Dynamic Information Flow Analysis for Featherweight JavaScript Technical Report #UCSC-SOE-11-19

Thomas H. Austin
UC Santa Cruz
taustin@ucsc.edu

Tim Disney
UC Santa Cruz
tdisney@ucsc.edu

Cormac Flanagan
UC Santa Cruz
cormac@ucsc.edu

Alan Jeffrey
Alcatel-Lucent
ajeffrey@bell-labs.com

September 22, 2011

Abstract

Although JavaScript is an important part of Web 2.0, it has historically been a major source of security holes. Code from malicious advertisers and cross-site-scripting (XSS) attacks are particularly pervasive problems. In this paper, we explore dynamic information flow to prevent the loss of confidential information from malicious JavaScript code. In particular, we extend prior dynamic information flow techniques to deal with the many complexities of JavaScript, including mutable and extensible objects and arrays, dynamic prototype chains for field and method inheritance, functions with implicit `this` arguments that are also used as methods and constructors, etc. We formally verify that our extended dynamic analysis provides termination-insensitive non-interference.

1 Introduction

JavaScript has become a pillar of Web 2.0. But while it has been used to create many powerful applications, JavaScript in the browser has also proven to be rife with security problems. The primary defense for JavaScript is the same origin policy, which prevents code from accessing documents that originate from outside the current domain [33]. However, this policy has failed to prevent attacks from malicious code injected in a victim site.

As an example, consider a banking website that stores a customer's account number in a JavaScript variable called `accountNumber`. The following JavaScript code, if injected into the bank's webpage, is sufficient to steal this confidential value:

```
var img = new Image();
img.src = "http://evil.com/"+accountNumber;
```

Setting the `src` attribute on an image element causes the browser to open a connection to the specified URL. Since the account number is included in the URL, the attacker can retrieve it from the web server logs on `evil.com`.

We propose using information flow analysis in JavaScript as a defense against malicious JavaScript code. Information flow analysis tracks sensitive information (such as `accountNumber`) as it flows though the execution of a program by associating security labels with values and preventing values marked with confidential labels from being leaked to a possible attacker. If `accountNumber` was tagged as confidential in the above example, the leak to `img.src` would be prevented.

Information flow has traditionally been analyzed statically, most commonly via specialized static type systems [27, 44, 35, 34]. However, JavaScript is a dynamically typed language, making type-based approaches difficult. Statically certifying code at runtime would lead to an undesirable performance penalty and is likely to be overly restrictive. Because of JavaScript's flexibility, offline certification is problematic; previously unseen code injected by an attacker could modify certified code at runtime, invalidating any security claims.

In prior work, we investigated dynamic information flow analysis in the simplified setting of the lambda calculus with mutable reference cells [6, 7]. However, Javascript is much more complex and involved language than the lambda calculus, with many additional features and complexities that are not addressed in this prior work. These features include mutable and extensible objects and arrays, prototype-based inheritance of fields and methods with dynamic prototype chains, functions that also double as methods and constructors, complex initial environment bindings, etc.

In this paper we address these additional complexities of JavaScript. We prove the security of our analysis on an interesting subset of JavaScript called Featherweight JavaScript (FWJS) that retains many of the interesting features of JavaScript, including its prototype-based object system, while being small enough to facilitate formal reasoning.

Much as buffer overrun vulnerabilities have been eliminated through the use of memory-safe languages, we hypothesize that information flow can provide a systemic defense against data confidentiality and integrity violations, even in an adversarial environment that contains malicious JavaScript code.

The outline of this paper is as follows: Section 2 gives an overview of information flow analysis, Section 3 introduces Featherweight JavaScript, Section 4 presents our dynamic information flow analysis and non-interference proof for Featherweight JavaScript, Section 5 gives an overview of related work in this area, and Section 6 concludes.

2 Information Flow Overview

In this section, we review the basics of information flow analysis. While our discussion focuses on protecting confidential information, the same techniques may be used to protect the integrity of information as well.

The standard guarantee of information flow analysis is *termination-insensitive non-interference*. Non-interference means that the public outputs will be unaffected by private data, or for integrity, that trusted outputs will not be affected by untrusted data. Termination-insensitive non-interference loosens this restriction to allow for a single bit of private data to be lost, but only through the termination behavior of the program.[1]

For our discussion, we will use the example of a government trying to determine the browsing habits of its citizens. Specifically, the government wishes to know if any of its citizens have visited `dissident.org`, `revolution.org`, or any number of other sites indicating disloyalty to the regime.

We assume that the government has been able to inject JavaScript code into another site, either through an XSS vulnerability or through compromised advertising code. Our example uses a function `getComputedColor(url)` that returns the color of a link to the specified URL, and a variable `visited` that denotes the color of visited links. If a visitor has visited `dissident.org`, then

```
getComputedColor("dissident.org") === visited
```

evaluates to true[2]. However, the result of `getComputedColor` in our example is confidential, and as such the government cannot directly export it to an external site.

Information flow analysis assumes that the attacker is able to inject arbitrary code into the webpage. In an extreme case, the attacker might be in control of all the JavaScript code running on the webpage (for example, when a user decides to visit `evil.gov`). As a result, it is not sufficient to analyze only *explicit*

[1] Askarov et al. [2] note that intermediary output channels can leak more than a single bit, but the attacker is reduced to a brute-force assault.

[2] A common technique for learning the user's browsing history is to look at the computed color of links. The latest versions of browsers have begun to block this technique. Nonetheless, we use this example since it is particularly illustrative of the challenges involved in information flow analysis.

flows, where information leaks through direct assignment. We must also consider the much more problematic *implicit flows*, where data is leaked through the control flow of the program.

For an example of explicit flows, consider the following snippet of code:

```
function declassify(c) {
    var x = c;
    return x;
}
declassify(getComputedColor("dissident.org"));
```

This function attempts to declassify the color of the link by reassigning it On the assignment of the color `c` to the (implicitly) public variable `x`, the leak can be prevented simply by making `x`'s security label confidential. However, with control of the source code, the government can attempt to deduce the variable without direct assignment.

The following code uses implicit flows to determine whether a user has visited the specified URL. While `v` is never assigned to directly, its value will surrender the secret that we wish to protect.

```
function wasVisited(url) {
  var v = false;
  if (getComputedColor(url) === visited)
    v = true;
  return v;
}
```

The bulk of information flow research has focused on static approaches, but as discussed in the introduction, static analysis is a poor fit for JavaScript. Recent work [46, 6, 7] has addressed the perceived weakness of dynamic information flow analysis and established the proper semantic checks to guarantee termination-insensitive non-interference. The *no-sensitive-upgrade check* [46, 6] guarantees non-interference by disallowing updates to public variables from code whose execution is conditional on sensitive data. In the `wasVisited` function, the no-sensitive-upgrade check would cause the program to terminate on the assignment to `v` to avoid leaking information about the link color.

Through termination, the government might still be able to learn whether the user had visited `dissident.org`. Critically, however, they could not glean more than one bit of information per program execution. To illustrate the importance of that restriction, consider the following example.

```
var sitesVisited = [
  wasVisited("dissident.org"),
  wasVisited("revolution.org"),
  ...
  wasVisited("downWithTheMan.org")];
return sitesVisited;
```

Normal termination of the above code would reveal that a user had not visited *any* of the listed sites. Abrupt termination due to a failed no-sensitive-upgrade check would reveal only that *some* site had been visited. While that might be useful information, the government would not be able to learn which site was fostering the most dissent. (The government could use intermediate output channels as part of the `wasVisited` function, and could therefore identify which site caused execution to terminate; however, the government would receive no information about sites later in the list of URLs).

Previous research on purely dynamic information flow analysis focused on variations of lambda calculus. We apply dynamic information flow analysis to a substantial subset of JavaScript and show how to address the complexities of a prototype-based object system in a secure manner.

3 Featherweight JavaScript Language

The full JavaScript language is quite complex and difficult to reason about, and is a difficult foundation for proving formal security guarantees. To circumvent these difficulties, we take a two-phase approach to JavaScript specification. We first present a minimal core language called LambdaScript that captures the essence of JavaScript: higher-order functions, prototype-based objects, and `eval`. LambdaScript is intentionally minimalistic, facilitating formal security proofs. We then show how Featherweight JavaScript (a substantial portion of the full JavaScript language, as shown in Figure 1) can be translated or desugared into LambdaScript.

3.1 Featherweight JavaScript Syntax

The syntax of Featherweight JavaScript is presented in Figure 1, and captures a significant subset of the JavaScript language. The language includes mutable variables, the `this` keyword, constants, object literals $\{\ \overline{\mathtt{x}_i : e_i}\ \}$, array literals $[\ \overline{e_i}\ ]$, and function definitions. It includes both direct (e.`x`) and computed (e`[`e`]`) object and array accesses and updates, as well as direct and computed method calls. FWJS captures much of the complexity surrounding functions. In addition to being called as a method, functions can also be called directly, as in $e(\overline{f_i})$, or as a constructor, as in `new` $e(\overline{f_i})$. The language includes a number of useful features, such as conditional expressions, sequential composition, and binary operators. Given its prevalence in JavaScript, we include dynamic code evaluation through *eval*. Finally, although it is not part of the JavaScript language, we include the ability to add a security label k to an expression ($\langle k \rangle e$) in order to facilitate information flow analysis.

Although Featherweight JavaScript does not include every feature of JavaScript, it nonetheless captures an extensive subset of the full language. While we could attempt to provide an operational semantics for FWJS, the language still has more complexity than we wish to deal with directly. For example, consider the steps involved in evaluating a method call `e[x](a1,a2)`.

1. The expression `e` must first be evaluated to an object.
2. Next, the variable `x` must be evaluated to determine which property to fetch.
3. Now, the property that `e[x]` refers to must be fetched from the object. If not available, the prototype chain must be searched for the desired property, since JavaScript objects include a prototype field for member inheritence.
4. Once the property has been found and the proper function has been returned, the evaluation rules must ensure that the `this` register binds correctly.
5. The parameters `a1` and `a2` must be evaluated and made available within the scope of the function. Furthermore, the implicit `arguments` array must be allocated and initialized with the specified parameters.
6. Finally, the method call may be evaluated.

While the subtleties of this evaluation are far from insurmountable, we take a simpler approach of translating or desugaring FWJS to its essential core in LambdaScript.

