Protection Against Malicious JavaScript Using Hybrid Flow-Sensitive Information Flow Monitoring

by

Bassam Sayed

B.Sc., Helwan University, 2003

M.A.Sc., University of Victoria, 2009

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

© Bassam Sayed, 2015

University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Protection Against Malicious JavaScript Using Hybrid Flow-Sensitive Information Flow Monitoring

by

Bassam Sayed

B.Sc., Helwan University, 2003

M.A.Sc., University of Victoria, 2009

Supervisory Committee

Prof. Dr. Issa Traoré , Supervisor

(Department of Electrical and Computer Engineering)

Prof. Dr. Kin Li, Departmental Member

(Department of Electrical and Computer Engineering)

Prof. Dr. Jens Weber, Outside Member

(Department of Computer Science)

Supervisory Committee

Prof. Dr. Issa Traoré , Supervisor
(Department of Electrical and Computer Engineering)

Prof. Dr. Kin Li, Departmental Member
(Department of Electrical and Computer Engineering)

Prof. Dr. Jens Weber, Outside Member
(Department of Computer Science)

ABSTRACT

Modern web applications use several third-party JavaScript libraries to achieve higher levels of engagement. The third-party libraries range from utility libraries such as jQuery to libraries that provide services such as Google Analytics and context-sensitive advertisement. These third-party libraries have access to most (if not all) the elements of the displayed webpage. This allows malicious third-party libraries to perform attacks that steal information from the end-user or perform an action without the end-user consent. These types of attacks are the stealthiest and the hardest to defend against, because they are agnostic to the browser type and platform of the end-user and at the same time they rely on web standards when performing the attacks. Such kind of attacks can perform actions using the victim's browser without

her permission. The nature of such actions can range from posting an embarrassing message on the victim's behalf over her social network account, to performing online biding using the victim's account. This poses the need to develop effective mechanisms for protecting against client-side web attacks that mainly target the end-user. In the proposed research, we address the above challenges from information flow monitoring perspective by developing a framework that restricts the flow of information on the client-side to legitimate channels. The proposed model tracks sensitive information flow in the JavaScript code and prevents information leakage from happening. The main component of the framework is a hybrid flow-sensitive security monitor that controls, at runtime, the dissemination of information flow and its inlining. The security monitor is hybrid as it combines both static analysis and runtime monitoring of the running JavaScript program. We provide the soundness proof of the model with respect to termination-insensitive non-interference security policy and develop a new security benchmark to establish experimentally its effectiveness in detecting and preventing illicit information flow. When applied to the context of client-side web-based attacks, the proposed model provides a more secure browsing environment for the end-user.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents v

List of Tables viii

List of Figures x

Acknowledgements xi

Dedication xii

1 Introduction 1

1.1 Context 1

1.2 Problem Statement 4

1.3 General Approach 8

1.4 Research Contributions 9

1.5 Dissertation Outline 11

2 Background and Related Work 13

2.1 JavaScript Language Features and Challenges 13

2.1.1 JavaScript Language Features 14

2.1.2 Challenges in Reasoning About JavaScript 15
2.2 Related Work on Detecting Malicious Web Content 15
2.2.1 Language-based Sandboxing 16
2.2.2 Using Machine Learning to Detect Malicious JavaScript 18
2.2.3 Using Web Proxy to Protect End User 20
2.2.4 Virtual Machine Honeynets 21
2.2.5 Dynamic Data Tainting and Static Analysis 21
2.2.6 Limitations 22
2.3 Related Work on Information Flow Control 24
2.4 Summary 27

3 Proposed JavaScript Security Type System 28

3.1 Motivating Example 28
3.2 Syntactic Conventions and Meta-notation 30
3.3 Operational Semantics 31
3.4 Extending The Semantics 34
3.5 Flow-Sensitive Security Type System 36
3.6 Summary and Discussion 48

4 Hybrid Flow-Sensitive Security Monitor For JavaScript 51

4.1 VM Monitor 51
4.2 Attack Model and Security Property 60
4.3 Summary 66

5 IF-Transpiler: Inlining of Hybrid Flow-Sensitive Security Monitor For JavaScript 67

5.1 Transformation Stage 67
5.2 Inlining Stage 68

5.3 Soundness of the Inlined Security Monitor 71
5.4 Summary 78

6 Implementation and Experimental Evaluation 80

6.1 Implementation 80
6.2 Experiments and Results 83
6.2.1 Performance and Size Overhead 84
6.2.2 Performance Comparison 86
6.3 Evaluating Effectiveness 87
6.3.1 Language-Based IFC Benchmark 88
6.3.2 Experiment Results 89
6.4 Summary and Discussion 94

7 Conclusions and Future Work 100

7.1 Summary of Contributions 100
7.2 Limitations and Future Work 101

A Proof of the Soundness of the Hybrid Flow-Sensitive Security Monitor 103

B Proof of the Observational Equivalence of the Inlined Monitor 110

Bibliography 149

List of Tables

Table 2.1 Classification of relevant work in the area of Information Flow Control with respect to flow-sensitivity and analysis approach. 24

Table 3.1 Syntactical conventions, meta-variables, and syntax for values. . . 31

Table 3.2 Syntax of Expressions 32

Table 3.3 Syntax of Statements. 33

Table 3.4 Statement Typing Part(A) 38

Table 3.5 Statement Typing Part(B) 39

Table 3.6 Expression Typing. 49

Table 4.1 Additional statements for monitored execution. 53

Table 4.2 Statements transitions events. 53

Table 4.3 JavaScript operational semantics with events Part (A). 61

Table 4.4 JavaScript operational semantics with events Part (B). 62

Table 4.5 VM Monitor Transitions. 63

Table 4.6 Collecting variables and functions from a statement S. 64

Table 5.1 Transformation function $\mathcal{T}(\mathcal{S})$. 69

Table 5.2 Security level of expression e. 71

Table 5.3 Inlining function $\mathcal{I}(\mathcal{S})$ (Part $\mathcal{A}$). 72

Table 5.4 Inlining function $\mathcal{I}(\mathcal{S})$ (Part $\mathcal{B}$). 73

Table 5.5 Inlining function $\mathcal{I}(\mathcal{S})$ (Part $\mathcal{C}$). 74

Table 6.1 Performance overhead of transformed version of the benchmark JavaScript code. 85
Table 6.2 Performance overhead of instrumented version of the benchmark JavaScript code. 86
Table 6.3 Size overhead of transformed version of the benchmark JavaScript code. 86
Table 6.4 Size overhead of instrumented version of the benchmark JavaScript code. 86
Table 6.5 Performance comparison of our approach with JSFlow and Faceted-Values. 87
Table 6.6 Information flow test cases and their description (Part A). 95
Table 6.7 Information flow test cases and their description (Part B). 96
Table 6.8 Information flow test cases and their description (Part C). 97
Table 6.9 Language feature used in each test case (TC= Test Case, l=loop, r=return, c=continue/break, thr=throw, arr= array, OProp=Object Property, OProto=Object Prototype, and try=try-catch). 98
Table 6.10 Results of the language-based IFC benchmark. 99

List of Figures

Figure 3.1 Flow-Sensitive vs. Flow-Insensitive model example. 29

Figure 4.1 Two examples of the flow-sensitivity attack that demonstrate how the `collect()` and `upgrade()` functions can be used. 57

Figure 4.2 Implicit information flow using block structured control-flow in the left side and using non-block structured control-flow in the right side. 59

Figure 6.1 IF-Transpiler Pipeline Architecture. 82

ACKNOWLEDGEMENTS

It is a pleasure and honour to thank the many people who made this thesis possible:

It is difficult to express my gratefulness and gratitude to my supervisor, Dr. Issa Traore. If it were not for his mentoring, support, encouragement, and patience, this thesis would not have been possible.

I also would like to thank very close friends of mine; Dr. Sherif Saad and Dr. Yousry Abdel-hamid. As they always encouraged me at times of distress.

