delta, pc), r, \Box) \rangle & \\
\end{align*}
\]

which define \(H', mst', \hat{S}'\).

By applying rule \(\Gamma', \Lambda, \Delta, pc \vdash t (\Box) \Rightarrow t (\Box)\) we obtain a valid transformation relation between \(S'\) and \(\hat{S}'\) and by taking \(\Gamma'_m = \Gamma' \uplus \Sigma\) and \(H' = H',\) we get \(\langle \Gamma', \Sigma, \Lambda, \Delta, pc \rangle \equiv (mst' (\Gamma' \uplus \Sigma, \Lambda' \uplus \Delta' \uplus \text{pc} \uplus \text{in}))\), consequently we get \(\langle H', r, S' \rangle \approx \langle H' \uplus mst', r, \hat{S}' \rangle\).

For clause (b) of Theorem 2, the proof goes by the induction on the structure of statement \(S\), the rules outlined in Tables 15 to 17, and the events that get generated by the language VM as outlined in the operational semantics of the language in Tables 9 and 10. Based on the rules in Tables 15 to 17, every instrumented statement \(\hat{S}'\) encompasses the semantics of the original non-instrumented statement \(S\) plus one or more additional statements, formally, \(\hat{S}' = S^* + S + S^* + \checkmark\), where \(S^*\) means zero or more statements and + is concatenation operator. Which leads us to the following: When \(\langle (\Gamma, \Sigma, \Lambda, \Delta, pc), (H, r, S) \rangle \overset{\alpha}{\Rightarrow} \langle (\Gamma', \Sigma', \Lambda', \Delta', pc), (H', r, S') \rangle\) and \(\langle H \uplus mst, r, \hat{S} \rangle \overset{\beta}{\Rightarrow} \langle H' \uplus mst', r, \hat{S}' \rangle\) then \(\beta\) will contain the same events generated by \(\alpha\) and if \(\alpha\mid_{\text{out}} \bar{w}_1\) and \(\beta\mid_{\text{out}} \bar{w}_2\) then \(\bar{w}_1 = \bar{w}_2\) or both are
empty when all the events that got generated are internal events $\epsilon$. In other words, both the VM monitor and the inlined monitor have to agree on the output events. For instance, if we take the previous example used in the proof of clause (a) where $\vec{\alpha} = \{a_v(x, e)\}$ in the case of the VM monitor and $\vec{\beta} = \{\beta_1, \beta_2, \beta_3, \beta_4\}$ in the case of the inlined monitor. If we apply the output relation $|_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$, we get $\vec{\alpha} |_{out} \emptyset$ and $\vec{\beta} |_{out} \emptyset$, trivially $\emptyset = \emptyset$ since all the events that got generated by the two monitors are internal events $\epsilon$ as outlined in Table 8.

What remains is to check that the configuration of both monitors are coupled, which has been done in the proof of clause (a).

A direct consequence of Theorems 1 and 2, and their proofs, is the following corollary:

**Corollary 3.** Every run of the instrumented program satisfies termination-insensitive non-interference (TINI) with respect to the chosen policy.

**References**


---

5 Corollary 3 corresponds to Theorem 3.


5 Appendix A: Proof of the Soundness of the Hybrid Flow-Sensitive Security Monitor

Definition 16. \( \sim \) For any two statements \( d \) and \( S \) such that \( d \) is not any of the \( \Box, \times \), or the \( \text{throw} \) statements, predicate \( d \sim \langle \langle \Gamma, \Sigma, \Delta, pc \rangle, \langle H, r, S' \rangle \rangle \) holds iff \( \exists H_0, r_0 \) and \( \langle \Gamma_0, \Sigma_0, \Delta_0, pc_0 \rangle \) such that
\[
\langle \langle \Gamma_0, \Sigma_0, \Delta_0, pc_0 \rangle, \langle H_0, r_0, d \rangle \rangle \rightarrow^* \langle \langle \Gamma, \Sigma, \Delta, pc \rangle, \langle H, r, S \rangle \rangle
\]
where \( \Delta_0 = \{ \bot \} \), \( \Delta_0 = \emptyset \), and \( r_0 = \#\text{Global} \).

Lemma 1 establishes some properties of \( \sim \) predicate.

Lemma 1. (Reachability).
- If \( d \sim \langle \langle \Gamma, \Sigma, \Delta, pc \rangle, \langle H, r, S' \rangle \rangle \) and \( \langle \langle \Gamma, \Sigma, \Delta, pc \rangle, \langle H, r, S' \rangle \rangle \rightarrow^* \langle \langle \Gamma', \Sigma', \Delta', pc' \rangle, \langle H', r', S'' \rangle \rangle \) and if an exception got thrown, we assume the existence of a try-catch clause in an outer scope, then \( d \sim \langle \langle \Gamma', \Sigma', \Delta', pc' \rangle, \langle H', r', S'' \rangle \rangle \) holds.
- If \( d \sim \langle \langle \Gamma, \Sigma, \Delta, pc \rangle, \langle H, r, if(e) \ S_1 \ else \ S_2 \rangle \rangle \) holds, then both \( S_1 \) and \( S_2 \) do not contain \( \Box \) or \( \times \) statements.
- If \( d \sim \langle \langle \Gamma, \Sigma, \Delta, pc \rangle, \langle H, r, for(e_1; e_2; e_3) \ S \rangle \rangle \) holds, then \( S \) does not contain \( \Box \) or \( \times \) statements.
- If \( d \sim \langle \langle \Gamma, \Sigma, \Delta, pc \rangle, \langle H, r, while(e) \ S \rangle \rangle \) holds, then \( S \) does not contain \( \Box \) or \( \times \) statements.
- If \( d \sim \langle \langle \Gamma, \Sigma, \Delta, pc \rangle, \langle H, r, do\{S\}while(e) \rangle \rangle \) holds, then \( S \) does not contain \( \Box \) or \( \times \) statements.
- If \( d \sim \langle \langle \Gamma, \Sigma, \Delta, pc \rangle, \langle H, r, for(x \ in \ e) \ S \rangle \rangle \) holds, then \( S \) does not contain \( \Box \) or \( \times \) statements.
- If \( d \sim \langle \langle \Gamma, \Sigma, \Delta, pc \rangle, \langle H, r, for(var \ x=e_1 \ in \ e_2) \ S \rangle \rangle \) holds, then \( S \) does not contain \( \Box \) or \( \times \) statements.
- If \( d \sim \langle \langle \Gamma, \Sigma, \Delta, pc \rangle, \langle H, r, with(e)\{S\} \rangle \rangle \) holds, then \( S \) does not contain \( \Box \) or \( \times \) statements.

From now on, we only consider configurations of the VM monitor that are reachable from statement \( d \) that do not contain any of the \( \Box, \times \), or the \( \text{throw} \) statements.

Lemma 2. (\( \sqcup \) on typing environments) Given \( H_1, H_2 \) and \( H_3 \), typing environments \( \Gamma, \Gamma' \), and \( \Gamma'' \), and structure security environments \( \Sigma, \Sigma' \), and \( \Sigma'' \), it holds that
- \( \Gamma \sqcup \Gamma' \sqcup \Gamma'' = \Gamma \sqcup \Gamma'' \sqcup \Gamma' \)
- \( \Sigma \sqcup \Sigma' \sqcup \Sigma'' = \Sigma \sqcup \Sigma'' \sqcup \Sigma' \)
- If \( H_1 \sim_{\Gamma, \Sigma} H_2 \) then \( H_1 \sim_{\Gamma \sqcup \Gamma', \Sigma \sqcup \Sigma', \Sigma''} H_2 \)
- If \( H_1 \sim_{\Gamma, \Sigma} H_2 \) and \( H_2 \sim_{\Gamma \sqcup \Gamma', \Sigma \sqcup \Sigma', \Sigma''} H_3 \) then \( H_1 \sim_{\Gamma \sqcup \Gamma' \sqcup \Gamma'', \Sigma \sqcup \Sigma', \Sigma''} H_3 \)

Lemma 3. (Behaviour of the VM Monitor when transitioning in arbitrary security level \( \ell \)) Given a statement \( S \), \( \Lambda \neq \emptyset \), and the monitored steps \( \langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, S \rangle \rangle \rightarrow^* \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc' \rangle, \langle H', r', \square \rangle \rangle \) then

1. \( \Lambda' \neq \emptyset \)
2. \( H' \sim_{\Gamma', \Sigma'} H \)

Proof: by induction on \( \rightarrow^* \)

- **x=e;** by reachability property (definition 1), item 1 holds since the execution have to start from global scope where \( \Lambda_0 = \{ \bot, \ell \} \), by applying M-Assign-Var in Table 11 then \( \Gamma' = \Gamma(r_x)[x \mapsto pc.\ell \cup \tau], \Sigma \) is not changed \( \Sigma' = \Sigma \), then \( H' \sim_{\Gamma', \Sigma'} H \) which means that item 2 holds by Lemma 2.

- **x.y=e;** by reachability property (definition 1), item 1 holds, by applying M-Assign-Obj-Prop in Table 11 then \( \Gamma' = \Gamma(r_x)[x \mapsto [y \mapsto pc.\ell \cup \tau_e]], \Sigma' = [x \mapsto pc.\ell \cup \tau_e \cup \Sigma(r_x)(x)], \) let \( \ell' = pc.\ell \cup \tau_e \cup \Sigma(r_x)(x) \) and by Lemma 2 item 2 holds with respect to \( \ell' \).

- **x[e1]=e2;** by reachability property (definition 1), item 1 holds, by applying M-Assign-Obj-Prop in Table 11 then \( \Gamma' = \Gamma(r_x)[x \mapsto [m \mapsto pc.\ell \cup \tau_1 \cup \tau_2]], \Sigma' = \Sigma(r_x)(x) \), let \( \ell' = pc.\ell \cup \tau_1 \cup \tau_2 \cup \Sigma(r_x)(x) \) and by Lemma 2 item 2 holds with respect to \( \ell' \).