3.2 LambdaScript Syntax

Figure 2 lists the constructs for LambdaScript. We use the term *registers* for LambdaScript immutable registers, to distinguish them from FWJS variables, which are stored in a mutable scope object. Expressions include registers (a), lambda abstraction ($\lambda a.e$), lambda application ($e\ e$), object allocation (`new`), object access ($e[e]$), object update ($e[e] = e$), binary operators ($e \oplus e$), dynamic code evaluation ($eval(e)$), and

Figure 1: Featherweight JavaScript Syntax

$e, f, g ::=$		*Terms:*
	$\texttt{x}$	variable
	$\texttt{x} = e$	variable update
	`this`	this register
	c	constants
	`true`	boolean true
	`false`	boolean false
	$\{\ \overline{\texttt{x}_i : e_i}\ \}$	object literal
	$[\ \overline{e_i}\ ]$	array literal
	`function` $f(\ \overline{\texttt{x}_i}\ )\ \{$ `var` $\overline{\texttt{y}_i};$ `return` $e;\ \}$	function
	$e.\texttt{x}$	direct object access
	$e[e]$	computed object access
	$e.\texttt{x} = f$	direct object update
	$e[f] = g$	computed object update
	$e.\texttt{x}(\ \overline{f_i}\ )$	direct method call
	$e[f](\ \overline{g_i}\ )$	computed method call
	$e(\ \overline{f_i}\ )$	function invocation
	`new` $e(\ \overline{f_i}\ )$	constructor invocation
	`if` $(e)\ \{e\}$ `else` $\{e\}$	conditional
	$e; f$	sequential composition
	$e \oplus e$	binary operator
	$\mathit{eval}(e)$	dynamic evaluation
	$\langle k \rangle e$	add security label
$c ::=$		*Constants:*
	s	string
	`null`	null value
	`undefined`	undefined value
$\oplus ::=$	$+, ==, ...$	*Operators*
$\texttt{x}, \texttt{y}, \texttt{z}$		*Variables/Fields*
k, l, m		*Labels*
s		*Strings*

constants (c). Lambda abstractions, lambda applications, and registers, are not available to developers within FWJS. Nonetheless, these constructs are crucial, since they are the building blocks for more complex features.

Objects in LambdaScript rely on the `_proto_` field to provide inheritance. If a property is unavailable in the current object, then it may be available in the object's prototype, which is specified by `_proto_`. If `_proto_` is `undefined` (indicating that the object is at the top of the hierarchy), then the value `undefined` is returned.

3.3 Desugaring Featherweight JavaScript

LambdaScript is intentionally minimal. Nonetheless, its design allows us to formally define the semantics of

Figure 2: LambdaScript Syntax

$e, f, g ::=$	*Terms:*
a	registers
$\lambda a.\, e$	abstraction
$e\ e$	application
`new`	object allocation
$e[e]$	object selection
$e[e] = e$	object update
$e \oplus e$	binary operators
$eval(e)$	dynamic evaluation
c	constant
$\langle k \rangle e$	labeled expression
a, b	*Registers*

FWJS via a translation or desugaring into this core language. Figure 3 shows how Featherweight JavaScript constructs can be desugared in a manner that is consistent with JavaScript's semantics.

An FWJS object allocation $\{\ \overline{s_i : e_i}\ \}$ is desugared into separate operations that allocate a new object (via `new`), initialize its `_proto_` to *ObjectProto* (the default initial value of the `prototype` field of the global *Object* binding), and then initialize each field $\overline{s_i}$.

$$\{\ \overline{s_i : e_i}\ \} \stackrel{\text{def}}{=} \texttt{let}\ a = \texttt{new};\ a.\texttt{\_proto\_} = \mathit{ObjectProto};\ \overline{a[s_i] = e_i};\ a$$

Array allocations $[\ \overline{e_i}\ ]$ are desugared in a similar manner. The construct

$$\texttt{rec}\ a = \{\ \overline{s_i : e_i}\ \}\ \texttt{in}\ e$$

defines cyclic data structures in a convenient manner, where the register a can appear free in the e_i, and is extended in Figure 3 to simultaneously allocate and initialize multiple objects.

Much of the complexity in JavaScript centers around functions, which also double as constructors and methods, and which are actually objects with (hidden) `_call_` and `_construct_` properties. The desugaring of a FWJS function explicates that the `_call_` property contains a lambda abstraction that expects two (curried) arguments: a binding for `this`, and an argument array *args*. The body of the lambda abstraction then creates a new scope object `_scope_` that includes the local variables $\texttt{y}_i$ and the implicit variable `arguments`, and the replaces references to local and argument variables with accesses to the scope object and the arguments array, respectively. Consistent with JavaScript's semantics, the parameters are also available through the `arguments` variable.

A corresponding method invocation $e.\texttt{x}(\overline{f_i})$ is then desugared (via several steps, as illustrated below) into code that

1. binds a register a to the result of evaluating e;
2. extracts the function object at field "`x`";
3. extracts the `_call_` field of that function object, which is a lambda abstraction, and
4. applies that lambda abstraction to two curried arguments: the object a itself (which will be bound to `this`), and an array ($[\ \overline{f_i}\ ]$) of argument expressions.

Figure 3: Translating FWJS into LambdaScript

Note: $\overline{X_i}$ abbreviates $X_i^{i \in 0..n-1}$

$$
\begin{array}{rcl}
\texttt{let}\ a = e;\ f & \stackrel{\text{def}}{=} & (\lambda a.\, f)\ e \\
e; f & \stackrel{\text{def}}{=} & \texttt{let}\ a = e;\ f \qquad\qquad a \notin FV(f) \\
\texttt{true} & \stackrel{\text{def}}{=} & \lambda x.\lambda y.x \\
\texttt{false} & \stackrel{\text{def}}{=} & \lambda x.\lambda y.y \\
\texttt{if}\ (e_1)\ \{e_2\}\ \texttt{else}\ \{e_3\} & \stackrel{\text{def}}{=} & (e_1\ (\lambda d.e_2)\ (\lambda d.e_3))\ (\lambda x.x) \\
e.\texttt{x} & \stackrel{\text{def}}{=} & e[\text{“}\texttt{x}\text{”}] \\
e.\texttt{x} = f & \stackrel{\text{def}}{=} & e[\text{“}\texttt{x}\text{”}] = f \\
e.\texttt{x}(\ \overline{f_i}\ ) & \stackrel{\text{def}}{=} & e[\text{“}\texttt{x}\text{”}](\ \overline{f_i}\ ) \\
\texttt{x} : e & \stackrel{\text{def}}{=} & \text{“}\texttt{x}\text{”} : e \\
e[f](\overline{g_i}) & \stackrel{\text{def}}{=} & \texttt{let}\ a = e;\ (a[f].\texttt{\_call\_})(a)[\ \overline{g_i}\ ] \\
e(\ \overline{f_i}\ ) & \stackrel{\text{def}}{=} & (e.\texttt{\_call\_})(\mathit{GlobalObj})[\ \overline{f_i}\ ] \\
\texttt{new}\ e(\overline{f_i}) & \stackrel{\text{def}}{=} & (e.\texttt{\_construct\_})[\ \overline{f_i}\ ] \\
\texttt{rec}\ \overline{a_i = \{\ \overline{s_{ij} : e_{ij}}\ \}}\ \texttt{in}\ e & \stackrel{\text{def}}{=} & \texttt{let}\ \overline{a_i = \texttt{new}}\ \texttt{in}\ \overline{\overline{a_i[s_{ij}] = e_{ij}};}\ e
\end{array}
$$

$$
\begin{array}{rcl}
\{\ \overline{s_i : e_i}\ \} & \stackrel{\text{def}}{=} & \texttt{let}\ a = \texttt{new};\ a.\texttt{\_proto\_} = \mathit{ObjectProto};\ \overline{a[s_i] = e_i};\ a \\
[\ \overline{e_i}\ ] & \stackrel{\text{def}}{=} & \texttt{let}\ a = \texttt{new};\ a.\texttt{\_proto\_} = \mathit{ArrayProto};\ \overline{a[\text{“}i\text{”}] = e_i};\ a
\end{array}
$$

$$
\begin{array}{l}
\texttt{function}\ f(\ \overline{\texttt{x}_i}\ )\ \{\ \texttt{var}\ \overline{\texttt{y}_i};\ \texttt{return}\ e;\ \} \stackrel{\text{def}}{=} \\
\texttt{rec}\ a = \{ \\
\quad \texttt{prototype} : \{\texttt{constructor} : a\}, \\
\quad \texttt{\_proto\_} : \mathit{FunctionProto}, \\
\quad \texttt{\_call\_} : \lambda \texttt{this}.\, \lambda \mathit{args}. \\
\qquad \texttt{let}\ \texttt{\_scope\_} = \{\texttt{\_proto\_} : \texttt{null}, \texttt{arguments} : \mathit{args}\}; \\
\qquad e[\ \overline{\texttt{x}_i := \mathit{args}[\text{“}i\text{”}]},\ \ \overline{\texttt{y}_i := \texttt{\_scope\_}[\text{“}\texttt{y}_i\text{”}]}, \\
\qquad\quad \texttt{arguments} := \texttt{\_scope\_}[\text{“}\texttt{arguments}\text{”}]\ \ ], \\
\quad \texttt{\_construct\_} : \lambda \mathit{args}.\, \texttt{let}\ b = \{\texttt{\_proto\_} : a.\texttt{prototype}\}; \\
\qquad (a.\texttt{\_call\_})(b)(\mathit{args});\ b \\
\}\ \texttt{in}\ a
\end{array}
$$

The array expression $[\ \overline{f_i}\ ]$ in turn desugars into separate allocation and initialization operations, including for the implicit `_proto_` field:

$$
\begin{array}{rl}
& e.\texttt{x}(\ \overline{f_i}\ ) \\
= & e[\text{“}\texttt{x}\text{”}](\ \overline{f_i}\ ) \\
= & \texttt{let}\ a = e;\ (a[\text{“}\texttt{x}\text{”}].\texttt{\_call\_})(a)[\ \overline{f_i}\ ] \\
= & \texttt{let}\ a = e;\ (a[\text{“}\texttt{x}\text{”}][\text{“}\texttt{\_call\_}\text{”}])(a)[\ \overline{f_i}\ ] \\
= & \texttt{let}\ a = e;\ (a[\text{“}\texttt{x}\text{”}][\text{“}\texttt{\_call\_}\text{”}])(a) \\
& \quad (\texttt{let}\ b = \texttt{new};\ b.\texttt{\_proto\_} = \mathit{ArrayProto};\ \overline{b[\text{“}i\text{”}] = f_i};\ b)
\end{array}
$$

A constructor invocation $\texttt{new}\ e(\ \overline{f_i}\ )$ invokes the special `_construct_` property of the function e. This `_construct_` property is a lambda abstraction that allocates a new object `b`, sets b.`_proto_` to the `prototype` field of the function, and then calls the `_call_` of the function to initialize the newly allocated object. The

return value of a constructor invocation is then the newly created object.

Figure 4 shows the initial environment for FWJS programs. It includes the global object as well as initial bindings for the variables `Object`, `Array`, and `Function`, and concisely captures many detailed aspects of ECMAScript [29].

An implementation of FWJS is available online [21] that desugars FWJS into LambdaScript and evaluates the resulting LambdaScript program, allowing us to verify that our semantics is consistent with the intended behavior of JavaScript programs.