I am mostly grateful to my mother, Ayda Fahmy. She raised me, supported me, guided me, and loved me.

Lastly, I wish to thank all my family and friends, especially my wife, Dr. Amany Abdelhalim. She supported me, and encouraged me. We passed a lot of hard times together. My brother M.D. Amr Sayed Morsy for his encouragement, guidance, and support.

I almost forgot, I would like to thank my four kids, Muhammad, Amr, Adam, and Ayah for making so much noise and causing so much trouble while I was writing this thesis!!

I strongly believe that, before anybody or anything, if it were not for the tremendous amount of blessing and guidance from Allah (SUB), I would have not achieved what I have achieved, nor be here today. As prophet Mouses peace be upon him once said "My Lord! I am indeed in need of any good You may send down to me!".

Bassam Sayed

DEDICATION

For my mother, my wife, and my kids.

Chapter 1

Introduction

1.1 Context

In the early days of the design of the Internet in the 1960s, the main design goal of the DARPA funded project was to have a network that covers the entire US, coast to coast that does not have a single point of failure. The idea was that if certain parts of the network were destroyed as a result of an attack of the USSR, the rest of the network should recover and function normally. This particular design goal made the researchers at that time focus their research on reliability and fault tolerance of the network rather than security.

The world wide web that we know today is very different from what it used to be over 40 years ago. Initially, the Internet was a set of static HTML pages that contain hyperlinks to other static web pages mainly used by researchers at the universities. Then the Internet evolved into a platform containing images and some interactive content that was based on the client-server architecture. The end-user web browser plays the role of the client and the web server plays the role of the server. Each time the user performs an action the whole web page reloads (disappears and reappears) from

the server even if the action is a simple change in the web page content. Such process of reloading web pages each time the user performs any action proved to be inefficient for the web server and consumes the network bandwidth. Ultimately, the partial content that needs to be updated or changed should only be the content that gets retrieved asynchronously from the server in the background without interfering with the current state of the web page. On the 18th of February 2005 Jesse James Garrett coined the term Ajax (for Asynchronous JavaScript and XML) in an article on the technologies that enable the dynamic asynchronous creation of web content; the title of the article was "Ajax: A new Approach to Web Applications" [18]. On the 5th of April 2006, the World Wide Web Consortium (W3C) released the first draft specification of the XMLHttpRequest object [61] which is one of the main components that enable the creation of Ajax web applications. Ajax as a technology is one of the main factors that enabled the existence of extremely sizable web applications such as Facebook or Twitter. Nowadays the Internet is used for business, pleasure, socialization, and with the emergence of online learning, it is even used for studying. This makes it the ultimate target for cyber-criminals, as more and more end-users spend more time using the Internet for different reasons. Driven mainly by financial gains, cyber-criminals try to steel information or cause damage to connected infrastructures and assets. AJAX with its asynchronous and dynamic nature presents a real challenge to conventional security systems such as Firewalls and Intrusion Detection Systems (IDS). Firewalls must be configured to allow web-based traffic so that end-users can communicate with outside world. Application-based Firewalls must allow web browsers to communicate with the Internet. Network-based IDS systems are blind to encrypted traffic. However, if the traffic is not encrypted it is extremely hard to keep up with the dynamic content of AJAX-based web applications. On the other hand, host-based IDS systems or anti-virus systems can only monitor the web browser behaviour or scan the binaries

downloaded by the web browsers, both are completely blind to the actual content rendered inside the web browser itself.

AJAX is a group of web technologies that can be used together to enable the dynamic asynchronous (happens in the background) web content retrieval from the web server. The process of retrieving content in the background does not interfere with the current state of the page. Garrett in his article [18] listed the following technologies as the enabling tools for AJAX:

- Hyper Text Markup Language (HTML) and Cascaded Style Sheets (CSS) rules for the presentation of the HTML elements.

- Document Object Model (DOM) implemented by web browsers for dynamic display of web content. The DOM object is the data structure that represents the HTML document being rendered.

- Extendable Markup Language (XML) and Extendable Stylesheet Language Transformation (XSLT) for manipulating the XML based data.

- XMLHttpRequest Object implemented by the web browsers for asynchronous communication with the web servers.

- JavaScript programming language to tie all of these technologies together. For example, based on a certain action, using JavaScript an XMLHttpRequest object can be instantiated to request some XML data from the server and when the data arrives the JavaScript will modify the DOM object to insert a new HTML element in the current page and the related CSS rules will be applied on such element. All of these can happen without reloading the entire content of the web page.

In practice the data doesn't have to be in XML format, it can be any kind of format, which means that XSLT is optional as well. The most widely used format is JavaScript

Object Notation (JSON) [62] as the JavaScript has built-in support to interpret and manipulate such format.

1.2 Problem Statement

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 [17]. 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 [63, 60]. 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 [8] and ClojureScript [42] 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" [29]. Dart is only supported by an experimental version of Google Chrome [19]. 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-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 dissertation is to mitigate client-side web attacks by developing a framework for rigorously enforcing information flow policies in the underlying client JavaScript implementation.

1.3 General Approach

In this research, we focus on the JavaScript language since it plays the central role in modern client-side web attacks. In particular, we introduce and develop a hybrid flow-sensitive security monitor that controls, at runtime, the dissemination of information flow and its inlining.

The security monitor is hybrid because it combines both, static analysis and runtime monitoring of the running JavaScript program. The security monitor implements three policies for enforcement: stop, suppress, or rewrite. The enforcement happens whenever there is a flow of private information to a public channel. Our approach operates as a source-to-source compiler (a transpiler), in which, the input is JavaScript source and the output is an instrumented version with the security monitors inlined. Hence the output of our approach is portable JavaScript code that is not tied to a particular JavaScript engine. The transpiler could operate in two different modes. In the first mode, the transpiler could be built as a library (or a browser plugin) that rewrites and transforms the JavaScript code in the webpage just before its execution. This is considered as on-the-fly inlining mode. The second mode, is an offline mode where the input JavaScript code is rewritten and transformed, then stored to a file for later

execution. This is similar in principle to other JavaScript transpilers such as Google's Tracuer source-to-source compiler [28] and BabelJS [33].

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 [47] 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. Finally, we present and discuss the implementation of our inlining transpiler and its performance with respect to un-instrumented code and to other implementations in the literature.

1.4 Research Contributions

Although we believe that there are several sub-contributions to our work we focus only on the following main contributions:

1. The development of a hybrid flow-sensitive security monitor for JavaScript and its soundness proof with respect to termination-insensitive non-interference security policy. This work has been submitted to the ACM Transactions on Programming Languages and Systems (ACM TOPLAS) and it is currently under review [52].

2. The development of the "IF-Transpiler" which is a source-to-source compiler that inline the proposed hybrid flow-sensitive security monitor and its proof of observational equivalence.

3. The implementation of the IF-Transpiler and its performance results with respect to non-instrumented code and to other work in the literature.

4. The development of a language-based information flow security benchmark for evaluating empirically the effectiveness of information flow models protecting against client-side web attacks.

The development and implementation of the IF-Tranpiler along with the language-based benchmark are in the process of being submitted to ELSEVIER's Journal of Computers & Security [53] . The main idea of using information flow control to detect client-side web attacks was published in a workshop collocated with AINA 2014 [51]. The idea was more formulated in the paper that we published in PST 2014 [50]. Our preliminary work in the area of malware and botnet detection yielded two conference papers, one published in PST 2011 [48] and the other in IFIP SEC 2012 [64], and one journal paper published in ELSEVIER's Journal of Computers & Security in 2013 [65].

Compared to other work in the literature that target malicious JavaScript, our proposed framework has the following characteristics:

- Resilient to obfuscated JavaScript.

- Does not enforce the usage of any sort of SDK or libraries.

- Does not require any dataset for training.
- The detection of malicious JavaScript happens in real-time while the program is being executed at runtime.
- Does not depend on any heuristic rule sets.
- Builds upon the most complete JavaScript operational semantics in the literature, proposed by Maffeis *et al.* in [37].