- **output\(_\ell\)(e,:)** by reachability property (definition 1), item 1 holds, based on the chosen policy, the VM monitor will apply one of the following rules, M-OutputFailStop, M-OutputSuppress, or M-OutputDefault, none of the three rules modify VM monitor policy, the VM monitor will apply one of the following rules, M-OutputFailStop, M-output

- **if(e) S\(_1\) else S\(_2\)** by reachability property (definition 1) item 1 holds, we assume "e" evaluates to a va \( \in \{ 0, \text{null}, \text{undefined}, \text{false} \} \) (the proof when "e" evaluates to a va \( \in \{ 0, \text{null}, \text{undefined}, \text{false} \} \) proceeds similarly) by T-If-Stmt rule in Table 4, If-Then rule in Table 9, and M-Branch rule in Table 11 we have

\[
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \text{if(e) S\(_1\) else S\(_2\)} \rangle \rangle \xrightarrow{b(e,S\(_2\))} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc' \rangle, \langle H', r, \Lambda; \ S\(_1\) \rangle \rangle
\]

when M-Branch rule is applied, the \( \Delta' \) control-flow stack will contain all the variables and functions in the untaken branch \( S_2 \) and the security level \( \ell \) of the e conditional expression, \( \Delta' = \text{push}(\Delta, (\overline{x}, \tau)) \) where \( \overline{x} = \text{collect}(S_2) \)
when T-If-stmt rule is applied, the \( \Lambda' \) security-context stack will contain the higher of the security context where the If-Statement called and security level of expression e, \( \Lambda' = \text{push}(\Lambda, (\text{pc.}\ell \cup \tau)) \), then End rule in Table 10 and the M-Join rule in Table 11 will get applied, we get

\[
\langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc' \rangle, \langle H', r, \Lambda; \ S\(_1\) \rangle \rangle \xrightarrow{\text{End}} \langle \langle \Gamma'', \Sigma'', \Lambda', \Delta, pc' \rangle, \langle H'', r, \square \rangle \rangle
\]
then \( S\(_1\) \) (Then part of the If-Statement) statement will execute which will result in

\[
\langle \langle \Gamma'', \Sigma'', \Lambda', \Delta, pc' \rangle, \langle H'', r, S\(_1\) \rangle \rangle \rightarrow^* \langle \langle \Gamma''', \Sigma''', \Lambda, \Delta, pc \rangle, \langle H''', r, \square \rangle \rangle
\]
by Lemma 2 \( H'''' \sim_{\Gamma''', \Sigma'''} H \) where \( \ell'' = pc.\ell \cup \tau \) and \( \tau \) is the security level of the conditional expression "e" and \( \Gamma'''' = \Gamma \cup \Gamma' \cup \Gamma'' \), \( \Sigma'' = \Sigma \cup \Sigma' \cup \Sigma'' \) hence item 2 holds.
Hybrid Flow-Sensitive Security Monitor for JavaScript

- **while(e)S;** same as the If-Statement case (previous case) with the difference that when the monitor encounters a join statement that corresponds to a branch event \(b(e, \text{skip})\) the monitor performs the join transition without upgrading any variables or functions since \(\text{collect} (\text{skip}) = \emptyset\), therefore \(\Gamma' = \Gamma \sqcup \emptyset = \Gamma\) and \(\Sigma' = \Sigma \sqcup \emptyset = \Sigma\), consequently item 2 holds.

- **do{S}while(e);** same as previous case.

- **for(e_1;e_2;e_3) S;** same as while-loop case with the difference that the monitor branches on \(e_3\), \(b(\vec{e} = \{e_1,e_2\},S)\).

- **for(var x=e_1 in e_2) S;** same as while-loop case with the difference that the monitor branches on both \(e_1\) and \(e_2\), \(b(\vec{e} = \{e_1,e_2\},S)\) and \(\tau = \tau_{e_1} \sqcup \tau_{e_2}\).

- **for(x in e) S;** same as while-loop case with the difference that the monitor branches on expression \(e\), \(b(\vec{e},S)\).

- **x=e(\vec{e});** by reachability property (definition 1) item 1 holds, by applying rule T-Function Call in Table 4, \(\Gamma'' = \Gamma \sqcup \hat{\Gamma}[x \mapsto \tau_{e_1} \sqcup \ell], \Sigma'' = \Sigma \sqcup \hat{\Sigma}, H'' = \hat{H}[x \mapsto \text{va}], H \sim^*_{\Gamma'', \Sigma''} H''\), the monitor skips over function calls since the statements of the function body will be monitored.

- **x=e[e'](\vec{e});** same as previous case with the difference of applying rule T-Method Call in Table 5.

- **return e;** by reachability property (definition 1) item 1 holds, by applying T-Return rule in Table 5, Return rule in Table 10, the monitor will apply M-Compensate rule in Table 11 to upgrade the entries of the security context stack \(\Lambda\) with \(pc.\ell\) until one level above the label \(\text{lbl}\),

\[
\langle \Gamma, \Sigma, \Lambda, \Delta, pc, (H, r, e) \rangle \rightarrow \langle \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H', r, \text{va} \rangle \rangle,
\]

\[
\langle \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H', r, \text{return va} \rangle \rangle \xrightarrow{\text{k(FUNC)}} \langle \langle \Gamma'', \Sigma'', \Lambda', \Delta, pc \rangle, \langle H'', r, \square \rangle \rangle
\]

where \(\Lambda' = \text{comp}_{\text{A}}(\text{FUNC}, \ell), \ell = pc.\ell \sqcup \tau\) and \(\tau\) is the security type of expression \(e\), where \(\Gamma'' = \Gamma \sqcup \Gamma', \Sigma'' = \Sigma \sqcup \Sigma'\) by Lemma 2 then \(H'' \sim^*_{\Gamma'', \Sigma''} H\) hence item 2 holds.

- **throw e;** similar to the previous case with the difference that compensate function will upgrade security contexts up until "TRY" or "FUNC" label \(\text{comp}_\Lambda(\text{TRY} \mid \text{FUNC}, \ell)\).

- **break;** by reachability property (definition 1) item 1 holds, by applying T-Break rule in Table 5 and Break rule in Table 10, the monitor will apply M-Compensate rule in Table 11 which modifies only security context stack \(\Lambda\) hence, \(\Gamma' = \Gamma, \Sigma' = \Sigma\), therefore \(H \sim^*_{\Gamma, \Sigma} H'\) and item 2 holds.

- **continue;** same as previous case.

- **break lbl;** same as break case with the difference that the monitor upgrades the security context stack \(\Lambda\) to the label "lbl" \(\text{comp}_\Lambda(\text{lbl}, pc.\ell)\).

- **continue lbl;** same as previous case.
• try\{S\}) by reachability property (definition 1) item 1 holds, by applying T-Try rule in Table 5 and Try rule in Table 10, the monitor steps over try-statements. The only change is in \(\Lambda\), hence \(\Gamma' = \Gamma, \Sigma' = \Sigma\) and item 2 holds.

• catch(x)\{\}) by reachability property (definition 1) item 1 holds, by applying T-Catch rule in Table 5 and Catch rule in Table 10, the monitor steps over catch-statements. The only change is in \(\Lambda\), hence \(\Gamma' = \Gamma, \Sigma' = \Sigma\) and item 2 holds.

• finally\{S\}) by reachability property (definition 1) item 1 holds, by applying T-Finally rule in Table 5 and Finally rule in Table 10, the monitor steps over finally-statements, hence \(\Gamma' = \Gamma, \Sigma' = \Sigma\) and item 2 holds.

**Theorem 4.** (Soundness) For any heap memory \(H\) and statement \(S\), the execution of \(S\) starting at configuration \(\langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, S \rangle \rangle\) is secure according to Definition 12 (TINI).

**proof:** By induction over the number of the \(\ell\)-visible outputs and application of Lemma 3. ■

**Corollary 5.** (Batch-Job Soundness) Given a program \(S\), heap memories \(H_1\) and \(H_2\) such that \(H_1 \sim^{\ell}_{\Gamma, \Sigma} H_2\) and two terminating runs:

\[
\langle \Gamma_1, \Sigma_1, \Lambda_1, \Delta_1, pc_1 \rangle, \langle H_1, r, S \rangle \rangle \xrightarrow{w_1} \langle \Gamma_1', \Sigma_1', \Lambda_1', \Delta_1', pc_1 \rangle, \langle H_1', r, \square \rangle \rangle
\]

and

\[
\langle \Gamma_2, \Sigma_2, \Lambda_2, \Delta_2, pc_2 \rangle, \langle H_2, r, S \rangle \rangle \xrightarrow{w_2} \langle \Gamma_2', \Sigma_2', \Lambda_2', \Delta_2', pc_2 \rangle, \langle H_2', r, \square \rangle \rangle
\]

then it holds that \(w_1 = w_2\), with respect to security level \(\ell\), and execution of program \(S\) is secure according to Definition 12.

**proof:** By Theorem 4 and application of Lemma 3. ■
Appendix B: Proof of the Observational Equivalence of the Inlined Monitor

As we mentioned before, the proof of clause (a) of Theorem 2 goes by induction on the number of steps performed by the VM monitor starting from a coupled configuration with the inlined monitor \( \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, S \rangle \sim \langle H \uplus mst, r, \hat{S} \rangle \) and the corresponding steps performed by the inlined monitor. Coupled configuration means that the monitor states are coupled and the current scope object reference \( r \) is pointing to the same scope in the heap memory, and \( \hat{S} \) is the instrumented version of statement \( S \) as defined by \( I \). We show that every execution of the VM monitor ending with a transition to the \( \Box \) statement, has an equivalent execution of the inlined monitor ending with a \( t \)-transition to the \( \Box \) statement. By applying rule \( \Gamma', \Lambda, \Delta, pc \vdash I(\Box) \Rightarrow I \Box \) we always obtain a valid transformation relation between \( S' \) and \( \hat{S}' \). Then we show that the monitor states are coupled, hence, the configuration of the two monitors are coupled.

For clause (b) of Theorem 2, our proof strategy will be based mainly on the application of the output relation \( \mid_{\text{out}} \) on both the events generated by the execution of the VM monitor \( \vec{\alpha} \) and the events generated by the inlined monitor \( \vec{\beta} \), showing that both are equal in all the different cases of statements \( S \).

For the sake of brevity, from now on, in the proof of clause (b) we will skip over the events corresponding to the statements that are the result of the inlining stage, e.g. the statements that modify \( mst = (\Gamma, \Lambda, \Delta, pc) \), since all of them will be internal events and not visible to any observer (attacker).

**proof:** By rule induction \( \rightarrow \) on the different cases:

We will start by the most interesting cases first.