Figure 4: Initial Environment

$$
\begin{array}{ll}
\texttt{rec } \mathit{GlobalObj} & = \{\ \texttt{\_proto\_} : \mathit{ObjectProto},\\
& \quad \texttt{Object} : \mathit{Object},\\
& \quad \texttt{Function} : \mathit{Function},\\
& \quad \texttt{Array} : \mathit{Array}\\
& \}\\
\texttt{and } \mathit{Object} & = \{\ \texttt{\_proto\_} : \mathit{FunctionProto},\\
& \quad \texttt{prototype} : \mathit{ObjectProto},\\
& \quad \texttt{\_call\_} : \lambda \texttt{this}.\, \lambda \mathit{args}.\, \{\ \},\\
& \quad \texttt{\_construct\_} : \lambda \mathit{args}.\, \{\ \}\\
& \}\\
\texttt{and } \mathit{Function} & = \{\ \texttt{\_proto\_} : \mathit{FunctionProto},\\
& \quad \texttt{prototype} : \mathit{FunctionProto},\\
& \quad \texttt{\_call\_} : \lambda \texttt{this}.\, \lambda \mathit{args}.\ \texttt{function ()\{ return undefined; \}},\\
& \quad \texttt{\_construct\_} : \lambda \mathit{args}.\ \texttt{function ()\{ return undefined; \}}\\
& \}\\
\texttt{and } \mathit{Array} & = \{\ \texttt{\_proto\_} : \mathit{FunctionProto},\\
& \quad \texttt{prototype} : \mathit{ArrayProto},\\
& \quad \texttt{\_call\_} : \lambda \texttt{this}.\, \lambda \mathit{args}.\, \mathit{args}\\
& \quad \texttt{\_construct\_} : \lambda \mathit{args}.\, \mathit{args}\\
& \}\\
\texttt{and } \mathit{ObjectProto} & = \{\ \texttt{\_proto\_} : \texttt{undefined},\\
& \quad \texttt{constructor} : \mathit{GlobalObj}.\texttt{Object}\\
& \}\\
\texttt{and } \mathit{ArrayProto} & = \{\ \texttt{\_proto\_} : \mathit{ObjectProto},\\
& \quad \texttt{constructor} : \mathit{GlobalObj}.\texttt{Array}\\
& \}\\
\texttt{and } \mathit{FunctionProto} & = \{\ \texttt{\_proto\_} : \mathit{ObjectProto},\\
& \quad \texttt{prototype} : \{\ \texttt{constructor} : \mathit{FunctionProto}\ \},\\
& \quad \texttt{\_call\_} : \lambda \texttt{this}.\, \lambda \mathit{args}.\, \texttt{undefined},\\
& \quad \texttt{\_construct\_} : \lambda \mathit{args}.\, \texttt{undefined},\\
& \quad \texttt{constructor} : \mathit{GlobalObj}.\texttt{Function},\\
& \quad \texttt{call} : \texttt{function}(t, a)\ \{\ \texttt{return}\ (\texttt{this.\_call\_})(t)(a);\}\\
& \}\\
\texttt{in } \bullet &
\end{array}
$$

3.4 Basic Evaluation Rules

We formulate the rules for evaluating LambdaScript using a big-step operational semantics. Figure 5 shows the basic rules for evaluating FWJS without non-interference guarantees.

The rules defining this evaluation relation are mostly straightforward, The [N-CONST] rule evaluates a constant to itself. The [N-FUN] rule evaluates a lambda $(\lambda x.e)$ to a closure $(\lambda x.e, \theta)$ that captures the

Figure 5: No Labeling for FWJS

Evaluation Rules: $\boxed{\sigma, \theta, e \Downarrow \sigma', v}$

$$\frac{}{\sigma, \theta, c \Downarrow \sigma, c} \quad [\text{N-CONST}]$$

$$\frac{}{\sigma, \theta, (\lambda x.e) \Downarrow \sigma, (\lambda x.e, \theta)} \quad [\text{N-FUN}]$$

$$\frac{}{\sigma, \theta, x \Downarrow \sigma, \theta(x)} \quad [\text{N-VAR}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow \sigma_1, (\lambda x.e, \theta') \\ \sigma_1, \theta, e_2 \Downarrow \sigma_2, v_1 \\ \sigma_2, \theta'[x := v_1], e \Downarrow \sigma', v\end{array}}{\sigma, \theta, e_1\ e_2 \Downarrow \sigma', v} \quad [\text{N-APP}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow \sigma_1, c \\ \sigma_1, \theta, e_2 \Downarrow \sigma_2, d \\ r = c[\![\oplus]\!]d\end{array}}{\sigma, \theta, e_1 \oplus e_2 \Downarrow \sigma', r} \quad [\text{N-BINOP}]$$

$$\frac{p \notin dom(\sigma)}{\sigma, \theta, \texttt{new} \Downarrow \sigma[p := \{\}], p} \quad [\text{N-NEW}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e \Downarrow \sigma_1, s \\ e_1 = stringtoexp(s) \\ \sigma_1, \theta, e_1 \Downarrow \sigma', v\end{array}}{\sigma, \theta, \texttt{eval}(e) \Downarrow \sigma', v} \quad [\text{N-EVAL}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow \sigma_1, p \\ \sigma_1, \theta, e_2 \Downarrow \sigma', s \\ R = \sigma'(p) \\ s \in dom(R) \\ v = R(s)\end{array}}{\sigma, \theta, e_1[e_2] \Downarrow \sigma', v} \quad [\text{N-GET-DIRECT}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow \sigma_1, p \\ \sigma_1, \theta, e_2 \Downarrow \sigma_2, s \\ R = \sigma_2(p) \\ s \notin dom(R) \\ \text{"\_proto\_"} \in dom(R) \\ q = R(\text{"\_proto\_"}) \\ \sigma_2, \theta, q[s] \Downarrow \sigma', v\end{array}}{\sigma, \theta, e_1[e_2] \Downarrow \sigma', v} \quad [\text{N-GET-PARENT}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow \sigma_1, p \\ \sigma_1, \theta, e_2 \Downarrow \sigma', s \\ R = \sigma'(p) \\ s \notin dom(R) \\ \text{"\_proto\_"} \notin dom(R)\end{array}}{\sigma, \theta, e_1[e_2] \Downarrow \sigma', \texttt{undefined}} \quad [\text{N-GET-UNDEFINED}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow \sigma_1, p \\ \sigma_1, \theta, e_2 \Downarrow \sigma_2, s \\ \sigma_2, \theta, e_3 \Downarrow \sigma', v \\ R = \sigma'(p) \\ R' = R + (s : v)\end{array}}{\sigma, \theta, e_1[e_2] = e_3 \Downarrow \sigma'[p := R'], v} \quad [\text{N-SET}]$$

current substitution. The [N-VAR] rule for a variable reference a extracts the corresponding value $\theta(a)$ from the environment. The [N-APP] rule applies a closure to an argument. The [N-BINOP] rule applies binary operators, where $[\![\oplus]\!]$ has the expected meaning for the operator. For instance, `"no"+"table"` would produce `"notable"`. The [N-EVAL] rule creates a new expression from a string and then executes that expression.

State in FWJS is handled through mutable objects. The [N-NEW] rule allocates a new object in the environment and returns a pointer to it. The [N-SET] rule updates a field in a object with a new value.

One of the more complex areas of JavaScript is combining extensible objects with prototype-based inheritance. Instead of class definitions, objects contain a reference to a parent object. When getting a field from an object, its own available fields are checked first, represented by the [N-GET-DIRECT] rule. If the field is not available, then the field will be retrieved from the parent object, handled by the [N-GET-PARENT] rule. This evaluation will continue until it reaches the global object, which has no parent, in which case `undefined` is returned, as indicated by the [N-GET-UNDEFINED] rule.

4 Information Flow Analysis for FWJS

We next study information flow for Featherweight JavaScript. Because we translate Featherweight JavaScript into LambdaScript, it suffices to first study dynamic information flow for LambdaScript itself, and this analysis will directly extend to FWJS as well.

We formalize our analysis as an evaluation semantics for LambdaScript programs that tracks the security label of each value. Labels include H for high-security (confidential) data and L for low-security (public) data, with a join operator $\sqcup$. Although a more complex security lattice [16] might be useful in practice, this two-element lattice suffices to illustrate our approach.

As shown in Figure 6, a *store* is a map from pointers to objects. An *object* is a collections of fieldname strings and associated values. Since JavaScript objects are extensible, each object includes a security label k that implicitly applies to fields that are not yet defined in the object. A *value* is a pair of a raw value and a security label, and a *raw value* is either a constant, a pointer, or a closure that combines a lambda abstraction with a *substitution* or *environment* θ that provides bindings for the free registers of the lambda abstraction.

We formalize LambdaScript semantics via the judgement

$$\sigma, \theta, e \Downarrow_{pc} \sigma', v$$

which states that evaluating the expression e from the store σ and substitution θ terminates with the value v and the (possibly modified) store σ'. Here, pc captures the security level of the program counter. The evaluation rules defining this judgement are shown in Figure 6.

For clarity, these rules present a universal-labeling strategy, though we note that a *sparse-labeling strategy* [6] would provide the same guarantees with less overhead. We include an equivalent sparse variant of our rules in the appendix.

There are a number of subtleties on how labels are handled in these rules. In particular, we adopt the invariant that the label on the resulting value v is at least as secure as the program counter ($pc \sqsubseteq label(v)$). Thus, for example, the [CONST] rule evaluates a constant c to the labeled value c^{pc}. The [FUN] rule evaluates a lambda abstraction $(\lambda a.e)$ to a closure $(\lambda a.e, \theta)^{pc}$ that captures the current substitution and includes the program counter label. The [VAR] rule for a register a extracts the corresponding value $\theta(a)$ from the environment and strengthens its label to be at least pc, using the following overloading of the join operator:

$$(r^l) \sqcup k \quad \stackrel{\text{def}}{=} \quad r^{(l \sqcup k)}$$

The [APP] rule applies a lambda to a value; the label on the result will be at least as secure as the label on the closure. The [BINOP] rule applies binary operators, where $[\![\oplus]\!]$ applies to two raw values and has the expected meaning for the operator. The labels on both values are joined together and applied to the result. The [LABEL] rule for $\langle k \rangle e$ explicitly tags the result of evaluating e with the label k.