In addition to the previously mentioned points, a comparison of our work to other work in the literature that target information flow control for JavaScript, yields the following characteristics:

- To our knowledge, our framework is the first in the literature that proposes a flow-sensitive information flow control model and its inlined version for JavaScript, along with their proof of correctness.
- Our framework targets the full syntax of the JavaScript language (with minor omissions for some syntactic sugar) in contrast with other works in the literature that focus on a core of the JavaScript language.
- Our approach operates as a source-to-source compiler (a transpiler), in which, the input is JavaScript source and the output is an instrumented version with the flow-sensitive security monitor inlined. Hence the output of our approach is portable JavaScript code that is not tied to a particular JavaScript engine.

1.5 Dissertation Outline

The rest of the dissertation is outlined as follows:

Chapter 2 provides a brief background about malicious JavaScript and its role in client-side web attacks and outlines the related work done in the literature with the detection and mitigation of malicious JavaScript.

Chapter 3 provides a brief outline of the JavaScript language syntax and semantics, outlines the extension of the semantics to support the prototype chain, scope chain and other aspects of the language and ends with a proposed flow-sensitive security type system for JavaScript.

Chapter 4 outlines an augmented version of the JavaScript operational semantics with events and describes the proposed hybrid flow-sensitive security monitor for JavaScript and sketches its proof idea with respect to termination-insensitive non-interference security policy.

Chapter 5 describes the "IF-Transpiler", which is a source-to-source compiler that instruments the JavaScript code to include an inlined version of the security monitor described in the previous chapter. It also sketches the idea of the proof that the inlined security monitor is observationally equivalent to the VM monitor outlined in the previous chapter.

Chapter 6 describes the implementation of the IF-Transpiler, the experiments that we conducted and presents our performance results with respect to un-instrumented code and with respect to other work in the literature.

Chapter 7 concludes by summarizing the contributions and results of the research and discusses future work.

Chapter 2

Background and Related Work

In this chapter we present some of the key features of the JavaScript language along with a discussion of the challenges involved in static analysis of JavaScript code. We cover related work in the literature on the detection of malicious web content and information flow control. We discuss some of the limitations of the related research and highlight the differences with our proposed model.

2.1 JavaScript Language Features and Challenges

Back in 1995, Netscape hired Brendan Eich mainly for the task of developing a new language that could make Netscape Navigator's support to Sun MicroSystems's Java language (currently Oracle) more accessible to non-Java programmers. Brendan came up with a highly dynamic and loosely-typed scripting language initially named "LiveScript". Based on marketing decisions "LiveScript" became "JavaScript". Initially, JavaScript was announced as a complement to both HTML and Sun's Java. In 1996, Microsoft responded with VBScript as their own scripting language for the web and released a port of JavaScript called "JScript". Internet Explorer 3.0 (IE 3.0) was the first browser from Microsoft to support both VBScript and JScript. Although JScript

is a port from JavaScript, there were differences between how IE 3.0 and Netscape Navigator handle JavaScript which led to the efforts to standardize JavaScript. Netscape and Sun turned to the European Computer Manufacturers Association (ECMA) to help in the standardization process of JavaScript. The standardized JavaScript is now known as ECMAScript and the latest version of the standard is ECMAScript 5 which is supported by all major browser vendors such as Microsoft, Apple, Google, Mozilla, and Opera. It is important to note that ECMAScript standard is concerned with the core language but not with the interaction with the DOM which was later standardized by the W3C organization. The W3C is responsible for the standardization of several technologies including HTML, CSS, and XML.

2.1.1 JavaScript Language Features

JavaScript is a dynamic, high-level, object-oriented, untyped interpreted programming language. JavaScript is both an imperative and a functional programming language. The syntax of JavaScript was influenced by Java, its first-class function feature by Scheme, and the prototype based inheritance by Self. Yet, the usage of JavaScript does not mandate the knowledge of all of these features. In the early days, JavaScript was mainly used by non-programmers namely web-designers helping creating interactive web pages. While HTML and CSS define the elements of a web page, and the presentation of the elements, respectively, JavaScript defines the behaviour of such elements.

There are three main categories of JavaScript code. The first category is a standalone JavaScript code which consists of mainly the core language with a limited set of Application Programming Interfaces (APIs) for handling text, dates, arrays and regular expressions. Input/Output, storage, networking, as well as other features are left to be defined by the hosting environment. The second category is a client-side JavaScript

code, in which case the hosting environment is a web browser. The third category is when a JavaScript code is hosted outside web browser such as Rhino [46] from Mozilla, and Node.js [32] which uses Google's V8 JavaScript interpreter (same interpreter used by Google Chrome web browser) to run JavaScript as server-side scripting language with emphases on asynchronous input/output programming model.

2.1.2 Challenges in Reasoning About JavaScript

By default the untyped and dynamic features of JavaScript make it hard to statically reason about the code before actually running it. In fact, the core JavaScript language is full of quirks and foreign syntax compared to other languages such as C++ and Java. Mainly, these quirks are heavily used by attackers to obfuscate their code which complicates the detection of the malicious intent.

2.2 Related Work on Detecting Malicious Web Content

There are several methods for detecting malicious web content, these methods can be grouped into five main groups. The first group is concerned with restricting the JavaScript language to a safer subset and requiring programmers to use the mathematically proven safe subset to program their web content. The second group extracts features from web contents and uses machine learning techniques to classify a web page as malicious or not. The third group uses web proxy to monitor the traffic exchanged between client and server for specific patterns. The fourth group uses honeynets in addition to anomaly detection to decide if a web page is malicious or not and then blacklists such website. We review in this section representative works under each of the above four categories. The fifth group combines dynamic data tainting and

static analysis to detect malicious scripts that disseminate information to un-intended channels.

2.2.1 Language-based Sandboxing

As previously mentioned, JavaScript provides the logic that ties different web technologies together. The first group of related works is concerned with statically analyzing JavaScript code to either infer or prevent the malicious intent. The proposed approaches concentrate on what is known as *language-based* sandboxes. The language-based sandbox consists of two main components, namely, a static code checker that filters out potentially unsafe code and run-time wrappers that limit direct access to the DOM and other capabilities. The idea is to use type theory to mathematically verify the security of the JavaScript code. The main usage of such technology is in deciding whether to allow or not JavaScript code from third-parties to be included within the main web page of a legitimate website. This is known as *mashup* web pages. For example, Facebook applications displayed as part of the main Facebook web page, are developed by possibly untrusted third-parties. Language-based sandboxing allows these untrusted JavaScript codes to run without compromising the integrity and security of the main page displayed. Cyber-criminals could craft a malicious JavaScript code that can access the contents of the main page displayed through the DOM object and transmit such information back to themselves. Language-based sandboxing prevents this attack from happening by using the static code checker to verify and transform the code of the ads before allowing them to be included into the main web page and the run-time wrappers ensure that at run-time the DOM capabilities are limited only to usage of the widgets. In other words, language-based sandboxing provides a framework to isolate the untrusted JavaScript from the main content of the web page they are embedded in. BrowserShield [45], FBJS from Facebook [15], Caja from Google

[20], and ADsafe which is widely used by Yahoo [44], are well known implementations of such approach.

Guha *et al.* in [22] introduced λ_{JS} a small-step operational semantics of the JavaScript language excluding "eval()". The idea was to reduce JavaScript language to a core calculus that models the JavaScript language and simple enough so that it can be used to define a set of security properties. The security properties defined a set of restrictions on the core calculus to limit its capabilities. The authors used in [22] a type system to check the security properties on the core calculus λ_{JS}. In [21], the same authors extended the language-based sandboxing work done in [22] to verify browser extensions. Browser extensions are components that can be downloaded to extend the functionality of the web browsers; they interact with the browser using a set of APIs.

Building on the work done in [22], Poltiz *et al.* in [44] implemented a type-based verification system for sandboxed JavaScript to protect the main content of the web page from advertisement widgets downloaded from third-party servers.