**Case** \( \text{if} \{S_1\} \text{ else } \{S_2\} \):

We show the VM monitor execution assuming the case where the value of \( e \notin \{0, \text{null, undefined, false}\} \), the other case is straightforward.

\[
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \text{if} \{S_1\} \text{ else } \{S_2\} \rangle \rangle
\]

By rule T-If-Stmt in Table 4

\[
\begin{align*}
\langle \langle \Gamma, \Sigma, \Lambda', \Delta, pc' \rangle, \langle H, r, \text{if} (e) \{x; S_1\} \text{ else } \{x; S_2\} \rangle \rangle & \\
\text{where } \alpha_1 = b(e, S_2) & \\
\langle \langle \Gamma, \Sigma, \Lambda', \Delta', pc' \rangle, \langle H, r, x; S_1 \rangle \rangle & \\
\downarrow_j & \\
\langle \langle \Gamma', \Sigma', \Lambda', \Delta, pc' \rangle, \langle H, r, \Box; S_1 \rangle \rangle & \\
\end{align*}
\]

which defines \( \Gamma', \Sigma', \Lambda', S' = \Box \)

Now we show the inlined monitor (the instrumented code) execution until the first \( t \)-event:
Now we show the inlined monitor (the instrumented code) execution until the first t-event:

By applying rule $Γ', Λ', Δ, pc ⊢ t(□) \Rightarrow □$ we obtain a valid transformation relation between $S'$ and $\hat{S}'$ and by taking $Γ'_m = Γ' ⊔ Σ', Λ'_m = Λ'$, and $pc'_m = pc'$, we get $⟨Γ', Σ', Λ', Δ, pc⟩ \cong (mst' = (Γ'_m, Λ'_m, Δ_m, pc'_m))$, consequently we get $⟨(H, r, S'), (Γ', Σ', Λ', Δ, pc')⟩ \sim (H \uplus mst', r, \hat{S}')$.

If we apply the output relation $|_{out}$ on both of $\bar{α}$ and $\bar{β}$ we get $\bar{α} |_{out} \emptyset$ and $\bar{β} |_{out} \emptyset$, trivially $\emptyset = \emptyset$ since all the events that got generated by the two monitors are internal events $e$ as outlined in Table 8.

Case while(e) {S}:

We show the VM monitor execution assuming the more interesting case where the value of $e \in \{0, null, undefined, false\}$, in the other case the VM generates a $b(e, Skip)$ event which is less interesting since the VM monitor skips over it.

- By rule T-While in Table 4

$$⟨(Γ, Σ, Λ, Δ, pc), (H, r, while(e) \{S\})⟩$$

$$\xrightarrow{α_1}$$

where $α_1 = b(e, S)$

$$⟨(Γ, Σ, Λ', Δ', pc'), (H, r, □)⟩$$

- By rule T-While in Table 4

$$⟨(Γ', Σ', Λ', Δ, pc'), (H, r, □)⟩$$

$$= ⟨(Γ', Σ', Λ', Δ, pc), (H, r, S')⟩$$

which defines $Γ', Σ', S' = □$
Hybrid Flow-Sensitive Security Monitor for JavaScript

\[ H \sqsubseteq \text{mst}, r, "\Lambda \cdot \text{push}(\{\ell : \text{sec \_lvl}(e), \text{lbl} : \text{LOOP}\}); ..." \]

\[ \text{sk} \xrightarrow{b(e,S)} \langle H \sqsubseteq \text{mst}, r, "\Delta \cdot \text{push}(\{xs : \text{Collect}(S), \ell : \text{pc}\}); ..." \rangle \]

Assuming \( e \in \{0, \text{null}, \text{undefined}, \text{false}\} \)

\[ \text{sk} \xrightarrow{j} \langle H \sqsubseteq \text{mst} = (\Gamma', \Lambda', \Delta', \text{pc}'), r, "\text{if}(\text{should\_comp})\{\text{Comp}_\Lambda($should\_comp$.lbl, $\text{pc\.ℓ}$); ..." \rangle \]

\[ \text{sk} \xrightarrow{t} \langle H \sqsubseteq \text{mst} = (\Gamma', \Lambda', \Delta', \text{pc}'), r, \square \rangle \]

\[ = \langle H \sqsubseteq \text{mst}', r, \hat{S}' \rangle \text{ which defines } \text{mst}', \hat{S}' = \square \]

By applying rule \( \Gamma', \Lambda, \Delta, \text{pc} \vdash t(\square) \Rightarrow, \square \) we obtain a valid transformation relation between \( S' \) and \( \hat{S}' \) and by taking \( \Gamma'_\text{in} = \Gamma' \uplus \Sigma' \), we get \( (\Gamma', \Sigma', \Lambda, \Delta, \text{pc}) \equiv (\text{mst}' = (\Gamma'_\text{in}, \Lambda_\text{in}, \Delta_\text{in}, \text{pc}_\text{in})) \), consequently we get \( \langle (H, r, S'), (\Gamma', \Sigma', \Lambda, \Delta, \text{pc}) \rangle \sim \langle H \sqsubseteq \text{mst}', r, \hat{S}' \rangle \).

If we apply the output relation \( |_\text{out} \) on both of \( \vec{\alpha} \) and \( \vec{\beta} \) we get \( \vec{\alpha} |_\text{out} \emptyset \) and \( \vec{\beta} |_\text{out} \emptyset \), trivially \( \emptyset = \emptyset \) since all the events that got generated by the two monitors are internal events \( e \) as outlined in Table 8.

**Case** \( \text{for}(; e; )\{S\} \):

Similar to while loop case.

**Case** \( \text{for}(x \text{ in } e)\{S\} \):

Similar to while loop case.

**Case** \( \text{throw } e \):

We show the VM monitor execution first.

\[ \langle (\Gamma, \Sigma, \Lambda, \Delta, \text{pc}), \langle H, r, \text{throw } e \rangle \rangle \]

- By rule T-Throw in Table 5

\[ \alpha_1 \xrightarrow{} \langle (\Gamma, \Sigma, \Lambda', \Delta, \text{pc'}), \langle H, r, \text{throw } e \rangle \rangle \]

\[ \text{where } \alpha_1 = k(\text{TRY} | \text{FUNC}) \]

\[ \langle (\Gamma', \Sigma', \Lambda'', \Delta, \text{pc''}), \langle H, r, \square \rangle \rangle \]

\[ = \langle (\Gamma, \Sigma, \Lambda'', \Delta, \text{pc''}), \langle H, r, S' \rangle \rangle \text{ which defines } \Lambda'', \text{pc''}, S' = \square \]

Now we show the inlined monitor (the instrumented code) execution until the first \( t \)-event:
By applying rule $\Gamma, \Lambda'', \Delta, pc'' \vdash I(\square) \Rightarrow \square$ we obtain a valid transformation relation between $\hat{S}'$ and $\hat{S}'$ and by taking $\Lambda''_{in} = \Lambda''$, and $pc''_{in} = pc''$, we get $\langle H \cup \mathcal{S}'', \Gamma, \Lambda'', \Delta, pc'' \rangle \Rightarrow (mst' = (\Gamma_{in}, \Lambda''_{in}, \Delta_{in}, pc''_{in}))$, consequently we get $\langle \langle H \cup \mathcal{S}', r, \hat{S}' \rangle \rangle \sim \langle H \cup mst', r, \hat{S}' \rangle$.

If we apply the output relation $|_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$ we get $\vec{\alpha} |_{out} \emptyset$ and $\vec{\beta} |_{out} \emptyset$, trivially $\emptyset = \emptyset$ since all the events that got generated by the two monitors are internal events $e$ as outlined in Table 8.
Case return;
We show the VM monitor execution first.

By rule T-Return in Table 5

\[\langle\langle \Gamma, \Sigma, \Lambda', \Delta, pc\rangle, \langle H, r, return; \rangle \rangle\]

\[\alpha_1 \overset{\rightarrow}{\Rightarrow} \langle\langle \Gamma', \Sigma', \Lambda''', \Delta, pc''\rangle, \langle H, r, \square \rangle \rangle\]

\[\alpha_1 = k(\text{FUNC})\]

\[\langle\langle \Gamma, \Sigma, \Lambda'', \Delta, pc''\rangle, \langle H, r, S' \rangle \rangle \quad \text{which defines } \Lambda'', pc'', S' = \square\]

Now we show the inlined monitor (the instrumented code) execution until the first t-event:

\[\langle H \uplus mst, r, "$\text{old}_pc = \text{pc}; \ldots$" \rangle\]

\[\overset{s}{\Rightarrow} \langle H \uplus mst, r, "$\text{while}(\text{pc}.id !== \text{\textquote{\text{FUNC}}} )\{\text{\textquote{\text{pop}}}; \}; \ldots$" \rangle\]

\[\overset{sk}{\Rightarrow} \langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, "$\text{\textquote{\text{return}}; \checkmark" } \rangle\]

\[\overset{k(\text{FUNC})}{\Rightarrow} \langle H \uplus mst = (\Gamma, \Lambda'', \Delta, pc''), r, "$\text{\textquote{\text{return}}; \checkmark" } \rangle\]

\[\overset{t}{\Rightarrow} \langle H \uplus mst = (\Gamma, \Lambda'', \Delta, pc'') , r, \square \rangle\]

\[\overset{=}{\Rightarrow} \langle H \uplus mst', r, \hat{S}' \rangle \quad \text{which define } mst', \hat{S}' = \square\]

By applying rule \(\Gamma, \Lambda'', \Delta, pc'' \vdash I(\square) \Rightarrow I(\square)\) we obtain a valid transformation relation between \(S'\) and \(\hat{S}'\) and by taking \(\Lambda''_{in} = \Lambda''\), and \(pc''_{in} = pc''\), we get \(\langle\langle \Gamma, \Sigma, \Lambda'', \Delta, pc''\rangle \equiv (mst' = (\Gamma_{in}, \Lambda''_{in}, \Delta_{in}, pc''_{in}))\), consequently we get \(\langle\langle H, r, S' \rangle, \langle\langle \Gamma, \Sigma, \Lambda'', \Delta, pc'' \rangle \rangle \sim \langle H \uplus mst', r, \hat{S}' \rangle \).

If we apply the output relation \(|_{out}\) on both of \(\alpha\) and \(\beta\) we get \(\alpha |_{out} \theta\) and \(\beta |_{out} \theta\), trivially \(\theta = \emptyset\) since all the events that got generated by the two monitors are internal events \(\epsilon\) as outlined in Table 8.