Figure 6: LambdaScript Semantics with Dynamic Information Flow

Runtime Syntax:

$$
\begin{array}{rcll}
p, q, r & \in & \mathit{Pointer} & \\
\sigma & \in & \mathit{Store} & = \mathit{Pointer} \rightarrow_p \mathit{Object} \\
\theta & \in & \mathit{Subst} & = \mathit{Register} \rightarrow_p \mathit{Value} \\
r & \in & \mathit{RawValue} & ::= c \mid p \mid (\lambda x.e, \theta) \\
v & \in & \mathit{Value} & ::= r^k \\
R & \in & \mathit{Object} & = \{\overline{s : v}\}^k
\end{array}
$$

Evaluation Rules: $\boxed{\sigma, \theta, e \Downarrow_{pc} \sigma', v}$

$$\frac{}{\sigma, \theta, c \Downarrow_{pc} \sigma, c^{pc}} \quad [\textsc{const}]$$

$$\frac{}{\sigma, \theta, (\lambda x.e) \Downarrow_{pc} \sigma, (\lambda x.e, \theta)^{pc}} \quad [\textsc{fun}]$$

$$\frac{}{\sigma, \theta, x \Downarrow_{pc} \sigma, (\theta(x) \sqcup pc)} \quad [\textsc{var}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_1 \\ \sigma_2, \theta'[x := v_1], e \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, e_1\ e_2 \Downarrow_{pc} \sigma', v} \quad [\textsc{app}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, c^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, d^l \\ r = c[\![\oplus]\!]d\end{array}}{\sigma, \theta, e_1 \oplus e_2 \Downarrow_{pc} \sigma', r^{k \sqcup l}} \quad [\textsc{binop}]$$

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', v}{\sigma, \theta, \langle k \rangle e \Downarrow_{pc} \sigma', v \sqcup k} \quad [\textsc{label}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e \Downarrow_{pc} \sigma_1, s^k \\ e_1 = \mathit{stringtoexp}(s) \\ \sigma_1, \theta, e_1 \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, \texttt{eval}(e) \Downarrow_{pc} \sigma', v} \quad [\textsc{eval}]$$

$$\frac{p \notin dom(\sigma)}{\sigma, \theta, \texttt{new} \Downarrow_{pc} \sigma[p := \{\}^{pc}], p^{pc}} \quad [\textsc{new}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, p^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, s^l \\ \sigma_2, \theta, e_3 \Downarrow_{pc} \sigma', v \\ R = \sigma'(p) \\ R' = R + (s : v \sqcup l) \\ k \sqsubseteq \mathit{label}(R) \\ l \sqsubseteq \mathit{label}(R, s)\end{array}}{\sigma, \theta, e_1[e_2] = e_3 \Downarrow_{pc} \sigma'[p := R'], v} \quad [\textsc{set}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, p^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma', s^l \\ R = \sigma'(p) \\ s \in dom(R) \\ r^m = R(s) \sqcup \mathit{label}(R)\end{array}}{\sigma, \theta, e_1[e_2] \Downarrow_{pc} \sigma', r^{m \sqcup k \sqcup l}} \quad [\textsc{get-direct}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, p^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, s^l \\ R = \sigma_2(p) \\ s \notin dom(R) \\ \texttt{"\_proto\_"} \in dom(R) \\ q^m = R(\texttt{"\_proto\_"}) \sqcup \mathit{label}(R) \\ \sigma_2, \theta, q[s] \Downarrow_{m \sqcup k \sqcup l} \sigma', v\end{array}}{\sigma, \theta, e_1[e_2] \Downarrow_{pc} \sigma', v} \quad [\textsc{get-parent}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, p^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma', s^l \\ R = \sigma'(p) \\ s \notin dom(R) \\ \texttt{"\_proto\_"} \notin dom(R) \\ m = \mathit{label}(R)\end{array}}{\sigma, \theta, e_1[e_2] \Downarrow_{pc} \sigma', \texttt{undefined}^{\texttt{m}\sqcup\texttt{k}\sqcup\texttt{l}}} \quad [\textsc{get-undefined}]$$

Derived Evaluation Rules for Select Encodings:

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\mathit{true}, \theta)^k \\ \sigma_1, \theta, e_2 \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, (\texttt{if}\ (e_1)\ \{e_2\}\ \texttt{else}\ \{e_3\}) \Downarrow_{pc} \sigma', v} \quad [\textsc{then}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\mathit{false}, \theta)^k \\ \sigma_1, \theta, e_3 \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, (\texttt{if}\ (e_1)\ \{e_2\}\ \texttt{else}\ \{e_3\}) \Downarrow_{pc} \sigma', v} \quad [\textsc{else}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \\ \sigma_1, \theta[x := v_1], e_2 \Downarrow_{pc} \sigma', v\end{array}}{\sigma, \theta, (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \Downarrow_{pc} \sigma', v} \quad [\textsc{let}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma', v\end{array}}{\sigma, \theta, (e_1; e_2) \Downarrow_{pc} \sigma', v} \quad [\textsc{seq}]$$

The [EVAL] rule uses the function *stringtoexp* to parse a string argument into a LambdaScript expression, which is then evaluated normally. This rule highlights one of the key benefits of a purely-dynamic analysis, which is the simplicity of handling `eval`. In contrast, many other security measures must outlaw the use of `eval`, despite its wide popularity [36].

State in FWJS is handled through mutable objects. The [NEW] rule allocates a new object in the store and returns a pointer to it. Both the pointer and the object use the program counter (pc) as their label.

The rules for fetching properties have the same complexity as in our basic semantics: the [GET-DIRECT] rule retrieves values from an object; the [GET-PARENT] rule searches the prototype chain from missing properties; and the [GET-UNDEFINED] rule returns `undefined` if the property is not available and there is no parent object. In all cases, the labels on e_1 and e_2, plus the labels on all accessed objects and fields, are attached to the final value.

The [SET] rule for $e_1[e_2] = e_3$ updates a field in an object with a new value. The first check ($k \sqsubseteq \mathit{label}(R)$) blocks any attempt to update a low-security object in a high-security context. This part of the check is identical to the handling of references in the more traditional no-sensitive-upgrade check.

However, an attacker could also attempt to leak data through the keys. In the following snippet of code, the public object `nums` could be used to leak the value of `secretNum` unless additional checks are put in place:

```
nums[secretNum] = true;
```

The needed check is $l \sqsubseteq \mathit{label}(R, s)$, where $\mathit{label}(R, s)$ is defined as:

$$\mathit{label}(R, s) = \begin{cases} \mathit{label}(R(s)) & \texttt{if } s \in \mathit{dom}(R) \\ \mathit{label}(R) & \texttt{otherwise} \end{cases}$$

This check will cause program execution to terminate, unless the previous value for `nums[secretNum]` is high-security. If the assignment is successful, the label l from evaluating e_2 is attached to the newly stored value, so that a field may never be made less secure.

From these core rules, it is straightforward to derive corresponding evaluation rules for FWJS, some of which are shown in Figure 6. Critically, the [THEN] and [ELSE] rules use the security label k of the test argument as the program counter label when evaluating the then or else branches, and so will get stuck if the conditional could potentially lead to an information leak via an implicit flow.

4.1 Termination-Insensitive Non-Interference

We now show that our evaluation semantics guarantees non-interference. In particular, if two program states differ only in H-labeled data, then these differences cannot propagate into L-labeled data.

To formalize this idea, we say two values are *H-equivalent* (written $v_1 \sim_H v_2$) if either:

1. $v_1 = v_2$, or
2. both v_1 and v_2 have the label at least H, or
3. $v_1 = (\lambda x.e, \theta_1)^k$ and $v_2 = (\lambda x.e, \theta_2)^k$ and $\theta_1 \sim_H \theta_2$.

Similarly, two substitutions are *H-equivalent* (written $\theta_1 \sim_H \theta_2$) if they have the same domain and

$$\forall x \in \mathit{dom}(\theta_1).\ \ \theta_1(x) \sim_H \theta_2(x)$$

We also define equivalence for objects. If $R_1 \sim_H R_2$, then $\mathit{dom}(R_1) = \mathit{dom}(R_2)$, and $\forall s \in \mathit{dom}(R_1)$ $R_1(s) \sim_H R_2(s)$ or $H \sqsubseteq \mathit{label}(R_1)$ and $H \sqsubseteq \mathit{label}(R_2)$.

Lemma 1 (Equivalence)**.** *The two $\sim_H$ relations on values and substitutions are equivalence relations.*

We define an analogous notion of *H-compatible* stores: two stores σ_1 and σ_2 are H-compatible (written $\sigma_1 \approx_H \sigma_2$) if they are H-equivalent at all common addresses, *i.e.*,

$$\sigma_1 \approx_H \sigma_2 \stackrel{\text{def}}{=} \forall a \in (dom(\sigma_1) \cap dom(\sigma_2)).\ \ \sigma_1(a) \sim_H \sigma_2(a)$$

Note that the H-compatible relation on stores is not transitive, *i.e.*, $\sigma_1 \approx_H \sigma_2$ and $\sigma_2 \approx_H \sigma_3$ does not imply $\sigma_1 \approx_H \sigma_3$, since σ_1 and σ_3 could have a common address that is not in σ_2. However, there is a limited degree of transitivity, provided that the domains are monotonically increasing. Critically, our evaluation rules will monotonically increase the domain of the store.

Lemma 2 (Limited Transitivity of Compatibility)**.**
If $\sigma_1 \approx_H \sigma_2$ *and* $\sigma_2 \approx_H \sigma_3$ *and* $dom(\sigma_1) \sqsubseteq dom(\sigma_2)$ *and* $dom(\sigma_2) \sqsubseteq dom(\sigma_3)$ *then* $\sigma_1 \approx_H \sigma_3$*.*

Lemma 3 (Evaluation Increases Store Domains)**.**
If $\sigma, \theta, e \Downarrow_{pc} \sigma', v$ *then* $dom(\sigma) \sqsubseteq dom(\sigma')$

The evaluation rules enforce a key invariant, namely that the label on the result of an evaluation always includes at least the program counter label:

Lemma 4 (Label Invariant)**.**
If $\sigma, \theta, e \Downarrow_{pc} \sigma', r^k$ *then* $pc \sqsubseteq k$*.*

Another important property of our dynamic analysis is that any value pulled from a H-labeled object will produce and H-labeled value.

Lemma 5 (Secure Objects Produces Secure Values)**.**
If $H \sqsubseteq label(R)$ *then* $\forall s.\ H \sqsubseteq label(R(s))$

As long as the program is evaluated in a secure context ($pc = H$), the evaluation rules preserve H-compatibility of the store.

Lemma 6 (Evaluation Preserves Compatibility)**.**
If $\sigma, \theta, e \Downarrow_H \sigma', v$ *then* $\sigma \approx_H \sigma'$*.*

Proof. By induction on the derivation of $\sigma, \theta, e \Downarrow_H \sigma', v$ and case analysis on the final rule in the derivation.

- [CONST], [FUN], [VAR]: $\sigma' = \sigma$.
- [APP], [BINOP], [GET-DIRECT], [GET-PARENT], [GET-UNDEFINED], [LABEL], [EVAL] By induction.
- [NEW]: Then $\sigma' = \sigma[p := \{\}^k]$ Therefore, σ and σ' agree on their common domain.
- [SET]: Then e = $(e_1[e_2] = e_3)$ from the antecedents of this rule we have:

$$\begin{array}{l} \sigma, \theta, e_1 \Downarrow_H \sigma_1, p^k \\ \sigma_1, \theta, e_2 \Downarrow_H \sigma_2, s^l \\ \sigma_2, \theta, e_3 \Downarrow_H \sigma_3, v \\ R = \sigma_3(p) \\ R' = R + (s : v \sqcup l) \\ k \sqsubseteq label(R) \end{array}$$

By induction we know that:

$$\begin{array}{l} \sigma \approx_H \sigma_1 \\ \sigma_1 \approx_H \sigma_2 \\ \sigma_2 \approx_H \sigma_3 \end{array}$$

By Lemma 2 and 3 $\sigma \approx_H \sigma_3$.