In [16], Finifter *et al.* discovered a vulnerability in ADsafe that could allow the advertisements created by third-parties to access some methods added to the built-in prototypes objects of the hosting page, effectively leaking some of the capabilities of the main page to the third-parties advertisements. Finifter *et al.* proposed an improved statically verified subset of JavaScript that does not contain such limitation.

In [56], Taly *et al.* developed a tool to verify the soundness of a restricted version of JavaScript that cannot circumvent or subvert a given API. The main usage of such tool is to verify that a given API used by third-parties cannot be bypassed in anyway possible. The idea is to sandbox the code provided by third-parties and enforce the usage of the provided API. They applied their tool on ADsafe (widely used by Yahoo) and found a vulnerability that allows the third-party code to break the sandbox and access restricted content.

Although language sandboxing approach is effective when it comes to widgets and mashup web pages, it is completely ineffective if the main page is hosting itself the malicious code. In addition, for such approach to work, third-parties have to be forced to use Software Development Kits (SDKs) from the vendors of the sandboxes.

2.2.2 Using Machine Learning to Detect Malicious JavaScript

The second group of related works concentrates on analyzing web page contents to extract a specific set of features that might indicate if the web page is malicious or not. In [9], Cova *et al.* proposed JSAND, a system that relies on dynamic analysis and anomaly detection. JSAND relies on a set of ten features extracted from the web pages. The system operates in two modes, "training" mode where it learns the characteristics of normal web pages, and "detection" mode were it uses the learned features to detect anomalies in malicious web pages. They extended and used libAnomaly, a library for developing anomaly detection systems. Examples of features extracted by JSAND include the following:

- *Ratio of string definitions and string uses:* measures the number of invocations of JavaScript functions that can be used to define new strings and the number of string uses such as *document.write*() and *eval*().

- *Number of dynamic code executions:* measures the number of DOM changes and function calls that are used to dynamically interpret JavaScript code such as *eval*() and *setTimeout*().

- *Length of dynamically evaluated code:* measures the length of strings passed as arguments to *eval*() function.

Although all of the above features are relevant, attackers can easily evade such approach by not relying on the use of the *eval*() function. As the authors point out, the three features are geared towards obfuscation. Although most of the malicious web pages use obfuscation but so do many benign web pages mainly to protect their intellectual property. Obfuscated JavaScript by itself could be a source of false positive.

Likarish *et al.* in [36] proposed a technique similar to JSAND that extracts features from the web page and uses multiple classifiers to detect malicious JavaScript. They used a web crawler to crawl the top 500 Alexa websites and then reviewed manually the data downloaded from the top 500 websites yielding 62 malicious web pages. Later, they took a portion of the top 500 websites that they consider benign and added the 62 web pages identified as malicious to form a training dataset. They extracted 65 features from the JavaScript scripts in the dataset. Several classifiers were trained using the machine-learning toolkit Weka [23], namely, Naive Bayes, ADTree, Support Vector Machine (SVM), and RIPPER which is a propositional rule learner that greedily grows rules based on information gain. Through a 10-fold cross validation experiment, SVM achieved the highest detection rate of about 92%. Despite the promising results obtained, the approach suffers from the same problem as JSAND, which is the dependency on specific features in malicious JavaScript that can be simply avoided by the attackers or can be found in benign scripts which could be a source of false positive.

In [10], Curtsinger *et al.* developed a tool named Zozzle for in-browser detection of malicious JavaScript. Zozzle uses Bayesian classifier to identify *syntax* elements of the hierarchical structure of the JavaScript abstract syntax tree that are indicative of malicious intent. Although Zozzle is mostly a static JavaScript detector, it does have a run-time component used specifically to unfold obfuscated JavaScript. Although the work done by Curtsinger *et al.* is an in-browser solution capable of detecting malicious JavaScript in real-time, it does not consider browser agnostic attacks that

utilize JavaScript and other web technologies such as XSS attacks.

2.2.3 Using Web Proxy to Protect End User

In [31], Ismail *et al.* proposed an approach to detect XSS attacks by implementing a proxy that analyzes the HTTP traffic exchanged between the client (web browser) and the web application. Their approach relies on scanning the client requests to find if it contains any special characters that represent HTML tags. If the server replied back with the same special characters that would mean that the web application running on that particular server is vulnerable to XSS attacks. The response of the server gets changed by the proxy and the URL of the server is stored in a special database. This approach suffers from two main limitations. Firstly, it only detects what is known as reflected XSS, in other words, the proxy has to see the request and response that contain the special characters to decide if the web application is vulnerable or not. If the XSS attack script is already stored on the web server (stored XSS), the proxy will not be able to detect such attack. The proxy could become a performance bottleneck as it has to parse all the requests and responses going back and forth between the client and the server, which could affect the browsing experience of the user.

In [34], Kirda *et al.* proposed a web proxy named "Noxes" that analyzes browsed web pages for dynamically generated links and compares those links with a set of filtering rules for allowing or disallowing such links. They use a set of heuristics to generate automatic filter rules for suspicious links and then depend on user intervention to allow or disallow such connection. Noxes can best be described as application level firewall for web browsing. However, Noxes assumes that the user can decide on their own whether to allow or disallow a connection. In addition, if the heuristics used are specific for each connection, it would require excessive user intervention which can affect user browsing experience.

2.2.4 Virtual Machine Honeynets

In [40, 41, 59, 39], virtual machine based honeypots have been proposed to detect web pages with malicious content. Typically the VM-based approach decides if the visited web page is malicious or not, by monitoring the interactions between the web browser and the operating system, such as process creation and/or changes to the file system. While VM-based Honeynets may detect malicious web pages (URLs), pinpointing which specific vulnerability was exploited might be difficult in some cases. In addition, configuring and running several VMs with different exploitable software could be time consumming and expensive. Finally, after a malicious web page has been detected, the URL of the page get blacklisted. Few web browser vendors like Google maintain a blacklisting servers that can be contacted by web browsers to generate a warning if the end-user is about to visit a blacklisted website. Clearly, if the blacklist is not updated regularly, the end-user could end up visiting one of the malicious websites. Among the uses of VM-based honeynets are malware capture, and the detection of Botnets.

2.2.5 Dynamic Data Tainting and Static Analysis

In [57], Vogt *et al.* developed an approach that combines both dynamic data tainting and static analysis to detect XSS attacks on the client-side. They modified Mozilla Firefox web browser to implement their technique. The proposed approach involved modifying the JavaScript engine of the Firefox web browser to introduce, propagate, and check for tainted data. The modified Firefox web browser was able to detect XSS attacks in real-time regardless of whether the JavaScript code was obfuscated or not. Dynamic data tainting does not require any training and does not use heuristics. In fact, dynamic data tainting is a special case of information flow control with two security classes. The tracked information is either tainted, which is considered as the

first security class, or not tainted, which is the second security class.

2.2.6 Limitations

Although there are many work done in the literature on the detection of malicious web pages, the proposed approaches suffer from one or more of the following issues:

- **Obfuscated JavaScript**: Since the code of the JavaScript embedded in web pages is served in text form (anybody could see the source code and copy it), the authors of the JavaScript code tend to obfuscate their script to make it hard to be copied. In other words, obfuscated JavaScript does not always mean malicious web page. Relying on obfuscation features in the detection process will likely yield higher false positive rates.

- **Language Sandboxing**: using language sandboxing proved to be successful in limiting a JavaScript code from accessing sensitive information in a web page. However, as mentioned previously, the problem with this approach is the usage of the restricted subset of JavaScript which, usually shipped as a library or SDK with specific set of APIs has to be enforced in one way or another. In other words, if an attacker finds a way to inject his code in the main page or to bypass the enforced usage of the restricted library, the visitors of such website will end up being exposed to malicious scripts.