Case return x;
We show the VM monitor execution first.
Hybrid Flow-Sensitive Security Monitor for JavaScript

- By rule T-Return in Table 5

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \text{return } x; \rangle \]

\[ \alpha_1 \rightarrow \langle \Gamma, \Sigma, \Lambda', \Delta, pc' \rangle, \langle H, r, \text{return } x; \rangle \]

where \( \alpha_1 = k(\text{FUNC}) \)

\[ \langle \Gamma', \Sigma', \Lambda'', \Delta, pc'' \rangle, \langle H, r, \square \rangle \]

\[ = \langle \Gamma, \Sigma, \Lambda'', \Delta, pc'' \rangle, \langle H, r, S' \rangle \]

which defines \( \Lambda'', pc'', S' = \square \)

Now we show the inlined monitor (the instrumented code) execution until the first t-event:

\[ \langle H \uplus \text{mst}, r, "\text{old}_{-} pc = \$pc; ..." \rangle \]

\[ \rightarrow \langle H \uplus \text{mst}, r, "\text{while}(\$pc.id \neq 'FUNC')\{\$\Lambda.pop(); \}; ..." \rangle \]

\[ \rightarrow \langle H \uplus \text{mst}, r, "$rx = \$scope(\$CS, 'x')[x]; ..." \rangle \]

\[ \rightarrow \langle H \uplus \text{mst} = (\Gamma, \Lambda', \Delta, pc'), r, "\text{return } x; \\checkmark" \rangle \]

\[ \rightarrow \langle H \uplus \text{mst} = (\Gamma, \Lambda'', \Delta, pc''), r, "\text{return } x; \\checkmark" \rangle \]

By applying rule \( \Gamma, \Lambda'', \Delta, pc'' \vdash t(\square) \Rightarrow \square \) we obtain a valid transformation relation between \( \mathcal{S}' \) and \( \mathcal{S}'' \) and by taking \( \Lambda_{in}' = \Lambda'' \)' and \( pc_{in}'' = pc'' \)' we get \( \langle \Gamma, \Sigma, \Lambda'', \Delta, pc'' \rangle \equiv \langle \text{mst}', \Lambda_{in}' = \Lambda'' \rangle \equiv \langle \text{mst}', \text{mst}' \rangle \).

If we apply the output relation \( |_{\text{out}} \) on both of \( \vec{\alpha} \) and \( \vec{\beta} \) we get \( \vec{\alpha} \mid_{\text{out}} \emptyset \) and \( \vec{\beta} \mid_{\text{out}} \emptyset \), trivially \( \emptyset = \emptyset \) since all the events that got generated by the two monitors are internal events \( \epsilon \) as outlined in Table 8.

**Case  continue;**

We show the VM monitor execution first.
By rule T-Continue in Table 5

\[ \langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, continue; \rangle \rangle \]

\( \xrightarrow{\alpha_1} \)

\[ \langle \langle \Gamma, \Sigma, \Lambda', \Delta, pc' \rangle, \langle H, r, continue; \rangle \rangle \]

where \( \alpha_1 = k(\text{LOOP}) \)

\[ \langle \langle \Gamma', \Sigma', \Lambda'', \Delta, pc'' \rangle, \langle H, r, \square \rangle \rangle \]

\[ = \]

\[ \langle \langle \Gamma, \Sigma, \Lambda'', \Delta, pc'' \rangle, \langle H, r, S' \rangle \rangle \]

which defines \( \Lambda'', pc'', S' = \square \)

Now we show the inlined monitor (the instrumented code) execution until the first t-event:

\[ \langle H \uplus mst, r, "$\text{old}_pc =$ pc; ..." \rangle \]

\( \xrightarrow{sk} \)

\[ \langle H \uplus mst, r, "\text{while}($\text{pc}_id \neq \text{LOOP'}\}\{\$\Lambda.pop(); \}; \text{..."} \rangle \]

\( \xrightarrow{sk} \)

\[ \langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, "$\Lambda[\$\Lambda.len - 2] = \{\ell : \$\Lambda[\$\Lambda.len - 2] \sqcup \text{old}_pc.\ell\}; \text{..."} \rangle \]

\( \xrightarrow{k(\text{LOOP})} \)

\[ \langle H \uplus mst = (\Gamma, \Lambda'', \Delta, pc''), r, "\text{continue; } \checkmark" \rangle \]

\( \xrightarrow{t} \)

\[ \langle H \uplus mst = (\Gamma, \Lambda'', \Delta, pc''), r, \square \rangle \]

\[ = \]

\[ \langle H \uplus mst', r, \checkmark' \rangle \]

which define \( mst', \checkmark' = \square \)

By applying rule \( \Gamma, \Lambda'', \Delta, pc'' \vdash I(\square) \Rightarrow \square \) we obtain a valid transformation relation between \( \checkmark' \) and \( \checkmark \) and by taking \( \Lambda''_\text{in} = \Lambda'' \), and \( pc''_\text{in} = pc'' \), we get \( \langle \Gamma, \Sigma, \Lambda'', \Delta, pc'' \rangle \cong (mst' = (\Gamma_\text{in}, \Lambda''_\text{in}, \Delta_\text{in}, pc''_\text{in})) \), consequently we get \( \langle (H, r, S''), (\Gamma, \Sigma, \Lambda'', \Delta, pc'') \rangle \sim \langle H \uplus mst', r, \checkmark' \rangle \).

If we apply the output relation \( |\text{out} \) on both of \( \bar{\alpha} \) and \( \bar{\beta} \) we get \( \bar{\alpha} |\text{out } \emptyset \) and \( \bar{\beta} |\text{out } \emptyset \), trivially \( \emptyset = \emptyset \) since all the events that got generated by the two monitors are internal events \( \epsilon \) as outlined in Table 8.

Case break;:
Same as previous case.

Case continue label;:
We show the VM monitor execution first.

- By rule T-Continue in Table 5

\[ \langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \text{continue label;} \rangle \rangle \]

\( \xrightarrow{\alpha_1} \)

\[ \langle \langle \Gamma, \Sigma, \Lambda', \Delta, pc' \rangle, \langle H, r, \text{continue label;} \rangle \rangle \]

where \( \alpha_1 = k(\text{label}) \)

\[ \langle \langle \Gamma', \Sigma', \Lambda'', \Delta, pc'' \rangle, \langle H, r, \square \rangle \rangle \]

\[ = \]

\[ \langle \langle \Gamma, \Sigma, \Lambda'', \Delta, pc'' \rangle, \langle H, r, S' \rangle \rangle \]

which defines \( \Lambda'', pc'', S' = \square \)
Now we show the inlined monitor (the instrumented code) execution until the first \( t \)-event:

\[
\langle H \uplus \text{mst}, r, \"$old\_pc = \$pc; \ldots\" \rangle
\]

\[
\text{sk} \rightarrow \langle H \uplus \text{mst}, r, \"while(\$pc.id !== 'label')\{\$Λ.pop(); \}; \ldots\" \rangle
\]

\[
\langle H \uplus \text{mst} = (\Gamma, \Lambda', \Delta, \text{pc}'), r, \\"\while(\$pc.id !== 'label')\{\$Λ.pop(); \}; \ldots\" \rangle
\]

\[
\text{sk} \rightarrow \langle H \uplus \text{mst} = (\Gamma, \Lambda', \Delta, \text{pc}'), r, \"continue;✓\" \rangle
\]

\[
k(\text{label}) \rightarrow \langle H \uplus \text{mst} = (\Gamma, \Lambda', \Delta, \text{pc}'), r, \text{true} \rangle
\]

\[
\text{t} \rightarrow \langle H \uplus \text{mst} = (\Gamma, \Lambda'', \Delta, \text{pc}''), r, \Box \rangle
\]

\[
= \langle H \uplus \text{mst}', r, \hat{\text{S}}' \rangle \quad \text{which define \text{mst}', \hat{\text{S}}'} = \Box
\]

By applying rule \( \Gamma, \Lambda', \Delta, \text{pc}'' \vdash I(\Box) \Rightarrow I(\Box) \) we obtain a valid transformation relation between \( \hat{\text{S}}' \) and \( \hat{\text{S}}' \) and by taking \( \Lambda''_{in} = \Lambda'', \) and \( \text{pc}''_{in} = \text{pc}'' \), we get \( (\text{mst}' = (\Gamma_{in}, \Lambda''_{in}, \Delta_{in}, \text{pc}''_{in})) \), consequently we get \( \langle (H, r, \hat{\text{S}}'), (\Gamma, \Sigma, \Lambda', \Delta, \text{pc}'') \rangle \sim \langle H \uplus \text{mst}', r, \hat{\text{S}}' \rangle \).

If we apply the output relation \( |_{out} \) on both of \( \vec{\alpha} \) and \( \vec{\beta} \) we get \( \vec{\alpha} |_{out} \emptyset \) and \( \vec{\beta} |_{out} \emptyset \), trivially \( \emptyset = \emptyset \) since all the events that got generated by the two monitors are internal events \( \epsilon \) as outlined in Table 8.

**Case** break label;

Same as the previous case.

**Case** with\((x)\{S\}:

We show the VM monitor execution first.

- By rule T-With in Table 5

\[
\langle (\Gamma, \Sigma, \Lambda, \Delta, \text{pc}), (H, r, \text{with}(x)\{S\}) \rangle
\]

\[
\text{sk} \rightarrow \langle (\Gamma, \Sigma, \Lambda', \Delta, \text{pc}'), (H, r, \text{with}(x)\{S\}) \rangle
\]

assuming expression \( x \) points to an object-reference \( r_o \), otherwise, it is a runtime error.

\[
\langle (\Gamma, \Sigma, \Lambda', \Delta, \text{pc}'), (H, r_o, S) \rangle
\]

\[
\text{u}_1 \rightarrow \langle (\Gamma', \Sigma', \Lambda', \Delta, \text{pc}'), (H, r_o, \Box) \rangle
\]

- By rule T-With in Table 5

\[
\langle (\Gamma', \Sigma', \Lambda, \Delta, \text{pc}), (H, r, \Box) \rangle
\]

\[
= \langle (\Gamma', \Sigma', \Lambda, \Delta, \text{pc}), (H, r, \text{S'}) \rangle \quad \text{which defines \( \Gamma', \Sigma', \text{S'} = \Box \)
\]
Hybrid Flow-Sensitive Security Monitor for JavaScript

\[
\{H \uplus \textit{mst}, r, "\textit{\$Lambda.push}\{\ell : \textit{pc.}\ell \sqcup \textit{sec lvl}(x), \textit{lbl} : 'WITH'}\}; ...\}
\]

\[
\{H \uplus \textit{mst}, r, "\textit{\$ro = \$scope('x', \$\textit{CS})['x']}; \textit{\$ro.scope = \$\textit{CS}; ...}\}
\]

\[
\{H \uplus \textit{mst} = (\Gamma', \Lambda', \Delta, pc'), r, "\textit{\$Lambda.pop}; \checkmark"\}
\]

\[
\{H \uplus \textit{mst} = (\Gamma', \Lambda', \Delta, pc'), r, "\textit{\$Lambda.pop}; \checkmark"\}
\]

\[
\{H \uplus \textit{mst} = (\Gamma', \Lambda', \Delta, pc'), r, "\textit{\$Lambda.pop}; \checkmark"\}
\]

\[
\{H \uplus \textit{mst} = (\Gamma, \Lambda, \Delta, pc), r, \square\}
\]

\[
\{H \uplus \textit{mst}', r, \hat{S}'\} \quad \text{which define} \ mst', \hat{S}' = \square
\]

By applying rule $\Gamma', \Lambda, \Delta, pc \vdash I(\square) \Rightarrow \square$ we obtain a valid transformation relation between $S'$ and $\hat{S}'$ and by taking $\Gamma'_m = \Gamma' \uplus \Sigma'$, we get $\langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle \equiv (\textit{mst}' = (\Gamma'_m, \Lambda_m, \Delta_m, pc_m))$, consequently we get $\langle \{(H, r, S'), (\Gamma', \Sigma', \Lambda, \Delta, pc)\} \sim \langle H \uplus \textit{mst}', r, \hat{S}'\rangle$.