By Lemma 4 $H \sqsubseteq k$ and $H \sqsubseteq l$ and $H \sqsubseteq m$.

Since $\sigma' = \sigma_3[p := R']$, we know that $\forall p' \in dom(\sigma_3)$ where $p' \neq p$, $\sigma'(p') \sim_H \sigma_3(p')$. It therefore suffices to show that $R' \sim_H R$.

We know that $\forall s'$ where $s' \neq s, R(s') = R'(s')$. Therefore to prove that $R \sim_H R'$, it suffices to show that $R(s) \sim_H R'(s)$. Since $H \sqsubseteq label(R)$, by Lemma 5 we know that $H \sqsubseteq label(R(s))$. Because $H \sqsubseteq l$ we also know that $H \sqsubseteq label(R'(s))$, and therefore $R(s) \sim_H R'(s)$.

□

With these critical lemmas established, we now state non-interference for LambdaScript more formally.

Theorem 1. *If*

$$\begin{array}{l} \sigma_1 \approx_H \sigma_2 \\ \theta_1 \sim_H \theta_2 \\ \sigma_1, \theta_1, e \Downarrow_{pc} \sigma_1', v_1 \\ \sigma_2, \theta_2, e \Downarrow_{pc} \sigma_2', v_2 \end{array}$$

then

$$\begin{array}{l} \sigma_1' \approx_H \sigma_2' \\ v_1 \sim_H v_2 \end{array}$$

The full proof of non-interference is available in the appendix.

5 Related Work

Denning [16, 18] first brought many of the challenges with information flow analysis to light, including the complexities involved in implicit flows. Sabelfeld and Myers [40] give an overview of much of the research in this area.

Type systems for information flow analysis have been a popular approach. Some examples include Heintze and Riecke's SLam Calculus [27] and Volpano et al.'s type system [44]. Pottier and Simonet [35] introduce a more complex system for Core ML. Jif [30], one of the most production worthy information flow languages, was created using many ideas developed in Myers' JFlow [34]. Swamy et al. [15] develop a type system to handle dynamic policies.

More recent interest has arisen in dynamic approaches. Though Fenton [23] and Denning [17] both offered some hints in this direction, Zdancewic [46] first produced the evaluation rules to correctly handle mutable reference cells dynamically. The critical rule (later dubbed the *no-sensitive-upgrade* check [6]) is to forbid assignments to a public variable within a confidential context because the analysis cannot predict how the program would have behaved on different confidential input that executed a different control-flow path. Given this confusion, to avoid the potential for an information leak, the analysis halts execution.

An alternate strategy, pioneered by Le Guernic et al. [24], examines the code from branches that were not taken. This strategy increases precision, at the expense of run-time overhead. Shroff et al. [42] present yet another approach: with their purely-dynamic λ^{deps}, soundness is not guaranteed. However, over time their analysis will track and record dependencies between variables. Their analysis will not reject any valid programs, and will reject more and more insecure programs over time. Shinnar et al. [41] use a write-oracle to dynamically handle implicit flows, which provides soundness at some overhead cost. Chudnov and Naumann [13] show how to inline an information-flow monitor, with the long-term goal of applying their techniques to JavaScript.

Some work has been done regarding information flow in JavaScript. Vogt et al. [43] apply information flow analysis to JavaScript within Firefox. However, their system uses a static analysis phase at branch points, adding unnecessary overhead to the performance of the JavaScript engine. Chugh et al. [14] use

a static approach for analyzing JavaScript but have "holes" for dynamically generated code. Russo et al. study both the DOM [39] and timeout mechanisms [37].

Askarov and Sabelfeld [4] study dynamic code evaluation and declassification. Magazimius et al. [32] focus on JavaScript mashups and challenges of declassifying information. Devriese and Piessens [19] take the approach of executing the same program twice with both a high and a low security level, cleanly offering noninterference as well as protecting against the termination channel and timing channels. Askarov et al. [5] discuss defenses against timing channels.

Bandhakavi et al. [9] statically analyze Firefox plugins for information leaks, helping reviewers to identify potential issues with new plugins. Dhawan and Ganapathy [20] discuss JavaScript-based browser extensions (JSEs) and the associated risks. The authors modify Firefox to dynamically track information flow in GreaseMonkey scripts.

Hunt and Sands [28] describe a flow-sensitive type system. Hammer and Snelting [26] use program dependence graphs to analyze JVM bytecode to precisely guarantee termination-insensitive non-interference. Russo and Sabelfeld discuss the trade-offs between static and dynamic analyses; among other things, they prove that purely-dynamic analysis can only achieve a limited degree of flow-sensitivity, and therefore cannot be both sound and precise [38].

It has been argued that any real system for protecting confidentiality must provide a mechanism for declassifying data [4, 32]. Zdancewic [45] adds integrity labels to the security lattice and permits declassification only when the decision to do so is high-integrity. Askarov and Myers [3] use a similar approach, but also consider endorsement; they argue that *checked endorsements* are needed to prevent an attacker from endorsing an unauthorized declassification. Chong and Myers [12] instead propose using a framework for specifying application-specific declassification policies.

Featherweight JavaScript is one of several JavaScript subsets designed for formal analysis. Concurrent with our work, Guha et al. [25] use a similar approach of desugaring to a small core language they call λ_{JS}. Their work handles a larger subset than FWJS, and has been tested on large portions of Mozilla's JavaScript test suite. Maffeis et al. [31] describe an operational semantics for JavaScript as defined in the ECMAScript 3rd edition standard. Taking a different approach, Bohannon and Pierce [11] define a formal semantics for the entire browser called *Featherweight Firefox*.

One of the challenges in working with JavaScript lies in determining what features of JavaScript are used, and just as importantly, *how* they are be used. Richards et al. [36] do an extensive survey of JavaScript usage. They show that many difficult to analyze features, such as `eval` and `with`, are used extensively on production code. In a similar vein Jang et al. [1] survey a number of sites for information flow leaks; among the more interesting attacks, a number of sites use "heat maps" to track users' mouse movements.

6 Conclusions and Future Work

This paper presents a dynamic information flow analysis that addresses many of the complexities of JavaScript, including mutable and extensible objects and arrays, dynamic prototype chains for field and method inheritance, functions with implicit `this` arguments that are used as methods and constructors, etc. Our approach desugars Featherweight JavaScript into a small core language LambdaScript that then is the focus of our analysis; this modular approach facilitates the development of formal correctness proofs.

Future work involves extending FWJS to cover a fuller subset of JavaScript and designing a comprehensive security policy to guarantee the confidentiality of information in the browser. In ongoing work with Mozilla [22], we are exploring how to incorporate these ideas into Firefox.

References

[1] E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, editors. *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*. ACM, 2010.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] A. Askarov and A. Myers. A semantic framework for declassification and endorsement. In A. D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 64–84. Springer, 2010.

[4] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *IEEE Computer Security Foundations Symposium*, pages 43–59, Washington, DC, USA, 2009. IEEE Computer Society.

[5] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In Al-Shaer et al. [1], pages 297–307.

[6] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 113–124, New York, NY, USA, 2009. ACM.

[7] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–12. ACM, 2010.

[8] M. Backes and P. Ning, editors. *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, volume 5789 of *Lecture Notes in Computer Science*. Springer, 2009.

[9] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium'10*, pages 339–354, 2010.

[10] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *IEEE Computer Security Foundations Workshop*, pages 253–267. IEEE Computer Society, 2002.

[11] A. Bohannon and B. C. Pierce. Featherweight firefox: formalizing the core of a web browser. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[12] S. Chong and A. C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM.

[13] A. Chudnov and D. A. Naumann. Information flow monitor inlining. *Computer Security Foundations Symposium, IEEE*, 0:200–214, 2010.

[14] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 50–62, New York, NY, USA, 2009. ACM.

[15] B. J. Corcoran, N. Swamy, and M. W. Hicks. Cross-tier, label-based security enforcement for web applications. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *SIGMOD Conference*, pages 269–282. ACM, 2009.

[16] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[17] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[18] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[19] D. Devriese and F. Piessens. Noninterference through secure multi-execution. *Security and Privacy, IEEE Symposium on*, 0:109–124, 2010.

[20] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *ACSAC*, pages 382–391. IEEE Computer Society, 2009.

[21] T. Disney, T. H. Austin, and C. Flanagan. Featherweight javascript implementation. `http://slang.soe.ucsc.edu/fwjs-implementation.zip`, 2010.

[22] B. Eich. Mozilla FlowSafe: Information flow security for the browser. https://wiki.mozilla.org/FlowSafe, accessed October 2009.

[23] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[24] G. L. Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In M. Okada and I. Satoh, editors, *ASIAN*, volume 4435 of *Lecture Notes in Computer Science*, pages 75–89. Springer, 2006.

[25] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. Technical Report CS-09-10, Brown University, 2009.

[26] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 2009.

[27] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[28] S. Hunt and D. Sands. On flow-sensitive security types. In J. G. Morrisett and S. L. P. Jones, editors, *POPL*, pages 79–90. ACM, 2006.

[29] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, fifth edition, December 2009.

[30] Jif homepage. http://www.cs.cornell.edu/jif/, accessed October 2010.

[31] S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for javascript. In *Proc. of APLAS*, volume 8, pages 307–325. Springer, 2008.

[32] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proceedings of the ACM Symposium on Information Computer and Communications Security*, 2010.

[33] Same origin policy for JavaScript. `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`, accessed January 2010.

[34] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[35] F. Pottier and V. Simonet. Information flow inference for ML. *Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.

[36] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In B. G. Zorn and A. Aiken, editors, *PLDI*, pages 1–12. ACM, 2010.

[37] A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *IEEE Computer Security Foundations Symposium*, 2009.

[38] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2010.

[39] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In Backes and Ning [8], pages 86–103.

[40] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, Jan 2003.

[41] A. Shinnar, M. Pistoia, and A. Banerjee. A language for information flow: dynamic tracking in multiple interdependent dimensions. In S. Chong and D. A. Naumann, editors, *PLAS*, pages 125–131. ACM, 2009.

[42] P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *CSF*, pages 203–217. IEEE Computer Society, 2007.

[43] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*. The Internet Society, 2007.

[44] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[45] S. Zdancewic. A type system for robust declassification, 2003.

[46] S. A. Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.

A Sparse-Labeling Variant of FWJS

A sparse-labeling strategy can be used to reduce the overhead of our information flow controls [6]. In this section, we modify our evaluation rules to follow a sparse strategy.

The rules in Figure 7 are mostly the same as the universal labeling rules. However, whenever possible, the program counter is used in place of storing labels on values. The [*-SLOW] rules are used only when values include a label.