- **Proper Dataset**: the machine learning-based techniques mentioned previously need adequate datasets for training. Obtaining the proper dataset that genuinely represents web characteristics and use patterns has proven to be quite challenging. Crawling the web is a tedious and hard problem, as it requires a lot of resources. The other problem is labeling. Labeling which web page is malicious or not

requires extensive effort since web crawlers usually visit millions of web pages; labeling them for the purpose of training the classifier is time consuming.

- **Real-time Protection**: with the exception of the tool Zozzle and the web proxies mentioned above, none of the proposed solutions can detect in real-time if the currently visited web page is malicious or not.

- **Usage of Heuristic Rule Sets**: the process of finding the balance between a specific rule set and a general one proved to be hard. Too specific rules could generate a lot of false positives and too general rules would be a source of false negative. Furthermore, using heuristic rule sets with dynamic web content is challenging due to the dynamic nature of modern web applications.

To our knowledge the only existing work that attempts to address the previously mentioned challenges in detecting web attacks was carried out by Vogt and colleagues [57]. Although dynamic data tainting was used successfully by Vogt *et al.* in their proposed model, it suffers from some limitations. Firstly, it does not have the flexibility to define multiple security classes which would affect the usability of the model in different contexts. Secondly, unless formally defined, the model cannot show precisely how the tainted information propagates throughout the system. Thirdly, the taint checking is based on static rules that are domain specific and cannot change dynamically. Fourthly, dynamic data tainting does not clearly define how a tainted data could be untainted (or declassified).

Despite the fact that the work of Vogt *et al.* represents a valuable contribution in securing client-side web, it allows preventing only one type of attack, namely, XSS. Furthermore, the authors have not described formally how the tainted data is introduced, propagated, and checked in the proposed system. We believe it is essential to precisely describe the dynamic taint analysis in a formal manner and show how exactly

Table 2.1: Classification of relevant work in the area of Information Flow Control with respect to flow-sensitivity and analysis approach.

	Analysis Approach		
	Static	**Dynamic**	**Hybrid**
Flow-Insensitive	[12, 13] [58]	[49, 5, 3] [11, 4, 38] [54, 14] [55, 2, 25]	[57, 55, 24]
Flow-Sensitive	[27]		[47, 7] our work

the taint propagation happens when the JavaScript code is executed.

In that regard, we believe that information flow monitoring (also known as information flow control or information flow tracking) is a more formal, and flexible model than the dynamic taint analysis approach used by Vogt *et al.* Information flow monitoring is a security mechanism that focuses on the protection against unauthorized dissemination of information through programs, which makes it capable of detecting stealthy web attacks that steal information about the end-user without her consent.

2.3 Related Work on Information Flow Control

Table 2.1 lists the most relevant work done in the information flow control research area. The intent of the table is to show where exactly our work fits in the general picture of information flow control. In the table, these related works are categorized according to two main factors: *Flow-sensitivity* and *analysis approach*. In flow-sensitive models, 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.

As outlined in Table 2.1, most of the relevant work done in the literature falls under

the *flow-insensitive dynamic* approach. This is in part because of the fact that static approaches are more restrictive than dynamic ones [47] and the dynamic nature of the JavaScript language makes it hard to reason about statically.

To our knowledge, the work in [57], [24], and [55] are the only hybrid analysis approaches that targeted JavaScript, however all of them are flow-insensitive. In particular, in [24] the authors proposed a hybrid approach that is more permissive than the purely dynamic approaches in the literature, where they employed a static component that compensate for the untaken branches in the control-flow. Although, their proposed approach does allow for more permissiveness over purely dynamic approaches, yet still sound, they only cover a small subset of the JavaScript language. For example, their approach does not model prototype based inheritance, exceptions, continue and break statements among other features of the language.

Our model, similar to [47] and [7], implements a *flow-sensitive* type system for information flow tracking as previously presented in [27]. This is in contrast to many proposals in the literature as outlined in Table 2.1 that implement information flow control for JavaScript based on *flow-insensitive* analysis or what is sometimes referred to as *no-sensitive upgrade* policy [3].

In [7], the authors proposed inlining security monitors for a **WHILE** language [1] incorporating a hybrid approach which was proposed by Russo *et al.* in [47]. The hybrid approach proposed in [47] combines static analysis of the conditional branches that were not taken in the current run and dynamic flow-sensitive monitors that update the security levels of variables during the execution of the program. Our work, as a hybrid approach, is closely related to [7] and [47], however we are targeting the JavaScript language.

Magazinius *et al.* in [38] developed a framework to inline dynamic information

[1] A simple imperative language, with assignment to local variables, if statements, while loops, and simple integer and boolean expressions.

flow monitors that track the security level of each variable in the source program. To achieve that, they make use of shadow variables that represent the security level of the variables in the program. In addition they store the current security context level in a *pc* variable to avoid the pitfall of implicit flow as outlined in [13]. Their work in it's essence is similar to ours, however, they presented the formalization of their work on a simple language (with eval() statement) that lakes many features that exist in JavaScript language like objects with properties, scope chain and complex control flow to name a few. These language features present an enormous challenge and require significant amount of additional work. In addition, their framework is *flow-insensitive* in contrast to our approach.

Bello and Bonelli in [6] presented an alternative approach to flow-sensitive monitors that depend on maintaining two caches of dependencies of program points. One cache for direct flows and the other for indirect flows. They make use of the two caches to detect insecure flows when the program run, however, as they stated in their work, it is possible that an initial run of a program misses insecure flows since unexplored passes need to be collected and added to the indirect cache. Their work is similar to ours when it comes to the inlining of the dependency analysis which happens at runtime.

The work in [5], [11], and [14] provide information flow security to JavaScript, however, all of them are flow-insensitive and require modifications to the runtime environment, hence their work is tied to a particular engine. For example, Devriese *et al.* in [14] modified Mozilla SpiderMonkey JavaScript engine which makes their solution tied with this particular JavaScript engine. This is in contrast to our approach of inlining where the security monitors are inlined in the JavaScript code itself hence, it does not depend on a particular engine.

To our knowledge, the work of Santos *et al.* in [49] is the first work in the literature that developed and inlined security monitor to a core of JavaScript. This is in contrast

to modified interpreter approach like in [26, 11, 14, 3, 5]. Inlining security monitors in the code has its advantage over modified interpreter approach in the portability aspect [7]. Since the instrumented code can run on any JavaScript interpreter and can benefit from modern engines that implement Just-In-Time (JIT) compilation and other performance advances such as Google's V8 [30]. However, the work in [49] is flow-insensitive and focuses on a core of JavaScript and does not cover essential aspects of the JavaScript language such as *throw*, *continue*, *break*, or *with* statements and loops. Inlining security checks have also been studied before in [38, 6], however, they worked on a simple imperative language with dynamic code evaluation that lacks the complex features that exist in the JavaScript language. To our knowledge, our work is the first attempt in the literature to inline *flow-sensitive* security monitor for JavaScript by implementing a hybrid analysis approach.

2.4 Summary

In this chapter we presented some of the challenges associated with the analysis of the JavaScript language. We outlined some of the related work in the area of malicious web content detection and discussed about their limitations. We also presented related work in the area of information flow control and we highlighted how our proposed model differs from other work in the literature. In the next chapter we introduce our syntactic conventions and meta-notation that are used in the rest of the dissertation and present our proposed flow-sensitive security type system for the JavaScript language.

Chapter 3

Proposed JavaScript Security Type System

In this chapter 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 [37].

We briefly present the operational semantics of the language and extend them to cover a hypothetical output statement $\texttt{output}_\ell(e)$ 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.

3.1 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 like the ones outlined in Table 2.1, the security classes of the data elements do not float. By allowing the security types of program

```
1  // Global variables
2  var private=?, public = false;
3  if(private) {
4    public = true; // Direct information flow
5  } // Indirect information flow
6
7  // Public's security class is as high as the private's
8  // variable at this point
9
10 public = 10; // Assigning a literal to public variable yields
11              // a low security class
12 // This output statement will be allowed by our model since
13 // the last update to public variable yielded
14 // a low security class.
15 output(public);
16 // Output the value of the public variable on a low security
17 // output channel.
```

Figure 3.1: Flow-Sensitive vs. Flow-Insensitive model example.

data to "float", our type system can accept more programs than other models that are flow-insensitive. For example, the code in Figure 3.1 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 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.