If we assume that the observable events generated by the VM monitor as the result of the execution of the body of the with-statement $S$ is $\vec{w}_1$ and the observable events generated by the inlined monitor as $\vec{w}_2$, and we apply the output relation $|_{\text{out}}$ on both of $\vec{\alpha}$ and $\vec{\beta}$ we get $\vec{\alpha} |_{\text{out}} \vec{w}_1$ and $\vec{\beta} |_{\text{out}} \vec{w}_2$ since the remaining events are internal events, then either $\vec{w}_1 = \vec{w}_2$ or both are empty due to lack of any observable events when statement $S$ was executed. The proof of that is based on the structural induction and semantics of statement $S$.

**Case** try\{S\}:

We show the VM monitor execution first.

- By rule T-Try in Table 5

\[
\langle (\Gamma, \Sigma, \Lambda, \Delta, pc), (H, r, \text{try}\{S\})\rangle
\]

\[
\langle (\Gamma, \Sigma, \Lambda', \Delta, pc'), (H, r, \text{try}\{S\})\rangle
\]

\[
\langle (\Gamma, \Sigma, \Lambda', \Delta, pc'), (H, r, S)\rangle
\]

- By rule T-With in Table 5

\[
\langle (\Gamma', \Sigma', \Lambda', \Delta, pc'), (H, r, \square)\rangle
\]

\[
\langle (\Gamma', \Sigma', \Lambda, \Delta, pc), (H, r, \square)\rangle
\]

\[
\langle (\Gamma', \Sigma', \Lambda, \Delta, pc), (H, r, S')\rangle \quad \text{which defines} \ (\Gamma', \Sigma', S' = \square
\]

Now we show the inlined monitor (the instrumented code) execution until the first $t$-event:
\[ \langle H \uplus \text{mst}, r, "\text{try}\{\ell : \text{pc.}\ell, \text{lbl} : \text{'TRY'}\}\}; S; \Lambda.\text{pop}(); \text{✓} \rangle \]

\[ \xrightarrow{sk} \quad \langle H \uplus \text{mst} = (\Gamma, \Lambda', \Delta, pc'), r, "S; \Lambda.\text{pop}(); \text{✓} \rangle \]

\[ \xrightarrow{\vec{w}_2} \quad \langle H \uplus \text{mst} = (\Gamma', \Lambda', \Delta, pc'), r, "S; \Lambda.\text{pop}(); \text{✓} \rangle \]

\[ \xrightarrow{sk} \quad \langle H \uplus \text{mst} = (\Gamma', \Lambda, \Delta, pc), r, "S; \Lambda.\text{pop}(); \text{✓} \rangle \]

\[ \xrightarrow{t} \quad \langle H \uplus \text{mst} = (\Gamma', \Lambda, \Delta, pc), r, \Box \rangle \]

\[ = \quad \langle H \uplus \text{mst}', r, \hat{S}' \rangle \quad \text{which define mst', \hat{S}' = \Box} \]

By applying rule \(\Gamma', \Lambda, \Delta, pc \vdash I(\Box) \Rightarrow \Box\) we obtain a valid transformation relation between \(S'\) and \(\hat{S}'\) and by taking \(\Gamma'_{in} = \Gamma' \uplus \Sigma\), we get \(\langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle \equiv (\text{mst'} = (\Gamma'_{in}, \Lambda_{in}, \Delta_{in}, pc_{in}))\), consequently we get \(\langle H, r, S'\rangle, \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle \sim \langle H \uplus \text{mst}', r, \hat{S}' \rangle\).

If we assume that the observable events generated by the VM monitor as the result of the execution of the body of the try-statement \(S\) is \(\vec{w}_1\) and the observable events generated by the inlined monitor as \(\vec{w}_2\), and we apply the output relation \(|_{\text{out}}\) on both of \(\vec{a}\) and \(\vec{b}\) we get \(\vec{a} \mid_{\text{out}} \vec{w}_1\) and \(\vec{b} \mid_{\text{out}} \vec{w}_2\) since the remaining events are internal events, then either \(\vec{w}_1 = \vec{w}_2\) or both are empty due to lack of any observable events when statement \(S\) was executed. The proof of that is based on the structural induction and semantics of statement \(S\).

**Case catch(x){S}**: 

It is important to note that every catch-clause must be preceded with a try-clause in a try-catch-finally statement and the execution of the body of a catch-clause happens when a statement in the body of the try-clause throws an exception. This means that the initial monitor state of the catch-clause will \(\langle \Gamma, \Sigma, \Lambda', \Delta, pc' \rangle\) in the case of the VM monitor and \(\langle \Gamma, \Sigma', \Lambda, \Delta, pc' \rangle\) for the inlined monitor.

We show the VM monitor execution first.

\[ \langle \Gamma, \Sigma, \Lambda', \Delta, pc' \rangle, \langle H, r, \text{catch}(x)\{S; \} \rangle \]

\[ \xrightarrow{sk} \quad \langle \Gamma, \Sigma, \Lambda', \Delta, pc' \rangle, \langle H, r, S \rangle \]

\[ \xrightarrow{\vec{w}_1} \quad \langle \Gamma', \Sigma', \Lambda', \Delta, pc', \langle H, r, \Box \rangle \rangle \]

- By rule T-Catch in Table 5

\[ \langle \Gamma', \Sigma', \Lambda', \Delta, pc, \langle H, r, \Box \rangle \rangle \]

\[ = \quad \langle \Gamma', \Sigma', \Lambda, \Delta, pc, \langle H, r, \Box \rangle \rangle \quad \text{which defines \(\Gamma', \Sigma', S' = \Box\)} \]

Now we show the inlined monitor (the instrumented code) execution until the first t-event:
\[
\langle H \triangledown \text{mst} = (\Gamma, \Lambda', \Delta, pc'), r, "catch(x)\{S; $\Lambda.pop(); \}\checkmark"angle
\]

\[
\text{sk} \rightarrow \langle H \triangledown \text{mst} = (\Gamma, \Lambda', \Delta, pc'), r, "S; $\Lambda.pop(); \}\checkmark"angle
\]

\[
\text{w}^2 \rightarrow \langle H \triangledown \text{mst} = (\Gamma', \Lambda', \Delta, pc'), r, "\checkmark"angle
\]

\[
\text{sk} \rightarrow \langle H \triangledown \text{mst} = (\Gamma', \Lambda, \Delta, pc), r, "\checkmark"angle
\]

\[
\text{t} \rightarrow \langle H \triangledown \text{mst} = (\Gamma', \Lambda, \Delta, pc), r, "\Box" \rangle
\]

\[
= \langle H \triangledown \text{mst}', r, "\hat{S}'" \rangle \quad \text{which define } mst', "\hat{S}'" = "\Box"
\]

By applying rule \(\Gamma', \Lambda, \Delta, pc \vdash I(\Box) \Rightarrow \Box\) we obtain a valid transformation relation between \(S'\) and \(\hat{S}'\) and by taking \(\Gamma'_\text{in} = \Gamma' \cup \Sigma'\), we get \(\langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle \approx (mst' = (\Gamma'_\text{in}, \Lambda_m, \Delta_m, pc_\text{in}))\), consequently we get \(\langle \langle H, r, S' \rangle, \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle \rangle \approx \langle H \triangledown \text{mst}' , r, \hat{S}' \rangle\).

If we assume that the observable events generated by the VM monitor as the result of the execution of the body of the catch-clause \(S\) is \(\vec{w}_1\) and the observable events generated by the inlined monitor as \(\vec{w}_2\), and we apply the output relation \(|_{\text{out}}\) on both of \(\vec{\alpha}\) and \(\vec{\beta}\) we get \(\vec{\alpha} |_{\text{out}} \vec{w}_1\) and \(\vec{\beta} |_{\text{out}} \vec{w}_2\) since the remaining events are internal events, then either \(\vec{w}_1 = \vec{w}_2\) or both are empty due to lack of any observable events when statement \(S\) was executed. The proof of that is based on the structural induction and semantics of statement \(S\).

**Case finally\{S\}:**

It is important to note that every catch-clause must be preceded with a try-clause in a try-catch-finally statement and the execution of the body of a catch-clause happens when a statement in the body of the try-clause throws an exception. This means that the initial monitor state of the catch-clause will \(\langle \Gamma, \Sigma, \Lambda, \Delta, pc' \rangle\) in the case of the VM monitor and \(\langle \Gamma', \Lambda', \Delta, pc' \rangle\) for the inlined monitor.