B Non-Interference Proof

In this section, we prove termination-insensitive non-interference. First, we note some key lemmas:

Lemma 7 (Labeling Equivalence)**.**
If $v_1 \sim_H v_2$ then $v_1 \sqcup k \sim_H v_2 \sqcup k$.

Lemma 8 (Evaluation of Get With Constants)**.**
If $\sigma, \theta, p[s] \Downarrow_{pc} \sigma', v$ then $\sigma' = \sigma$, since evaluating p and s will not modify the store in any of the get rules.

Now we restate Theorem 1 for the reader's convenience.

If

$$\begin{array}{l} \sigma_1 \approx_H \sigma_2 \\ \theta_1 \sim_H \theta_2 \\ \sigma_1, \theta_1, e \Downarrow_{pc} \sigma_1', v_1 \\ \sigma_2, \theta_2, e \Downarrow_{pc} \sigma_2', v_2 \end{array}$$

then

$$\begin{array}{l} \sigma_1' \approx_H \sigma_2' \\ v_1 \sim_H v_2 \end{array}$$

Figure 7: Sparse Labeling Semantics for FWJS

Big-Step Evaluation Rules: $\boxed{\sigma, \theta, e \downarrow_{pc} \sigma', v}$

$$\frac{}{\sigma, \theta, c \downarrow_{pc} \sigma, c} \quad [\text{S-CONST}]$$

$$\frac{}{\sigma, \theta, x \downarrow_{pc} \sigma, \theta(x)} \quad [\text{S-VAR}]$$

$$\frac{}{\sigma, \theta, (\lambda x.e) \downarrow_{pc} \sigma, (\lambda x.e, \theta)} \quad [\text{S-FUN}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, (\lambda x.e, \theta') \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \downarrow_{pc} \sigma', v\end{array}}{\sigma, \theta, (e_1\ e_2) \downarrow_{pc} \sigma', v} \quad [\text{S-APP}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \downarrow_{pc \sqcup k} \sigma', v\end{array}}{\sigma, \theta, (e_1\ e_2) \downarrow_{pc} \sigma', \langle k \rangle^{pc}\, v} \quad [\text{S-APP-SLOW}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, c \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, d \\ r = c[\![\oplus]\!]d\end{array}}{\sigma, \theta, e_1 \oplus e_2 \downarrow_{pc} \sigma', r} \quad [\text{S-BINOP}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, c^k \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, d^l \\ r = c[\![\oplus]\!]d\end{array}}{\sigma, \theta, e_1 \oplus e_2 \downarrow_{pc} \sigma', \langle k \sqcup l \rangle^{pc}\, r} \quad [\text{S-BINOP-SLOW}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e \downarrow_{pc} \sigma_1, s \\ e_1 = \mathit{stringtoexp}(s) \\ \sigma_1, \theta, e_1 \downarrow_{pc} \sigma', v\end{array}}{\sigma, \theta, \texttt{eval}(e) \downarrow_{pc} \sigma', v} \quad [\text{S-EVAL}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e \downarrow_{pc} \sigma_1, s^k \\ e_1 = \mathit{stringtoexp}(s) \\ \sigma_1, \theta, e_1 \downarrow_{pc \sqcup k} \sigma', v\end{array}}{\sigma, \theta, \texttt{eval}(e) \downarrow_{pc} \sigma', v} \quad [\text{S-EVAL-SLOW}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, p \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, s \\ \sigma_2, \theta, e_3 \downarrow_{pc} \sigma', v \\ R = \sigma'(p) \\ R' = R + (s : \langle pc \rangle^{label(p)}\, v) \\ pc \sqsubseteq \mathit{label}(R, s)\end{array}}{\sigma, \theta, e_1[e_2] = e_3 \downarrow_{pc} \sigma'[p := R'], v} \quad [\text{S-SET}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, p^k \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, s^l \\ \sigma_2, \theta, e_3 \downarrow_{pc} \sigma', v \\ R = \sigma'(p) \\ v' = \langle k \sqcup l \rangle^{label(R,s)}\, v \\ R' = R + (s : v') \\ (pc \sqcup k) \sqsubseteq \mathit{label}(p) \\ l \sqsubseteq \mathit{label}(R, s)\end{array}}{\sigma, \theta, e_1[e_2] = e_3 \downarrow_{pc} \sigma'[p := R'], v} \quad [\text{S-SET-SLOW}]$$

$$\frac{}{\sigma, \theta, \langle k \rangle e \downarrow_{pc} \sigma', \langle k \rangle^{pc}\, v} \quad [\text{S-LABEL}]$$

$$\frac{\begin{array}{c}p \notin dom(\sigma) \\ \mathit{label}(p) = pc\end{array}}{\sigma, \theta, \texttt{new} \downarrow_{pc} \sigma[p := \{\}^{pc}], p} \quad [\text{S-NEW}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, p \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma', s \\ R = \sigma'(p) \\ s \in dom(R) \\ r^m = R(s) \sqcup \mathit{label}(p)\end{array}}{\sigma, \theta, e_1[e_2] \downarrow_{pc} \sigma', \langle m \rangle^{pc}\, r} \quad [\text{S-GET-DIRECT}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, p^k \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma', s^l \\ R = \sigma'(p) \\ s \in dom(R) \\ r^m = R(s) \sqcup \mathit{label}(p)\end{array}}{\sigma, \theta, e_1[e_2] \downarrow_{pc} \sigma', \langle m \sqcup k \sqcup l \rangle^{pc}\, r} \quad [\text{S-GET-DIRECT-SLOW}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, p \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, s \\ R = \sigma_2(p) \\ s \notin dom(R) \\ \texttt{"\_proto\_"} \in dom(R) \\ q^m = R(\texttt{"\_proto\_"}) \sqcup pc \\ \sigma_2, \theta, q[s] \downarrow_m \sigma', v\end{array}}{\sigma, \theta, e_1[e_2] \downarrow_{pc} \sigma', \langle m \rangle^{pc}\, v} \quad [\text{S-GET-PARENT}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, p^k \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, s^l \\ R = \sigma_2(p) \\ s \notin dom(R) \\ \texttt{"\_proto\_"} \in dom(R) \\ q^m = R(\texttt{"\_proto\_"}) \sqcup pc \sqcup k \sqcup l \\ \sigma_2, \theta, q[s] \downarrow_m \sigma', v\end{array}}{\sigma, \theta, e_1[e_2] \downarrow_{pc} \sigma', \langle m \rangle^{pc}\, v} \quad [\text{S-GET-PARENT-SLOW}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, p \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma', s \\ R = \sigma'(p) \\ s \notin dom(R) \\ \texttt{"\_proto\_"} \notin dom(R) \\ m = \mathit{label}(p)\end{array}}{\sigma, \theta, e_1[e_2] \downarrow_{pc} \sigma', \langle m \rangle^{pc}\, \texttt{undefined}} \quad [\text{S-GET-UNDEFINED}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, p^k \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma', s^l \\ R = \sigma'(p) \\ s \notin dom(R) \\ \texttt{"\_proto\_"} \notin dom(R) \\ m = \mathit{label}(p)\end{array}}{\sigma, \theta, e_1[e_2] \downarrow_{pc} \sigma', \langle m \sqcup k \sqcup l \rangle^{pc}\, \texttt{undefined}} \quad [\text{S-GET-UNDEFINED-SLOW}]$$

Figure 8: Sparse Labeling for Encodings

$$\frac{\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, v_1 \quad \sigma_1, \theta[x := v_1], e_2 \downarrow_{pc} \sigma', v}{\sigma, \theta, (\texttt{let } x = e_1 \texttt{ in } e_2) \downarrow_{pc} \sigma', v} \text{ [S-LET]}$$

$$\frac{\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, \mathit{true} \quad \sigma_1, \theta, e_2 \downarrow_{pc} \sigma', v}{\sigma, \theta, (\texttt{if } (e_1) \ \{e_2\} \texttt{ else } \{e_3\}) \downarrow_{pc} \sigma', v} \text{ [S-THEN]}$$

$$\frac{\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, v_1 \quad \sigma_1, \theta, e_2 \downarrow_{pc} \sigma', v}{\sigma, \theta, (e_1; e_2) \downarrow_{pc} \sigma', v} \text{ [S-SEQ]}$$

$$\frac{\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, \mathit{true}^k \quad \sigma_1, \theta, e_2 \downarrow_{k} \sigma', v}{\sigma, \theta, (\texttt{if } (e_1) \ \{e_2\} \texttt{ else } \{e_3\}) \downarrow_{pc} \sigma', v} \text{ [S-THEN-SLOW]}$$

Proof. By induction on the derivation $\sigma_1, \theta_1, e \Downarrow_{pc} \sigma_1', v_1$ and case analysis on the last rule used in that derivation.

- [CONST]: Then $e = c$ and $\sigma_1' = \sigma_1 \approx_H \sigma_2 = \sigma_2'$ and $v_1 = v_2 = c$.
- [FUN]: Then $e = \lambda x.e'$ and $\sigma_1' = \sigma_1 \approx_H \sigma_2 = \sigma_2'$ and $v_1 = (\lambda x.e', \theta_1) \sim_H (\lambda x.e', \theta_2) = v_2$.
- [VAR]: Then $e = x$ and $\sigma_1' = \sigma_1 \approx_H \sigma_2 = \sigma_2'$ and $v_1 = \theta_1(x) \sim_H \theta_2(x) = v_2$.
- [APP]: In this case, $e = (e_a \ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

  $$\begin{array}{l} \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', (\lambda x.e_i, \theta_i')^{k_i} \\ \sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', v_i' \\ \sigma_i''', \theta_i'[x := v_i'], e_i \Downarrow_{k_i} \sigma_i', v_i \end{array}$$

  By induction:

  $$\begin{array}{l} \sigma_1'' \approx_H \sigma_2'' \\ \sigma_1''' \approx_H \sigma_2''' \\ (\lambda x.e_1, \theta_1')^{k_1} \sim_H (\lambda x.e_2, \theta_2')^{k_2} \\ v_1' \sim_H v_2' \end{array}$$

  - If k_1 and k_2 are both at least H then $v_1 \sim_H v_2$, since they both have labels at least H.
    By Lemma 6, $\sigma_1' \approx_H \sigma_1''' \approx_H \sigma_2''' \approx_H \sigma_2'$, and we need to conclude that $\sigma_1' \approx_H \sigma_2'$.
    We know that $\mathit{dom}(\sigma_i') \supseteq \mathit{dom}(\sigma_i''')$, since execution only allocates additional reference cells. Without loss of generality, we assume that the two executions allocate reference cells from disjoint parts of the address space,[3] *i.e.*:

    $$\begin{array}{l} (\mathit{dom}(\sigma_1') \setminus \mathit{dom}(\sigma_1''')) \cap \mathit{dom}(\sigma_2') = \emptyset \\ (\mathit{dom}(\sigma_2') \setminus \mathit{dom}(\sigma_2''')) \cap \mathit{dom}(\sigma_1') = \emptyset \end{array}$$

    Under this assumption, the only common addresses in σ_1' and σ_2' are also the common addresses in σ_1''' and σ_2''', and hence we have that $\sigma_1' \approx_H \sigma_2'$.
  - If k_1 and k_2 are *not* both at least H, then $\theta_1' \sim_H \theta_2'$ and $e_1 = e_2$ and $k_1 = k_2$. By induction, $\sigma_1' \approx_H \sigma_2'$ and $v_1'' \sim_H v_2''$, and hence $v_1' \sim_H v_2'$.