3.2 Syntactic Conventions and Meta-notation

Our proposed meta-variables and syntactical conventions are outlined in Table 3.1. The values outlined in Table 3.1 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 $\mathcal{H}$ to be a mapping between set of references $\mathcal{R}ef$ to set of objects $\mathcal{O}bj$, formally, $\mathcal{H} : \mathcal{R}ef \mapsto \mathcal{O}bj$. In JavaScript, objects

$\vec{x}$		$\{x_1, x_2, ...\}$ % list of elements
$t*$		$t_1...t_n$ % t+ in the nonempty case
$[t]$		t % t is optional, in case of ambiguity
		% the [symbol is escaped by "["
$t \mid s$		% t or s
$\mathcal{Ident}$	$=$	$x \mid y \mid foo \mid bar \mid \ldots$ % Identifiers
$\mathcal{b}$	$=$	$true \mid false$ % Booleans
$\mathcal{m}$	$=$	$"foo" \mid "bar" \mid \ldots$ % Strings
$\mathcal{n}+$	$=$	Infinity $\mid 0 \mid 1 \mid 2 \mid \ldots$ % Positive numbers
$\mathcal{n}$	$=$	$\mathcal{n}+ \mid -\mathcal{n}+ \mid$ NaN % Positive and negative numbers plus NaN
$\mathcal{Prim}$	$=$	$\mathcal{b} \mid \mathcal{m} \mid \mathcal{n} \mid$ null $\mid$ undefined % Primitives
$\mathcal{Ref}$	$=$	r where $r \in dom(H)$ % references to objects in heap memory
va	$::=$	$\mathcal{m} \mid \mathcal{n} \mid \mathcal{b} \mid$ null $\mid$ undefined $\mid r$ % values

Table 3.1: Syntactical conventions, meta-variables, and syntax for values.

are records of primitive values, references, or functions indexed by strings, formally, $\mathcal{Obj} : \mathcal{m} \mapsto \mathcal{Ref} \cup \mathcal{Prim} \cup \mathcal{Func}$, where $\mathcal{Func}$ 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 $[p_1 \mapsto va_1, ..., p_n \mapsto va_n]$ 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.

3.3 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:

[1]This is also true for Arrays and Functions since both are types of objects.

$e ::=$		
	$this$	% the ”this” object
	x	% identifier
	pv	% primitive value
	$"["[\vec{e}]"]"$	% array literal
	$"\{"[\overrightarrow{pn : e}]"\}"$	% object literal
	$"("e")"$	% parenthesis expression
	$e.x$	% property accessor
	$e"["e"]"$	% member selector
	$new\ e["("[\vec{e}]")"]$	% constructor invocation
	$e["("[\vec{e}]")"]$	% function invocation
	$function[x]\ "("[\vec{x}]")"\ "\{"[P]"\}"$	% [named] function expr
	$e\ \Diamond_{po}$	% postfix operator
	$\Diamond_{un}\ e$	% unary operators
	$e\ \Diamond_{bin}\ e$	% binary operators
	$"("e"\ ?\ "e"\ :\ "e")"$	% conditional expression
	(e, e)	% sequential expression
$pn ::=$	n m x	% property name

Table 3.2: Syntax of Expressions

$s ::=$

$"\{" s * "\}"$	% block
$var\ x["="e](x["="e])^*$	% assignment
;	% skip
e	% expression not starting with "*function*"
$if\ "("e")"\ s\ [else\ s]$	% conditional
$while\ "("e")"\ s$	% while
$do\ s\ while\ "("e")";$	% do-while
$for\ "("e\ in\ e")"s$	% for-in
$for\ "("var\ x["="e]\ in\ e")"\ s$	% for-var-in
$continue\ [x];$	% continue
$break\ [x];$	% break
$return\ [e];$	% return
$with\ "("e")"\ s$	% with
$id : s$	% label
$throw\ e;$	% throw
$try\ "\{" s * "\}"$	
$[catch\ "("x")\ \{" s1 * "\}"]$	
$[finally\ "\{" s2 * "\}"]$	% try-catch-finally

Table 3.3: Syntax of Statements.

$$\frac{computation}{\langle current\ state\rangle \rightarrow \langle end\ state\rangle}$$

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 3.2 outlines the syntax for expressions. We use "$\Diamond_{bin}$" to denote binary operator, "$\Diamond_{un}$" to denote unary operator and "$\Diamond_{po}$" 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*".

3.4 Extending The Semantics

We add the following hypothetical output statement $\texttt{output}_\ell(e)$ to the list of statements defined in table 3.3. 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 [32]). The meaning of output statement $\texttt{output}_\ell(e)$ 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 ℓ. This was referred to as observational effect in [47]. The following operational semantics outlines the labeled transition of the output rule (assuming the evaluation of expression e did not generate an exception):

$$\frac{va = H(r)(e)}{\langle H, r, \texttt{output}_\ell(e)\rangle \rightarrow_{o_\ell(va)} \langle H, r, va\rangle}$$

The output event $o_\ell(va)$ is triggered only when stepping of $\texttt{output}_l(e)$ 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 $\texttt{output}_\ell(e)$ statement 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 ℓ is the security level of the output channel. We assume that every channel has a security level ℓ associated with it and it does not change over time. We assume attackers with security level ℓ can only observe outputs from channels having security level lower than or equal to ℓ denoted $\sqsubseteq \ell$, ignoring covert channels outlined in [35].

We define the following two relations for the inspection of the scope and prototype chain of objects [49]. Definition 1 defines the scope inspection relation. If $\langle H, r_0, x\rangle \ \triangleright_{Scope} \ r_1$, 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* $\triangleright_{Scope}$ *is defined recursively as follows:*

$$\text{NULL}\frac{}{\langle H, null, x\rangle \triangleright_{Scope} null} \qquad \text{BASE}\frac{x \in dom(H(r))}{\langle H, r, x\rangle \triangleright_{Scope} r}$$

$$\text{LOOKUP}\frac{x \notin dom(H(r)) \quad \langle H, H(r)(@scope), x\rangle \triangleright_{Scope} r'}{\langle H, r, x\rangle \triangleright_{Scope} r'}$$

Definition 2 defines the prototype-chain inspection relation. JavaScript implements prototypical based object inheritance. Every object (including object literals) includes a *__proto__* property that references its prototype object. The *__proto__* property is used when looking-up properties in objects. For example, if property $m \in dom(r_o)$, then the property lookup returns $r_o(m)$, otherwise the *__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 *__proto__* property is null. This is known as *prototype-chain*. If $\langle H, r, m\rangle \triangleright_{Proto} r'$, 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* $\triangleright_{Proto}$ *is defined recursively as follows:*

$$\text{NULL}\frac{}{\langle H, null, m\rangle \triangleright_{Proto} null} \qquad \text{BASE}\frac{m \in dom(H(r))}{\langle H, r, m\rangle \triangleright_{Proto} r}$$

$$\text{LOOKUP}\frac{m \notin dom(H(r)) \quad \langle H, H(r)(_proto_), m\rangle \triangleright_{Proto} r'}{\langle H, r, m\rangle \triangleright_{Proto} r'}$$

3.5 Flow-Sensitive Security Type System

Security Types τ 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:

- $S = \{l1, l2, ...\}$: 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 3.4 and 3.5. For a statement $\mathcal{S}$, the typing judgements have the form:

$$H, r, pc \vdash_{\mathcal{L}} \Gamma, \Sigma, \Lambda \ \{\mathcal{S}\} \ \Gamma', \Sigma', \Lambda'$$

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 [25]. $\Lambda : \mathcal{N} \rightarrow \langle \mathcal{L}, \mathcal{Id} \rangle$ is a security typing stack that maps integer indices $\mathcal{N}$ to pairs $\langle \mathcal{L}, \mathcal{Id} \rangle$, where $Id = \{\texttt{LOOP,IF,TRY,WITH,FUNC,LBL}\}$ 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 Λ. The program counter pc is always pointing to the top most element of the security typing stack Λ, hence, $pc = tos(\Lambda) = \Lambda(|\Lambda|) = \langle \mathcal{L}, \mathcal{Id} \rangle$, where $tos()$ 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 Λ gets updated, e.g. by pushing an element onto the Λ stack or by removing an element from the Λ

stack. We use the *dot* notation to access the elements of the pair pointed by the *pc*, for example $pc.l = \mathcal{L}$ and $pc.id = \mathcal{Id}$.