We show the VM monitor execution first.

\[
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \text{finally}\{S; \}\rangle \rangle
\]

\[
\text{sk} \rightarrow \langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, S \rangle \rangle
\]

\[
\text{w}^1 \rightarrow \langle \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H, r, "\Box" \rangle \rangle
\]

\[
= \langle \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H, r, S' \rangle \rangle \quad \text{which defines } \Gamma', \Sigma', S' = "\Box"
\]

Now we show the inlined monitor (the instrumented code) execution until the first \(t\)-event:
\[ \langle H \uplus \text{mst} = (\Gamma, \Lambda', \Delta, \text{pc}) \rangle, r, \text{"catch}(x)\{ S; \} \checkmark \rangle \]

\[ \xrightarrow{sk} \langle H \uplus \text{mst} = (\Gamma, \Lambda', \Delta, \text{pc}) \rangle, r, \text{"catch}(x)\{ } S; \} \checkmark \rangle \]

\[ \text{catch}(x) \{
S;
\}
\]

\[ \checkmark \]

\[ \langle H \uplus \text{mst} = (\Gamma, \Lambda, \Delta, \text{pc}) \rangle, r, \text{"catch}(x)\{ } S; \} \checkmark \rangle \]

\[ \checkmark \]

\[ \langle H \uplus \text{mst} = (\Gamma', \Lambda, \Delta, \text{pc}) \rangle, r, \text{"catch}(x)\{ } S; \} \checkmark \rangle \]

\[ \checkmark \]

\[ \langle H \uplus \text{mst} = (\Gamma', \Lambda', \Delta, \text{pc}) \rangle, r, \text{"catch}(x)\{ } S; \} \checkmark \rangle \]

\[ \checkmark \]

\[ \langle H \uplus \text{mst} = (\Gamma', \Lambda, \Delta, \text{pc}) \rangle, r, \square \rangle \]

\[ = \]

\[ \langle H \uplus \text{mst}', r, \hat{S}' \rangle \] which define \( \text{mst}', \hat{S}' = \square \)

By applying rule \( \Gamma', \Lambda, \Delta, \text{pc} \vdash I(\square) \Rightarrow I(\square) \) we obtain a valid transformation relation between \( \text{mst}' \) and \( \hat{S}' \) and by taking \( \Gamma'_\text{in} = \Gamma' \oplus \Sigma' \), we get \( \langle \text{mst}' = (\Gamma'_\text{in}, \Lambda_\text{in}, \Delta_\text{in}, \text{pc}_\text{in}) \rangle \), consequently we get \( \langle \langle H, r, S' \rangle, \langle \Gamma', \Sigma', \Lambda, \Delta, \text{pc} \rangle \rangle \sim \langle H \uplus \text{mst}', r, \hat{S}' \rangle \).

If we assume that the observable events generated by the VM monitor as the result of the execution of the body of the finally-clause \( S \) is \( \vec{w}_1 \) and the observable events generated by the inlined monitor as \( \vec{w}_2 \), and we apply the output relation \( |\text{out} \) on both of \( \vec{a} \) and \( \vec{b} \) we get \( \vec{a} \mid_{\text{out}} \vec{w}_1 \) and \( \vec{b} \mid_{\text{out}} \vec{w}_2 \) since the remaining events are internal events, then either \( \vec{w}_1 = \vec{w}_2 \) or both are empty due to lack of any observable events when statement \( S \) was executed. The proof of that is based on the structural induction and semantics of statement \( S \).

**Case** output\( _\ell (e) \):

The VM monitor execution will depend on the chosen policy:

**Case** OutputFailStop:

\[ \langle \langle \Gamma, \Sigma, \Lambda, \Delta, \text{pc} \rangle, \langle H, r, \text{output}_{\ell}(e) \rangle \rangle \]

\[ \xrightarrow{\alpha(e,va)} \]

\[ \xrightarrow{\tau \alpha(va)} \]

\[ \text{in case } \tau \sqsubseteq \text{pc.}\ell \sqsubseteq \ell, \text{where } \tau \text{ is the security level of } e \]

\[ \langle \langle \Gamma, \Sigma, \Lambda, \Delta, \text{pc} \rangle, \langle H, r, \square \rangle \rangle \]

\[ = \]

\[ \langle \langle \Gamma, \Sigma, \Lambda, \Delta, \text{pc} \rangle, \langle H, r, S' \rangle \rangle \] which defines \( S' = \square \)

Otherwise:

The VM monitor will get stuck as outlined in Table 11 rule M-OutputFailStop

Now we show the inlined monitor (the instrumented code) execution for the same case:
\( (H \uplus \text{mst} = (\Gamma, \Lambda, \Delta, pc), r, "if (\text{sec}_l(e) \cup \text{pc}.\ell \subseteq \ell) \{\text{output}_e(e); \checkmark\}) \) \\
\qquad \text{else } \text{\$diverge}\$; ") \\
\xrightarrow{sk} \langle H \uplus \text{mst} = (\Gamma, \Lambda, \Delta, pc), r, "\text{output}_e(e); \checkmark" \rangle \\
\text{in case where } sec_l(e) \cup \text{pc}.\ell \subseteq \ell \\
\xrightarrow{o_l(e,va)} \langle H \uplus \text{mst} = (\Gamma, \Lambda, \Delta, pc), r, "\checkmark" \rangle \\
\xrightarrow{t} \langle H \uplus \text{mst} = (\Gamma, \Lambda, \Delta, pc), r, \square \rangle \\
= \langle H \uplus \text{mst}, r, \hat{S}' \rangle \quad \text{which define } \hat{S}' = \square \\
\text{Otherwise:} \\
\langle H \uplus \text{mst} = (\Gamma, \Lambda, \Delta, pc), r, \text{\$diverge}\$ \rangle, \\
\text{where } \text{\$diverge}\$ \text{ means stuck or lack of progress.} \\

In case where \( sec_l(e) \cup \text{pc}.\ell \subseteq \ell \) is true, if we apply rule \( \Gamma, \Lambda, \Delta, pc \vdash I(\square) \Rightarrow_i \square \) \\
we obtain a valid transformation relation between \( S' \) and \( \hat{S}' \) and we get \( \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle \cong (\text{mst} = (\Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in})) \), consequently we get \( \langle (H, r, S'), (\Gamma, \Sigma, \Lambda, \Delta, pc) \rangle \sim \langle H \uplus \text{mst}, r, \hat{S}' \rangle \) \\
\text{hence, clause (a) of Theorem 2 holds .} \\

If we apply the output relation \( |_{out} \) on both of \( \vec{\alpha} \) and \( \vec{\beta} \) we get \( \vec{\alpha} |_{out} \alpha_l(va) \) and \( \vec{\beta} |_{out} \alpha_l(va) \), trivially \( \alpha_l(va) = \alpha_l(va) \) hence, clause (b) of Theorem 2 holds.

In the case where \( sec_l(e) \cup \text{pc}.\ell \subseteq \ell \) is \text{false}, both monitors are stuck.
Now we show the inlined monitor (the instrumented code) execution for the same case:

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \text{output}_e(e) \rangle \]

\[ \overset{\alpha(e, va)}{\rightarrow} \]

in case \( \tau \sqcup pc.\ell \subseteq \ell \), where \( \tau \) is the security level of \( e \)

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \square \rangle \]

\[ = \]

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, S' \rangle \]

which defines \( S' = \square \)

Otherwise:

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \text{output}_e(e) \rangle \]

\[ \overset{\alpha(e, va)}{\rightarrow} \text{nothing} \]

in case \( \tau \sqcup pc.\ell \not\subseteq \ell \), where \( \tau \) is the security level of \( e \)

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \square \rangle \]

\[ = \]

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, S' \rangle \]

which defines \( S' = \square \)

Now we show the inlined monitor (the instrumented code) execution for the same case:

\[ \langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"if (secJvl(e) \sqcup \$pc.\ell \subseteq \ell) \{ output}_e(e) \} else Skip; \text{✓"} } \rangle \]

\[ \overset{sk}{\rightarrow} \]

in case where \( \text{secJvl(e) \sqcup \$pc.\ell \subseteq \ell} \)

\[ \langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"if (secJvl(e) \sqcup \$pc.\ell \subseteq \ell) \{ output}_e(e) ; \} else Skip; \text{✓" } \rangle \]

\[ \overset{t}{\rightarrow} \]

\[ \langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \square \rangle \]

\[ = \]

\[ \langle H \uplus mst, r, \hat{S} \rangle \]

which define \( \hat{S} = \square \)

Otherwise:

\[ \langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"if (secJvl(e) \sqcup \$pc.\ell \subseteq \ell) \{ output}_e(e) ; \} else Skip; \text{✓" } \rangle \]

\[ \overset{sk}{\rightarrow} \]

in case where \( \text{secJvl(e) \sqcup \$pc.\ell \not\subseteq \ell} \)

\[ \langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"Skip; \text{✓" } \rangle \}

\[ \overset{sk}{\rightarrow} \]

\[ \langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{✓" } \rangle \]

\[ = \]

\[ \langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \square \rangle \]

\[ = \]

\[ \langle H \uplus mst, r, \hat{S} \rangle \]

which define \( \hat{S} = \square \)

In case where \( \text{secJvl(e) \sqcup \$pc.\ell \subseteq \ell} \) is true or false, if we apply rule \( \Gamma, \Lambda, \Delta, pc \vdash I (\square) \Rightarrow \)

\( \square \) we obtain a valid transformation relation between \( S' \) and \( \hat{S}' \) and we get \( \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle \equiv (mst = (\Gamma_m, \Lambda_m, \Delta_m, pc_m)) \), consequently we get \( \langle (H, r, S'), (\Gamma, \Sigma, \Lambda, \Delta, pc) \rangle \sim \langle H \uplus mst, r, \hat{S}' \rangle \) hence, clause (a) of Theorem 2 holds.