[3] We refer the interested reader to [10] for an alternative proof argument that does use of this assumption, but which involves a more complicated compatibility relation on stores.

- [BINOP]: In this case, $e = (e_a \oplus e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', c_i^{k_i}$$
$$\sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i', d_i^{l_i}$$
$$r_i = c_i [\![\oplus]\!] d_i$$

  By induction:

$$\sigma_1'' \approx_H \sigma_2''$$
$$\sigma_1' \approx_H \sigma_2'$$
$$c_1^{k_1} \sim_H c_2^{k_2}$$
$$d_1^{l_1} \sim_H d_2^{l_2}$$

  - If either $k_1 = k_2 = H$ or $l_1 = l_2 = H$, then both v_1 and v_2 will be H-labeled, and so $v_1 \sim_H v_2$.
  - Otherwise, it must be the case that $c_1 = c_2$, $k_1 = k_2$, $d_1 = d_2$, and $l_1 = l_2$. Since the raw values are identical, we know that $r_1 = r_2$. Therefore $v_1 = v_2$.

- [NEW]: In this case, $e =$ `new`. Without loss of generality, we assume that both evaluations allocate the same pointer $p \notin dom(\sigma_1) \cup dom(\sigma_2)$, and so $p^{pc} = v_1 = v_2$. Therefore: $\sigma_1' = \sigma_1[p := \{\}^{pc}]$ and $\sigma_2' = \sigma_2[p := \{\}^{pc}]$. Since $\sigma_1 \approx_H \sigma_2$, we know that $\forall p' \in dom(\sigma_1') \cap dom(\sigma_2')$ where $p' \neq p$, $\sigma_1'(p') \sim_H \sigma_2'(p')$. Finally, since $\sigma_1'(p) = \sigma_2'(p)$, we know that $\sigma_1' \approx_H \sigma_2'$.

- [GET-DIRECT]: In this case, $e = (e_a[e_b])$ and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', p_i^{k_i}$$
$$\sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', s_i^{l_i}$$
$$R_i = \sigma_i'''(p_i)$$

  Also by the antecedents:

$$s_1 \in dom(R_1)$$
$$r_1^{m_1} = R_1(s_1) \sqcup label(R_1)$$
$$v_1 = r_1^{m_1 \sqcup k_1 \sqcup l_1}$$
$$\sigma_1' = \sigma_1'''$$

  By induction:

$$\sigma_1'' \approx_H \sigma_2''$$
$$\sigma_1''' \approx_H \sigma_2'''$$
$$p_1^{k_1} \sim_H p_2^{k_2}$$
$$s_1^{l_1} \sim_H s_2^{l_2}$$

  - If $s_2 \in dom(R_2)$ then $r_2^{m_2} = R_2(s_2) \sqcup label(R_2)$. We also know that $\sigma_2''' = \sigma_2' \approx_H \sigma_1'$. Therefore it suffices to prove that $v_1 \sim_H v_2$. We prove this by contradiction. Assume $v_1 \not\sim_H v_2$ then $v_1 \neq v_2$ and either $H \not\sqsubseteq v_1$ or $H \not\sqsubseteq v_2$. Then either $H \not\sqsubseteq m_1 \sqcup k_1 \sqcup l_1$ or $H \not\sqsubseteq m_2 \sqcup k_2 \sqcup l_2$, where $m_2 = label(R_2) \sqcup label(R_2(s_2))$. This can only be true if $s_1^{l_1} = s_2^{l_2}$ and $p_1^{k_1} = p_2^{k_2}$. Because $p_1 = p_2$ and $\sigma_1''' \approx_H \sigma_2'''$, we know that $R_1 \sim_H R_2$. Since $R_1 \sim_H R_2$ and $s_1 = s_2$, we know that $m_1 = m_2$. Therefore, it must be the case that $H \not\sqsubseteq v_1$ and $H \not\sqsubseteq v_2$. But since $R_1 \sim_H R_2$, $r_1 = r_2$, which means $v_1 = v_2$. This is a contradiction. Thus we know that $v_1 \sim_H v_2$.
  - If $s_2 \notin dom(R_2)$ and "*_proto_*" $\notin dom(R_2)$ then $\sigma_2''' = \sigma_2' \approx_H \sigma_1'$. Therefore it suffices to prove that $v_1 \sim_H v_2$. We prove this by contradiction. Assume $v_1 \not\sim_H v_2$ then $v_1 \neq v_2$ and either $H \not\sqsubseteq v_1$ or $H \not\sqsubseteq v_2$. Then either $H \not\sqsubseteq m_1 \sqcup k_1 \sqcup l_1$ or $H \not\sqsubseteq m_2 \sqcup k_2 \sqcup l_2$, where $m_2 = label(R_2)$. This can only be true if $s_1^{l_1} = s_2^{l_2}$ and $p_1^{k_1} = p_2^{k_2}$. Because $p_1 = p_2$ and $\sigma_1''' \approx_H \sigma_2'''$, we know that $R_1 \sim_H R_2$. Since $R_1 \sim_H R_2$ and $H \not\sqsubseteq m_1$ or $H \not\sqsubseteq m_2$, $H \not\sqsubseteq label(R_1)$ and $H \not\sqsubseteq label(R_2)$. Then $s_1 \in dom(R_2)$, and since $s_1 = s_2$, $s_2 \in dom(R_2)$. This is a contradiction. Thus we know that $v_1 \sim_H v_2$.

– Otherwise, $s_2 \notin dom(R_2)$ and "$\_proto\_$" $\in dom(R_2)$ and so:

$$q_2^{m_2} = R_2[\text{"}\_proto\_\text{"}] \sqcup label(R_2)$$
$$\sigma_2''', \theta_2, q[s] \Downarrow_{m_2 \sqcup k_2 \sqcup l_2} \sigma_2', v_2$$

By Lemma 8, we know that $\sigma_2' = \sigma_2'''$, and therefore $\sigma_1' \sim_H \sigma_2'$. It suffices to show that $v_1 \sim_H v_2$. We prove this by contradiction. Assume $v_1 \not\sim_H v_2$. Then $v_1 \neq v_2$ and either $H \not\sqsubseteq v_1$ or $H \not\sqsubseteq v_2$. Therefore $H \not\sqsubseteq l_1 \sqcup k_1 \sqcup m_1$ or $H \not\sqsubseteq l_2 \sqcup k_2 \sqcup m_2$. This can only be true if $s_1^{l_1} = s_2^{l_2}$ and $p_1^{k_1} = p_2^{k_2}$. Because $p_1 = p_2$ and $\sigma_1''' \approx_H \sigma_2'''$, we know that $R_1 \sim_H R_2$. Since $R_1 \sim_H R_2$ and either $H \not\sqsubseteq m_1$ or $H \not\sqsubseteq m_2$, it must be the case that $H \not\sqsubseteq label(R_1)$ and $H \not\sqsubseteq label(R_2)$. But then $s_1 \in dom(R_2)$. Since $s_1 = s_2$, $s_2 \in dom(R_2)$, which is a contradiction. Thus, we know that $v_1 \sim_H v_2$.

- [GET-UNDEFINED]: In this case, $e = (e_a[e_b])$ and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', p_i^{k_i}$$
$$\sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', s_i^{l_i}$$
$$R_i = \sigma_i'''(p_i)$$

Also by the antecedents:

$$s_1 \notin dom(R_1)$$
$$\text{"}\_proto\_\text{"} \in dom(R_1)$$
$$m_1 = label(R_1)$$
$$v_1 = \texttt{undefined}$$
$$\sigma_1' = \sigma_1'''$$

By induction:

$$\sigma_1'' \approx_H \sigma_2''$$
$$\sigma_1''' \approx_H \sigma_2'''$$
$$p_1^{k_1} \sim_H p_2^{k_2}$$
$$s_1^{l_1} \sim_H s_2^{l_2}$$

– If $s_2 \in dom(R_2)$ the proof holds by reciprical argument from the first subcase of [GET-DIRECT].

– If $s_2 \notin dom(R_2)$ and "$\_proto\_$" $\notin dom(R_2)$ then $\sigma_2''' = \sigma_2' \approx_H \sigma_1'$. Therefore it suffices to prove that $v_1 \sim_H v_2$. We prove this by contradiction. Assume $v_1 \not\sim_H v_2$ then $v_1 \neq v_2$ and either $H \not\sqsubseteq v_1$ or $H \not\sqsubseteq v_2$. Then either $H \not\sqsubseteq m_1 \sqcup k_1 \sqcup l_1$ or $H \not\sqsubseteq m_2 \sqcup k_2 \sqcup l_2$, where $m_2 = label(R_2)$. This can only be true if $s_1^{l_1} = s_2^{l_2}$ and $p_1^{k_1} = p_2^{k_2}$. Because $p_1 = p_2$ and $\sigma_1''' \approx_H \sigma_2'''$, we know that $R_1 \sim_H R_2$. Since $R_1 \sim_H R_2$, we know that $m_1 = m_2$. Therefore, it must be the case that $H \not\sqsubseteq v_1$ and $H \not\sqsubseteq v_2$, and since $R_1 \sim_H R_2$, $s_1 \notin dom(R_2)$. But since $s_1 = s_2$ this means that $s_2 \notin dom(R_2)$, and therefore $v_2 = \texttt{undefined}^{m_2 \sqcup k_2 \sqcup l_2}$. However this means $v_1 = v_2$, which is a contradiction. Thus we know that $v_1 \sim_H v_2$.

– Otherwise, $s_2 \notin dom(R_2)$ and "$\_proto\_$" $\in dom(R_2)$ and so:

$$q_2^{m_2} = R_2[\text{"}\_proto\_\text{"}] \sqcup label(R_2)$$
$$\sigma_2''', \theta_2, q[s] \Downarrow_{m_2 \sqcup k_2 \sqcup l_2} \sigma_2', v_2$$

By Lemma 8, we know that $\sigma_2' = \sigma_2'''$, and therefore $\sigma_1' \sim_H \sigma_2'$. It suffices to show that $v_1 \sim_H v_2$. We prove this by contradiction. Assume $v_1 \not\sim_H v_2$. Then $v_1 \neq v_2$ and either $H \not\sqsubseteq v_1$ or $H \not\sqsubseteq v_2$. Therefore $H \not\sqsubseteq l_1 \sqcup k_1 \sqcup m_1$ or $H \not\sqsubseteq l_2 \sqcup k_2 \sqcup m_2$. This can only be true if $s_1^{l_1} = s_2^{l_2}$ and $p_1^{k_1} = p_2^{k_2}$. Because $p_1 = p_2$ and $\sigma_1''' \approx_H \sigma_2'''$, we know that $R_1 \sim_H R_2$. Since $R_1 \sim_H R_2$ and either $H \not\sqsubseteq m_1$ or $H \not\sqsubseteq m_2$, it must be the case that $H \not\sqsubseteq label(R_1)$ and $H \not\sqsubseteq label(R_2)$. But then since $R_1 \sim_H R_2$, "$\_proto\_ \notin dom(R_2)$ which is a contradiction. Thus, we know that $v_1 \sim_H v_2$.