The idea is that Γ, Σ, and Λ represent the security levels of all the variables, literals, and primitive values in the current security context that holds before the execution of statement $\mathcal{S}$ and Γ', Σ', and Λ' represent the security levels of all the variables, literals, and primitive values after the execution of statement $\mathcal{S}$ in the current security context. The *pc* is used to eliminate indirect information flows [13] and the typing stack Λ 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 *pc* as illustrated in Definition 3. Definition 3 defines $\triangleright_{ObjectLit}$ relation which models the store of a new object literal "$\{\overrightarrow{pn : e}\}$" in memory and tracks its security typing.

Table 3.4: Statement Typing Part(A)

$$\textbf{T-Block}\ \frac{\begin{array}{c} H, r, pc \vdash \Gamma, \Sigma, \Lambda\{S_1\}\Gamma^1, \Sigma^1, \Lambda^1 \\ H, r, pc^1 \vdash \Gamma^1, \Sigma^1, \Lambda^1\{S_2\}\Gamma^2, \Sigma^2, \Lambda^2 \\ \vdots \\ H, r, pc^{n-1} \vdash \Gamma^{n-1}, \Sigma^{n-1}, \Lambda^{n-1}\{S_n\}\Gamma^n, \Sigma^n, \Lambda^n \end{array}}{H, r, pc \vdash \Lambda, \Gamma\{S^*\}\Gamma^n, \Sigma^n, \Lambda^n}$$

$$\textbf{T-Assign-Var}\ \frac{\Gamma, \Sigma, \Lambda, H, r \vdash E : \tau}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{x = E\}\Gamma' = \Gamma(r)[x \mapsto pc.\ell \sqcup \tau], \Sigma, \Lambda}$$

$$\textbf{T-Assign-Obj-Prop}\ \frac{\Gamma, \Sigma, \Lambda, H, r \vdash E : \tau_E\ , x : \tau_x}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{x.y = E\}\Gamma(r)[x \mapsto [y \mapsto pc.\ell \sqcup \tau_E]], \Sigma(r)[x \mapsto \tau_x \sqcup pc.\ell \sqcup \tau_E], \Lambda}$$

$$\textbf{T-Assign-Obj-Prop}\ \frac{\Gamma, \Sigma, \Lambda, H, r \vdash E_1 : \tau_1, E_2 : \tau_2, x : \tau_x\ \ \langle H, r, E_1\rangle \to \langle H', r, m\rangle}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{x[E_1] = E_2\}\Gamma(r)[x \mapsto [m \mapsto pc.\ell \sqcup \tau_1 \sqcup \tau_2]], \Sigma(r)[x \mapsto \tau_x \sqcup pc.\ell \sqcup \tau_1 \sqcup \tau_2], \Lambda}$$

$$\textbf{T-Assign-Array-Index}\ \frac{\Gamma, \Sigma, \Lambda, H, r \vdash E_1 : \tau_1, E_2 : \tau_2, x : \tau_x\ \ \langle H, r, E_1\rangle \to \langle H', r, n\rangle}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{x[E_1] = E_2\}\Gamma, \Sigma(r)[x \mapsto \tau_x \sqcup pc.\ell \sqcup \tau_1 \sqcup \tau_2], \Lambda}$$

$$\textbf{T-Output}\ \frac{\Gamma, \Sigma, \Lambda, H, r \vdash E : \tau}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{output_\ell(E)\}\Gamma, \Sigma, \Lambda}$$

$$\textbf{T-If-Stmt}\ \frac{\begin{array}{c} \Gamma, \Sigma, \Lambda, H, r \vdash E : \tau \\ \Lambda', pc' = \text{push}(\Lambda, \langle pc.\ell \sqcup \tau, \texttt{IF}\rangle) \\ pc' \vdash \Gamma, \Sigma, \Lambda'\{S_i\}\Gamma', \Sigma', \Lambda'\ i = 1, 2 \quad \Lambda, pc = \text{pop}(\Lambda') \end{array}}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{if(E)\ S_1\ else\ S_2\}\Gamma', \Sigma', \Lambda}$$

$$\textbf{T-While}\ \frac{\begin{array}{c} \Gamma, \Sigma, \Lambda, H, r \vdash E : \tau \\ \Lambda', pc' = \text{push}(\Lambda, \langle pc.\ell \sqcup \tau, \texttt{LOOP}\rangle) \\ H, r, pc' \vdash \Gamma, \Sigma, \Lambda'\{S\}\Gamma', \Sigma', \Lambda' \quad \Lambda, pc = \text{pop}(\Lambda') \end{array}}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{while(E)\ S\}\Gamma', \Sigma', \Lambda}$$

Do-while same as While rule

$$\textbf{T-For-in}\ \frac{\begin{array}{c} \Gamma, \Sigma, \Lambda, H, r \vdash E_2 : \tau \\ \Lambda', pc' = \text{push}(\Lambda, \langle pc.\ell \sqcup \tau, \texttt{LOOP}\rangle) \\ H, r, pc' \vdash \Gamma, \Sigma, \Lambda'\{S\}\Gamma', \Sigma', \Lambda' \quad \Lambda, pc = \text{pop}(\Lambda') \end{array}}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{for(E_1\ in\ E_2)\ S\}\Gamma', \Sigma', \Lambda}$$

$$\textbf{T-For-var-in}\ \frac{\begin{array}{c} \Gamma, \Sigma, \Lambda, H, r \vdash E_1 : \tau_1, E_2 : \tau_2 \\ \Lambda', pc' = \text{push}(\Lambda, \langle pc.\ell \sqcup \tau_1 \sqcup \tau_2, \texttt{LOOP}\rangle) \\ H, r, pc' \vdash \Lambda', \Gamma\{S\}\Gamma', \Sigma', \Lambda' \quad \Lambda, pc = \text{pop}(\Lambda') \end{array}}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{for(x = E_1\ in\ E_2)\ S\}\Gamma', \Sigma', \Lambda}$$

$$\textbf{T-Func-Call}\ \frac{\begin{array}{c} \langle \Gamma, \Sigma, \Lambda, pc, H, r, \texttt{E}(\vec{\texttt{E}})\rangle \rhd_{FuncCall} \langle \hat{\Gamma}, \hat{\Sigma}, \Lambda', pc', \hat{H}, va\rangle, \\ \tau_{ret} = pc'.\ell,\ \Lambda, pc = \text{pop}(\Lambda') \end{array}}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{x = E(\vec{E})\}\hat{\Gamma}(r)[x \mapsto pc.\ell \sqcup \tau_{ret}], \hat{\Sigma}, \Lambda}$$