If we apply the output relation \( \text{out} \) on both of \( \alpha \) and \( \beta \) we either get \( \alpha \mid_{\text{out}} \alpha (va) \) and \( \beta \mid_{\text{out}} \alpha (va) \) when \( \text{secJvl(e) \sqcup \$pc.\ell \subseteq \ell} \) is true, or we get \( \alpha \mid_{\text{out}} \emptyset \) and \( \beta \mid_{\text{out}} \emptyset \) when \( \text{secJvl(e) \sqcup \$pc.\ell \not\subseteq \ell} \) is false. Trivially, in the first case \( \alpha (va) = \alpha (va) \) and in the second case \( \emptyset = \emptyset \), hence, clause (b) of Theorem 2 holds.
Hybrid Flow-Sensitive Security Monitor for JavaScript

Case OutputDefault:

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \{ output_\ell(e) \} \]

\[ [e, va] \rightarrow_{o_\ell} [va] \]

in case \( \tau \sqcup pc.\ell \sqsubseteq \ell \), where \( \tau \) is the security level of \( e \)

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \{ H, r, \square \} \]

\[ = \]

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \{ H, r, S' \} \]

which defines \( S' = \square \)

Otherwise:

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \{ H, r, output_\ell(e) \} \]

\[ [e, va] \rightarrow_{o_\ell} [D] \]

in case \( \tau \sqcup pc.\ell \not\sqsubseteq \ell \), where \( \tau \) is the security level of \( e \)

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \{ H, r, \square \} \]

\[ = \]

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \{ H, r, S' \} \]

which defines \( S' = \square \)

Now we show the inlined monitor (the instrumented code) execution for the same case:

\[ \langle H \uplus mst \rangle = \langle \Gamma, \Lambda, \Delta, pc, r \rangle \]

\[ \{ output_\ell(e); \} \]

\[ sk \rightarrow \]

in case where \( sec \sqcup l \uplus \sqsubseteq l \)

\[ \langle H \uplus mst \rangle = \langle \Gamma, \Lambda, \Delta, pc, r \rangle \]

\[ \{ output_\ell(e); \} \]

\[ \Lambda \rightarrow \]

\[ \langle H \uplus mst \rangle = \langle \Gamma, \Lambda, \Delta, pc, r, \square \rangle \]

\[ = \]

\[ \langle H \uplus mst \rangle = \langle \Gamma, \Lambda, \Delta, pc, r, \square \rangle \]

which define \( \hat{S}' = \square \)

Otherwise:

\[ \langle H \uplus mst \rangle = \langle \Gamma, \Lambda, \Delta, pc, r \rangle \]

\[ \{ output_\ell(e); \} \]

\[ sk \rightarrow \]

in case where \( sec \sqcup l \uplus \sqsubseteq l \)

\[ \langle H \uplus mst \rangle = \langle \Gamma, \Lambda, \Delta, pc, r \rangle \]

\[ \{ output_\ell(D); \} \]

\[ \Lambda \rightarrow \]

\[ \langle H \uplus mst \rangle = \langle \Gamma, \Lambda, \Delta, pc, r, \square \rangle \]

\[ = \]

\[ \langle H \uplus mst \rangle = \langle \Gamma, \Lambda, \Delta, pc, r, \square \rangle \]

which define \( \hat{S}' = \square \)

In case where \( sec \sqcup l \uplus \sqsubseteq l \) is true or false, if we apply rule \( \Gamma, \Lambda, \Delta, pc \vdash I(\square) \Rightarrow \)

\( \square \) we obtain a valid transformation relation between \( \hat{S}' \) and \( \hat{S}' \) and we get \( \langle H, \Gamma, \Lambda, \Delta, pc \rangle \Rightarrow mst = (mst) \)

\( \Rightarrow (mst) \rangle \Rightarrow \) consequently we get \( \langle H, r, S' \rangle, \langle H, \Gamma, \Lambda, \Delta, pc \rangle \rangle \sim \langle H \uplus mst, r, \hat{S}' \rangle \)

hence, clause (a) of Theorem 2 holds.

If we apply the output relation \( |_{out} \) on both of \( \hat{\alpha} \) and \( \hat{\beta} \) we either get \( \hat{\alpha} |_{out} o_\ell(va) \) and \( \hat{\beta} |_{out} o_\ell(va) \) when \( sec \sqcup l \uplus \sqsubseteq l \) is true, or we get \( \hat{\alpha} |_{out} o_\ell(D) \) and \( \hat{\beta} |_{out} o_\ell(D) \)
when \( \text{sec.var}(e) \sqcup \text{sec.ell} \sqsubseteq \ell \) is false. Trivially, in the first case \( \alpha_{\ell}(va) = \alpha_{\ell}(va) \) and in the second case \( \alpha_{\ell}(D) = \alpha_{\ell}(D) \), hence, clause (b) of Theorem 2 holds.

**Case** foo = function(\(\vec{x}\))\{S\}; | function foo(\(\vec{x}\))\{S\};:

VM monitor execution:

\[
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \text{func foo}(\vec{x})\{S\} \rangle \rangle
\]

by Definition 5

\[
\Rightarrow_{\text{FuncLit}}^{sk} \langle \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H', r, \Box \rangle \rangle
\]

= \[
\langle \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H', r, S' \rangle \rangle
\]

which defines \( \Gamma', \Sigma', H', S' = \Box \)

Now we show the inlined monitor (the instrumented code) execution for the same case:

\[
\langle H \sqcup \text{mst} = (\Gamma, \Lambda, \Delta, pc), r, "\text{function foo}(\vec{x})\{I(S')\}; ...") \rangle
\]

\[
\Rightarrow_{\text{sk}}^{sk} \langle H' \sqcup \text{mst} = (\Gamma, \Lambda, \Delta, pc), r, "\text{strict CS}[]\{\text{foo}\} = \{x_1 : \text{pc.ell}, ..., x_n : \text{pc.ell}, $\text{scope} : $\text{pc.ell}, \text{prototype} : \{\Sigma : \text{pc.ell}, \Sigma : \text{pc.ell}; \checkmark") \rangle
\]

\[
\Rightarrow_{\text{sk}}^{sk} \langle H' \sqcup \text{mst} = (\Gamma', \Lambda, \Delta, pc), r, \checkmark") \rangle
\]

= \[
\langle H' \sqcup \text{mst} = (\Gamma', \Lambda, \Delta, pc), r, \Box \rangle
\]

which defines \( H', \text{mst}', \hat{S}' = \Box \)

If we apply rule \( \Gamma', \Lambda, \Delta, pc \vdash I(\Box) \Rightarrow, \Box \) we obtain a valid transformation relation between \( S' \) and \( \hat{S}' \) and we get \( \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle \cong (\text{mst}' = (\Gamma'_m, \Lambda_m, \Delta_m, pc_{in})) \), consequently we get \( \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle \langle H', r, S' \rangle \rangle \sim \langle H' \sqcup \text{mst}', r, \hat{S}' \rangle \) hence, clause (a) of Theorem 2 holds .

If we apply the output relation \( |_{\text{out}} \) on both of \( \alpha \) and \( \beta \) we get \( \alpha |_{\text{out}} \emptyset \) and \( \beta |_{\text{out}} \emptyset \), trivially \( \emptyset = \emptyset \) since the two monitors skips over function literal statements.

**Case** obj = \( \{pn_1 : e_1, ..., pn_n : e_n\} ; : \)

VM monitor execution:

\[
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \text{obj} = \{pn_1 : e_1, ..., pn_n : e_n\}; \rangle \rangle
\]

by Definition 3

\[
\Rightarrow_{\text{ObjectLit}}^{Ob} \langle \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H', r, \Box \rangle \rangle
\]

= \[
\langle \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H', r, S' \rangle \rangle
\]

which defines \( \Gamma', \Sigma', H', S' = \Box \)

Now we show the inlined monitor (the instrumented code) execution for the same case:
Hybrid Flow-Sensitive Security Monitor for JavaScript

\[ \langle H \uplus \text{mst} = (\Gamma, \Lambda, \Delta, pc), r, "\text{obj} = \{pn_1 : e_1, ... , pn_n : e_n\}; ..." \rangle \]

\[ \xrightarrow{sk} \]

\[ \langle H' \uplus \text{mst} = (\Gamma', \Lambda, \Delta, pc), r, "\text{CS['foo'] = \{pn_1 : sec lvl(e_1) \cup \$pc.\ell, ... , pn_n : sec lvl(e_n) \cup \$pc.\ell\}; $\text{\checkmark}"} \rangle \]

\[ \xrightarrow{t} \]

\[ \langle H' \uplus \text{mst} = (\Gamma', \Lambda, \Delta, pc), r, $\text{\checkmark}$ \rangle \]

which defines \( H', \text{mst}', \hat{S}' = $\text{\checkmark}$

By applying rule \( \Gamma', \Lambda, \Delta, pc \vdash I(\Box) \Rightarrow \Box \) we obtain a valid transformation relation between \( S' \) and by taking \( \Gamma'_m = \Gamma'_m \uplus \Sigma_m, \hat{S}' \) and we get \( \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle \shortequivalent (\text{mst'} = (\Gamma'_m, \Lambda_m, \Delta_m, pc_m) \rangle \), consequently we get \( \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle \rangle \sim \langle H' \uplus \text{mst'} , r, \hat{S}' \rangle \rangle \) hence, clause (a) of Theorem 2 holds.

If we apply the output relation \( |_{out} \) on both of \( \alpha \) and \( \beta \) we get \( \alpha |_{out} \emptyset \) and \( \beta |_{out} \emptyset \), trivially \( \emptyset = \emptyset \) since the two monitors skips over object literal statements hence, clause (b) of Theorem 2 holds.

Case \( x = [e_0, e_1, ..., e_n] \) :

Similar to Object literal case with the difference of apply relation \( \triangleright_{\text{ArrayLit}} \) outlined in Definition 4.

Case \( x = f(\vec{e}); \) :

VM monitor execution:

\[ \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle \langle H, r, x = f(\vec{e}) ; \rangle \]

by Definition 6

\[ \xrightarrow{\triangleright_{\text{FuncCall}}} \]

\[ \langle \hat{\Gamma}, \hat{\Sigma}, \Lambda', \Delta, pc' \rangle \langle H, r, x = va \rangle \]

where \( \Lambda'.pop() \) and \( r_x \) is a reference to variable \( x \).

\[ \xrightarrow{a_v(x, va)} \]

\[ \langle \hat{\Gamma}(r_x)[x \mapsto pc.\ell \cup \tau_{ret}], \hat{\Sigma}, \Lambda, \Delta, pc', \langle \hat{H}(r_x)[x \mapsto va] , r, \Box \rangle \rangle \]

which defines \( \hat{\Gamma}', \hat{\Sigma}', \hat{H}', S' = \Box \)

Now we show the inlined monitor (the instrumented code) execution for the same case:
\[
\langle H \triangleright mst = (\Gamma, \Lambda, \Delta, pc), r, "rf = \$scope(\$CS', f')[x]; \$rf.scope = \$CS; ...angle \]

\[
\xrightarrow{sk}
\langle H \triangleright mst = (\Gamma, \Lambda, \Delta, pc), r, "$rf = \$scope(\$CS', f')[x]; \$rf.scope = \$CS; ...angle
\]

\[
\xrightarrow{sk}
\langle H \triangleright mst = (\Gamma, \Lambda, \Delta, pc), r, "$rf = \$scope(\$CS', f')[x]; \$rf.scope = \$CS; ...angle
\]

\[
\xrightarrow{sk}
\langle H \triangleright mst = (\Gamma, \Lambda, \Delta, pc), r, "$rf = \$scope(\$CS', f')[x]; \$rf.scope = \$CS; ...angle
\]

\[
\xrightarrow{sk}
\langle H \triangleright mst = (\Gamma', \Lambda', \Delta', pc'), r, "$rf = \$scope(\$CS', f')[x]; \$rf.scope = \$CS; ...angle
\]

\[
\langle H \triangleright mst = (\Gamma', \Lambda', \Delta', pc), r, \square \rangle = \langle H' \triangleright mst = (\Gamma', \Lambda', \Delta', pc), r, \hat{S}' \rangle \quad \text{which defines } \hat{H}', \ mst = (\Gamma', \Lambda, \Delta, pc), \hat{S}' = \square
\]

By applying rule \(\hat{\Gamma}', \Lambda, \Delta, pc \vdash t(\square) \Rightarrow_1 \square\) we obtain a valid transformation relation between \(S'\) and \(\hat{S}'\) and by taking \(\hat{\Gamma}' = \hat{\Gamma}' \triangleright \hat{\Sigma}'\), we get \((\hat{\Gamma}', \hat{\Sigma}', \Lambda, \Delta, pc) \equiv (mst' = (\hat{\Gamma}' \downarrow in, \Lambda_{in}, \Delta_{in}, pc_{in}))\), consequently we get \((\hat{\Gamma}', \hat{\Sigma}', \Lambda, \Delta, pc), \langle \langle H', r, S' \rangle \rangle \sim \langle H' \triangleright mst' \rangle, r, \hat{S}' \rangle\) hence, clause (a) of Theorem 2 holds.