- [GET-PARENT]: In this case, $e = (e_a[e_b])$ and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{l}\sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', p_i^{k_i} \\ \sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', s_i^{l_i} \\ R_i = \sigma_i'''(p_i)\end{array}$$

Also by the antecedents:

$$\begin{array}{l}s_1 \notin dom(R_1) \\ \text{"}\_proto\_\text{"} \in dom(R_1) \\ q_1^{m_1} = R_1(\text{"}\_proto\_\text{"}) \sqcup label(R_1) \\ \sigma_1''', \theta_1, q_1[s_1] \Downarrow_{m_1 \sqcup k_1 \sqcup l_1} \sigma_1', v_1\end{array}$$

By induction:

$$\begin{array}{l}\sigma_1'' \approx_H \sigma_2'' \\ \sigma_1''' \approx_H \sigma_2''' \\ p_1^{k_1} \sim_H p_2^{k_2} \\ s_1^{l_1} \sim_H s_2^{l_2}\end{array}$$

By Lemma 8 $\sigma_1' = \sigma_1'''$.

  - If $s_2 \in dom(R_2)$ then this holds by a reciprical argument of the third subcase of [GET-DIRECT].
  - If $s_2 \notin dom(R_2)$ and "$\_proto\_$" $\notin dom(R_2)$, then this holds by a reciprical argument of the third subcase of [GET-UNDEFINED].
  - Otherwise $s_2 \notin dom(R_2)$ and "$\_proto\_$" $\in dom(R_2)$. Therefore:

$$\begin{array}{l}q_2^{m_2} = R_2(\text{"}\_proto\_\text{"}) \sqcup label(R_2) \\ \sigma_2''', \theta_2, q_2[s_2] \Downarrow_{m_2 \sqcup k_2 \sqcup l_2} \sigma_2', v_2\end{array}$$

    By Lemma 8, $\sigma_1' = \sigma_1''' \approx_H \sigma_2''' = \sigma_2'$. Therefore, it suffices to prove that $v_1 \sim_H v_2$.
    * If $H \sqsubseteq m_1 \sqcup k_1 \sqcup l_1$ or $H \sqsubseteq m_2 \sqcup k_2 \sqcup l_2$, then $H \sqsubseteq m_1 \sqcup k_1 \sqcup l_1$ and $H \sqsubseteq m_2 \sqcup k_2 \sqcup l_2$. By Lemma 6, $H \sqsubseteq v_1$ and $H \sqsubseteq v_2$.
    * Otherwise $p_1^{k_1} = p_2^{k_2}$ and $s_1^{l_1} = s_2^{l_2}$. Since $p_1 = p_2$ and $\sigma_1''' \approx_H \sigma_2'''$, $R_1 \sim_H R_2$. Because $R_1 \sim_H R_2$ and $H \not\sqsubseteq m_1$ and $H \not\sqsubseteq m_2$, $q_1^{m_1} = q_2^{m_2}$. Then $v_1 = v_2$ by induction.

- [LABEL]: In this case, $e = \langle k \rangle e'$, and from the antecedents of this rule, we have that for $i \in 1, 2$: $\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma_i', v_i'$. By induction, $\sigma_1' \approx_H \sigma_2'$ and $v_1' \sim_H v_2'$. Since $v_1 = v_1' \sqcup k$ and $v_2 = v_2' \sqcup k$, by Lemma 7 we know that $v_1 \sim_H v_2$.

- [SET]: In this case, $e = (e_a[e_b] = e_c)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{l}\sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', p_i^{k_i} \\ \sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', s_i^{l_i} \\ \sigma_i''', \theta_i, e_c \Downarrow_{pc} \sigma_i'''', v_i \\ R_i = \sigma_i''''(p_i) \\ R_i' = R_i + (s_i : v_i \sqcup l_i) \\ k_i \sqsubseteq label(R_i) \\ \sigma_i' = \sigma_i''''[p_i := R_i']\end{array}$$

By induction:

$$\begin{array}{l}\sigma_1'' \approx_H \sigma_2'' \\ \sigma_1''' \approx_H \sigma_2''' \\ \sigma_1'''' \approx_H \sigma_2'''' \\ p_1^{k_1} \sim_H p_2^{k_2} \\ s_1^{l_1} \sim_H s_2^{l_2} \\ v_1 \sim_H v_2\end{array}$$

- If $H \sqsubseteq k_1$ and $H \sqsubseteq k_2$, then it must be the case that $H \sqsubseteq \mathit{label}(\sigma_1''''(p_1))$ and $H \sqsubseteq \mathit{label}(\sigma_2''''(p_2))$.Since $\sigma_1'''' \approx_H \sigma_2''''$, we know that $\forall p' dom(\sigma_1') \cap dom(\sigma_2')$ where $p_1 \neq p' \neq p_2, \sigma_1'(p') \sim_H \sigma_2'(p')$.
  - If $p_1 \notin dom(\sigma_2'''')$ and $p_2 \notin dom(\sigma_1'''')$, then $\sigma_1' \approx_H \sigma_2'$.
  - If $p_1 \in dom(\sigma_2'''')$ and $p_2 \in dom(\sigma_1'''')$, then since $\sigma_1'''' \approx_H \sigma_2''''$ we know that $H \sqsubseteq \mathit{label}(\sigma_1''''(p_2))$ and $H \sqsubseteq \mathit{label}(\sigma_2''''(p_1))$. Therefore, $\sigma_1' \approx_H \sigma_2'$.
  - If $p_1 \in dom(\sigma_2'''')$ and $p_2 \notin dom(\sigma_1'''')$, then since $\sigma_1'''' \approx_H \sigma_2''''$ we know that $H \sqsubseteq \mathit{label}(\sigma_2''''(p_1))$. Also, since $p_2 \notin dom(\sigma_1')$, we know that $\sigma_1' \approx_H \sigma_2'$.
  - If $p_1 \notin dom(\sigma_2'''')$ and $p_2 \in dom(\sigma_1'''')$, then this holds with a similar argument to the previous sub-case.
- Otherwise, $p_1^{k_1} = p_2^{k_2}$. In this case, it suffices to show that $R_1' \sim_H R_2'$. Since $R_1 \sim_H R_2$, we know that $\forall s'$ where $s_1 \neq s' \neq s_2$ it must be the case that $R_1'(s') \sim_H R_2'(s')$. Therefore, we only need to prove that $R_1'(s_1) \sim_H R_2'(s_1)$ and $R_1'(s_2) \sim_H R_2'(s_2)$.
  - If $H \sqsubseteq l_1$ and $H \sqsubseteq l_2$, then:
    - if $s_1 \in dom(R_1)$ and $s_2 \in dom(R_2)$, then $H \sqsubseteq \mathit{label}(R_1(s_1))$ and $H \sqsubseteq \mathit{label}(R_2(s_2))$.Since $R_1 \sim_H R_2$, it must be the case that $H \sqsubseteq \mathit{label}(R_1'(s_2))$ and $H \sqsubseteq \mathit{label}(R_2'(s_1))$. Since $R_1'(s_1), R_2'(s_1), R_1'(s_2)$, and $R_2'(s_2)$ are all H-labeled, we know that $R_1'(s_2) \sim_H R_2'(s_2)$.
    - if $s_1 \notin dom(R_1)$ and $s_2 \notin dom(R_2)$, then $H \sqsubseteq \mathit{label}(R_1))$ and $H \sqsubseteq \mathit{label}(R_2))$.Therefore $H \sqsubseteq \mathit{label}(R_1')$ and $H \sqsubseteq \mathit{label}(R_2')$, which means that $R_1' \sim_H R_2'$.
    - if $s_1 \in dom(R_1)$ and $s_2 \notin dom(R_2)$, then $H \sqsubseteq \mathit{label}(R_2))$ and $H \sqsubseteq \mathit{label}(R_1(s_2))$.Since $R_1 \sim_H R_2$, it must be the case that $H \sqsubseteq \mathit{label}(R_1'(s_2))$ and $H \sqsubseteq \mathit{label}(R_2')$, which means that $R_1' \sim_H R_2'$.
    - if $s_1 \notin dom(R_1)$ and $s_2 \in dom(R_2)$, then this holds with a similar argument to the previous sub-case.
  - Otherwise, $s_1^{l_1} = s_2^{l_2}$. Therefore, $R_1'(s_1) = R_1'(s_2) = v_1 \sqcup l_1$ and $R_2'(s_1) = R_2'(s_2) = v_2 \sqcup l_1$ Since $v_1 \sim_H v_2$, we know that $R_1' \sim_H R_2'$.

- [EVAL]: In this case, $e = \mathit{eval}(e')$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma'', s_i^{k_i}$$
$$e_i = \mathit{stringtoexp}(s_i)$$

  By induction, $\sigma_1'' \approx_H \sigma_2''$

  - If $k_1 = k_2 = H$, then for $i \in 1, 2$:

$$\sigma_i'', \theta_i, e' \Downarrow_H \sigma_i', v_i$$

    By Lemma 4 we know that $H \sqsubseteq \mathit{label}(v_1)$ and $H \sqsubseteq \mathit{label}(v_2)$. Therefore $v_1 \sim_H v_2$.
    By Lemma 6, $\sigma_1' \approx_H \sigma_1'' \approx_H \sigma_2'' \approx_H \sigma_2'$, and we need to conclude that $\sigma_1' \approx_H \sigma_2'$.
    We know that $dom(\sigma_i') \supseteq dom(\sigma_i'')$, since execution only allocates additional reference cells. Without loss of generality, we assume that the two executions allocate reference cells from disjoint parts of the address space *i.e.*:

$$(dom(\sigma_1') \setminus dom(\sigma_1'')) \cap dom(\sigma_2') = \emptyset$$
$$(dom(\sigma_2') \setminus dom(\sigma_2'')) \cap dom(\sigma_1') = \emptyset$$

    Under this assumption, the only common addresses in σ_1' and σ_2' are also the common addresses in σ_1'' and σ_2'', and hence we have that $\sigma_1' \approx_H \sigma_2'$.
  - Otherwise $s_1 = s_2$ and $k_1 = k_2$. Since $s_1 = s_2$ we know that $e_1 = e_2$. Then for $i \in 1, 2$:

$$\sigma_i'', \theta_i, e_1 \Downarrow_{k_1} \sigma_i', v_i$$

    Since $\sigma_1'' \approx_H \sigma_2''$, we know by induction that $\sigma_1' \approx_H \sigma_2'$ and $v_1 \sim_H v_2$.

□