If we apply the output relation \(|_{\text{out}}\) on both of \(\hat{\alpha}\) and \(\hat{\beta}\) we get \(\hat{\alpha} \mid_{\text{out}} \emptyset\) and \(\hat{\beta} \mid_{\text{out}} \emptyset\), trivially \(\emptyset = \emptyset\) since the two monitors did not generate any observable events hence, clause (b) of Theorem 2 holds.

**Case** \(x = y.f(\overline{e}); \mid x = y[f'[\overline{e}]](\overline{e});:\)

Similar to the previous case with the difference of the variable \texttt{this} will be pointing to object \(y\) instead of the global variable and the usage of the property lookup relation \(\triangleright_{\text{proto}}\) to lookup property \(f\) on object \(y\). This is the true in both the VM monitor case as outlined in Definition 7 and the inlined monitor case as outlined in Table 15.

**Case** \(x = \text{new} \ f(\overline{e});:\)

VM monitor execution:
by Definition 8
\[ (\langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, x=\text{new } f(\vec{e}); \rangle ) \]
\[ \xrightarrow{\triangleright_{\text{ConstrFunc}}} \]
\[ \langle \langle \Gamma'', \Sigma'', \Lambda, \Delta, pc \rangle, \langle H'', r, x = \vec{v} \rangle \rangle \]
\[ a_s(x, \vec{v}) \]
\[ \xrightarrow{\tau_0} \]
\[ \langle \Gamma''(r_x)[x \mapsto H''(\vec{v})], \Sigma''(r_x)[x \mapsto \Sigma''(\vec{v})], \Lambda, \Delta, pc, \rangle \]
\[ \langle H''(r_x)[x \mapsto \vec{v}], r, \Box \rangle \rangle \]
where \( \tau_0 \) is the security type of \( \vec{v} \) and \( r_x \) is a reference to variable \( x \).

= \[ \langle \langle \Gamma'', \Sigma'', \Lambda, \Delta, pc \rangle, \langle H'', r, S' \rangle \rangle \]
which defines \( \Gamma'', \Sigma'', H'', S' = \Box \)

Now we show the inlined monitor (the instrumented code) execution for the same case:

\[ (H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, "\$rf = \$scope($$CS, 'f')['f']; \$rf.scope = $$CS; ...") \]
\[ sk \]
\[ \xrightarrow{\text{sk}} \]
\[ (H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, "\$rf.this = \$scope($$CS, 'x')['x'] = \{ \Sigma : \$pc.\ell, \_proto_ : \$rf.prototype\}; ...") \]
\[ sk \]
\[ \xrightarrow{\text{sk}} \]
\[ (H \uplus mst = (\Gamma(r_x)[x \mapsto (\Sigma : \$pc.\ell, ...)], \Lambda, \Delta, pc), r, "\$rf['x'] = \text{sec}\_\text{eval}(e_i)\); \$rf.\text{InvokedAsConstr} = \text{true}; ...") \]
\[ sk \]
\[ \xrightarrow{\text{sk}} \]
\[ (H \uplus mst = (\Gamma', \Lambda, \Delta, pc'), r, "x = \text{new } f(\vec{e}); \$rx = \$scope($$CS, 'x')['x']; ..." \]
\[ a_s(x, \vec{v}) \]
\[ \xrightarrow{\text{sk}} \]
\[ (H''(r_x)[x \mapsto \vec{v} \uplus mst = (\Gamma', \Lambda', \Delta, pc'), r, "\$rx = \$Lambda.\text{pop}().\ell; ...") \]
\[ sk \]
\[ \xrightarrow{\text{sk}} \]
\[ (H'' \uplus mst = (\Gamma'(r_x)[x \mapsto \$Lambda.\text{pop}().\ell], \Lambda, \Delta, pc), r, "(\text{isObj}(\$rx))?\$rx.\Sigma = \$rx.\Sigma \uplus \$pc.\ell : \$rx = \$rx \uplus \$pc.\ell; \checkmark") \]
\[ sk \]
\[ \xrightarrow{\text{sk}} \]
\[ (H'' \uplus mst = (\Gamma''(r_x)[x \mapsto \text{pc.\ell} \uplus \Lambda.\text{pop}().\ell], \Lambda, \Delta, pc), r, \checkmark") \]
\[ t \]
\[ \xrightarrow{\text{t}} \]
\[ (H'' \uplus mst = (\Gamma'', \Lambda, \Delta, pc), r, \Box) \]
\[ \xrightarrow{\text{t}} \]
\[ (H'' \uplus mst = (\Gamma'', \Lambda, \Delta, pc), r, \Box) \]
which defines \( H'', mst = (\Gamma'', \Lambda, \Delta, pc), \Box' = \Box \)

By applying rule \( \Gamma''_m, \Lambda, \Delta, pc \vdash t(\Box) \Rightarrow t(\Box) \) we obtain a valid transformation relation between \( S' \) and \( \checkmark' \) and by taking \( \Gamma''_m = \Gamma'' \uplus \Sigma''' \), we get \( (\Gamma''_m, \Sigma'''_m, \Lambda, \Delta, pc) \cong (mst''' = (\Gamma''_m, \Lambda_{in}, \Delta_{in}, pc_{in})) \), consequently we get \( (\Gamma'', \Sigma''_m, \Lambda, \Delta, pc), \langle \langle H'', r, S' \rangle \rangle \) \( \sim \) \( (H'' \uplus mst'''_m, r, \checkmark' \rangle \) hence, clause (a) of Theorem 2 holds.

If we apply the output relation \( |_{out} \) on both of \( \vec{a} \) and \( \vec{b} \) we get \( \vec{a} |_{out} \emptyset \) and \( \vec{b} |_{out} \emptyset \), trivially \( \emptyset = \emptyset \) since the two monitors did not generate any observable events, hence, clause (b) of Theorem 2 holds.

Case \( x = \text{new } y.f(\vec{e}); \mid x = \text{new } y['f'](\vec{e}); : \)
Hybrid Flow-Sensitive Security Monitor for JavaScript

Similar to the previous case with the difference of using property lookup relation $\triangleright_{\text{proto}}$ outlined in definition 2 to lookup property method $f$ on object $y$. This is the true in both the VM monitor case as outlined in Definition 9 and the inlined monitor case as outlined in Table 15.

Case $x.y = e$; VM monitor execution:

$$
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, x.y = e \rangle \rangle
$$

Next, we show the trace of the instrumented statement(s) until the first $t$-event:

$$
\langle H \uplus \text{mst}, r, "x.y = e; } $\text{scope}(\text{sec}_x'(x'))[y'] = \text{sec}_y(e); ..." \rangle
$$

By applying rule $\Gamma', \Lambda, \Delta, pc \vdash t(\square) \Rightarrow_1 \square$ we obtain a valid transformation relation between $\hat{S}'$ and $\hat{S}$ and by taking $\Gamma'_m = \Gamma' \uplus \Sigma'$ and $H' = H'$, we get $\langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle \equiv (\text{mst}' = (\Gamma'_m, \Lambda_m, \Delta_m, pc_m))$, consequently we get $\langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H', r, S' \rangle \rangle \sim \langle H' \equiv mst', r, \hat{S}' \rangle$ which define $H', mst', \hat{S}'$.

Case $x[y] = e$;

VM monitor execution:
Hybrid Flow-Sensitive Security Monitor for JavaScript

Next, we show the trace of the instrumented statement(s) until the first $t$-event:

$$
\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, x[y] = e \rangle \rangle
$$

$$
\xrightarrow{a_i(x,y,e)}
\langle \langle \Gamma(r_x)[x \mapsto [y \mapsto pc.\ell \sqcup \tau_1 \sqcup \tau_2]], \Sigma(r_x)[x \mapsto pc.\ell \sqcup \tau_1 \sqcup \tau_2 \sqcup \Sigma(r_x)(x)], \Lambda, \Delta, pc \rangle, \langle H(r_x)[x \mapsto [y \mapsto va]], r, \square \rangle \rangle
$$

where $\langle H, r, x \rangle \triangleright_{scope} r_x \sqcup \tau_1 = \text{sec lvl}(y)$ and $\tau_2$ is the security level of expression $e$.

$$=
\langle \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H', r, S' \rangle \rangle \quad \text{which defines } H', \Gamma', \Sigma', S'
$$

By applying rule $\Gamma', \Lambda, \Delta, pc \vdash t(\square) \Rightarrow t \square$ we obtain a valid transformation relation between $S'$ and $\hat{S}'$ and by taking $\Gamma'_m = \Gamma' \sqcup \Sigma'$ and $H' = H'$, we get $\langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle \cong (mst' = (\Gamma'_m, \Lambda_m, \Delta_m, pc_m))$, consequently we get $\langle \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H', r, S' \rangle \rangle \sim (H' \sqcup mst', r, \hat{S}')$ hence, clause (a) of Theorem 2 holds.

If we apply the output relation $\mid_{out}$ on both of $\bar{\alpha}$ and $\bar{\beta}$ we get $\bar{\alpha} \mid_{out} \emptyset$ and $\bar{\beta} \mid_{out} \emptyset$, trivially $\emptyset = \emptyset$ since the two monitors did not generate any observable events hence, clause (b) of Theorem 2 holds.

End of induction on the different cases. ■