Table 3.5: Statement Typing Part(B)

$$\textbf{T-Catch}\frac{H, r, pc' \vdash \Gamma, \Sigma, \Lambda'\{S\}\Gamma', \Sigma', \Lambda' \quad \Lambda, pc = \text{pop}(\Lambda')}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{catch(x)\{S\}\}\Gamma', \Sigma', \Lambda}$$

$$\textbf{T-Method-Call}\frac{\langle \Gamma, \Sigma, \Lambda, pc, H, r, \texttt{E[E'](}\vec{\texttt{E}}\texttt{)}\rangle \rhd_{MethCall} \langle \hat{\Gamma}, \hat{\Sigma}, \Lambda', pc', \hat{H}, va\rangle,\ \tau_{ret} = pc'.\ell,\ \Lambda, pc = pop(\Lambda')}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{x = E[E'](\vec{E})\}\hat{\Gamma}(r)[x \mapsto pc.\ell \sqcup \tau_{ret}], \hat{\Sigma}, \Lambda}$$

$$\textbf{T-Finally}\frac{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{S\}\Gamma', \Sigma', \Lambda'}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{finally\{S\}\}\Gamma', \Sigma', \Lambda}$$

$$\textbf{T-Return}\frac{\begin{array}{c}\Gamma, \Sigma, \Lambda, H, r \vdash E : \tau, \\ pc_{old} = \langle pc.\ell \sqcup \tau, _\rangle, \quad while(pc.id \neq \texttt{FUNC})\{\Lambda', _ = \text{pop}(\Lambda)\}, \quad \Lambda'' = \Lambda'[|\Lambda'| \mapsto pc_{old}]\end{array}}{pc \vdash \Gamma, \Sigma, \Lambda\{return\ E\}\Gamma, \Sigma, \Lambda''}$$

$$\textbf{T-Label-Stmt}\frac{\Lambda', pc' = \text{push}(\Lambda, \langle pc.\ell, \texttt{LBL}\rangle) \quad H, r, pc' \vdash \Gamma, \Sigma, \Lambda'\{S\}\Gamma', \Sigma', \Lambda' \quad \Lambda, pc = \text{pop}(\Lambda')}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{lbl : S\}\Gamma', \Sigma', \Lambda}$$

$$\textbf{T-Skip}\frac{}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{;\ \}\Gamma, \Sigma, \Lambda}$$

$$\textbf{T-Cont-lbl, T-Brk-lbl}\frac{\begin{array}{c}pc' = \langle pc.\ell, _\rangle, \quad while(pc.id \neq \texttt{LBL})\ \{\Lambda', _ = \text{pop}(\Lambda)\}, \\ pc'' = \Lambda'(|\Lambda'| - 1), \quad \Lambda'' = \Lambda'[(|\Lambda'| - 1) \mapsto \langle pc'.\ell \sqcup pc''.\ell, _\rangle]\end{array}}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{[continue \mid break]\ \ lbl\}\Gamma, \Sigma, \Lambda''}$$

$$\textbf{T-Continue,T-Break}\frac{\begin{array}{c}pc' = \langle pc.\ell, _\rangle, \quad while(pc.id \neq \texttt{LOOP})\ \{\Lambda', _ = \text{pop}(\Lambda)\}, \\ pc'' = \Lambda'(|\Lambda'| - 1), \quad \Lambda'' = \Lambda'[(|\Lambda'| - 1) \mapsto \langle pc'.\ell \sqcup pc''.\ell, _\rangle]\end{array}}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{[continue \mid break]\}\Gamma, \Sigma, \Lambda''}$$

$$\textbf{T-Throw}\frac{\begin{array}{c}\Gamma, \Sigma, \Lambda, H, r \vdash E : \tau, \\ pc' = \langle pc.\ell \sqcup \tau, _\rangle,\ while(pc.id \neq \texttt{FUNC} \mid \texttt{TRY})\ \{\Lambda', _ = \text{pop}(\Lambda)\}, \quad \Lambda'' = \Lambda'[|\Lambda'| \mapsto pc']\end{array}}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{throw\ E\}\Gamma, \Sigma, \Lambda''}$$

$$\textbf{T-With}\frac{\begin{array}{c}\Gamma, \Sigma, \Lambda, H, r \vdash E : \tau \\ \Lambda', pc' = \text{push}(\Lambda, \langle pc.\ell \sqcup \tau, \texttt{WITH}\rangle), \\ \langle H, r, E\rangle \rightarrow \langle H', r, r_o\rangle, \quad H', r_o, pc' \vdash \Gamma, \Sigma, \Lambda'\{S\}\Gamma', \Sigma', \Lambda' \quad \Lambda, pc = \text{pop}(\Lambda')\end{array}}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{with(E)\ \ S\}\Gamma', \Sigma', \Lambda}$$

$$\textbf{T-Try}\frac{\Lambda', pc' = \text{push}(\Lambda, \langle pc.\ell, \texttt{TRY}\rangle), \quad H, r, pc' \vdash \Gamma, \Sigma, \Lambda'\{S\}\Gamma', \Sigma', \Lambda', \quad \Lambda, pc = \text{pop}(\Lambda', n)}{H, r, pc \vdash \Gamma, \Sigma, \Lambda\{try\{S\}\}\Gamma', \Sigma', \Lambda}$$

$$\textbf{Sub-Typing}\frac{pc_1 \vdash \Lambda, \Gamma_1\{S\}\Lambda, \Gamma_1'}{pc_2 \vdash \Lambda, \Gamma_2\{S\}\Lambda, \Gamma_2'} \quad pc_2 \sqsubseteq pc_1, \Gamma_2 \sqsubseteq \Gamma_1, \Gamma_1' \sqsubseteq \Gamma_2'$$

Definition 3. *New Object Literal Relation* $\rhd_{ObjectLit}$ *is defined as follows:*

$$\dfrac{\begin{gathered}\overrightarrow{pn:e} = \{pn_1 : e_1, pn_2 : e_2 \ldots pn_n : e_n\}, \quad n = |\overrightarrow{pn:e}|,\\ \langle H, r, \mathtt{e_1}\rangle \to \langle H_1 = H(r)[pn_1 \mapsto va_1]\rangle,\\ \vdots\\ \langle H_{n-1}, r, \mathtt{e_n}\rangle \to \langle H_n = H_{n-1}(r)[pn_n \mapsto va_n]\rangle,\\ \tau_1 = H, r, pc \vdash \Gamma, \Sigma, \Lambda\{\mathtt{e_1}\}\Gamma_1 = \Gamma(r)[pn_1 \mapsto \tau_1], \Sigma_1, \Lambda,\\ \vdots\\ \tau_n = H_{n-1}, r, pc \vdash \Gamma_{n-1}, \Sigma_{n-1}, \Lambda\{\mathtt{e_n}\}\Gamma_n = \Gamma_{n-1}(r)[pn_n \mapsto \tau_n], \Sigma_n, \Lambda,\\ r_o \notin dom(H_n),\\ H' = H_n\left[r_o \mapsto \left[\begin{array}{c}_proto_ \mapsto null,\\ pn_i \mapsto va_i\end{array}\right]\right],\\ \Gamma' = \Gamma_n\left[r_o \mapsto \left[\begin{array}{c}_proto_c \mapsto pc,\\ pn_i \mapsto pc \sqcup \tau_i\end{array}\right]\right],\\ \Sigma' = \Sigma_n\left[r_o \mapsto \left(\bigsqcup_{i=1,2..n} \Gamma'(r_o)(pn_i)\right)\right]\end{gathered}}{\langle \Gamma, \Sigma, \Lambda, pc, H, r, \{\overrightarrow{pn:e}\}\rangle \rhd_{ObjectLit} \langle \Gamma', \Sigma', \Lambda, pc, H', r\rangle}$$

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

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

$$\frac{\begin{array}{c}\overrightarrow{e} = \{e_0, e_1 \ldots e_{n-1}\}, \quad n = |\overrightarrow{e}|,\\ \langle H, r, \mathtt{e_0}\rangle \rightarrow \langle H_0 = H(r)[0 \mapsto va_0]\rangle,\\ \vdots \\ \langle H_{n-2}, r, \mathtt{e_{n-1}}\rangle \rightarrow \langle H_{n-1} = H_{n-2}(r)[n-1 \mapsto va_{n-1}]\rangle,\\ \tau_0 = H, r, pc \vdash \Gamma, \Sigma, \Lambda\{\mathtt{e_0}\}\Gamma_0 = \Gamma(r)[0 \mapsto \tau_0], \Sigma, \Lambda,\\ \vdots \\ \tau_{n-1} = H_{n-2}, r, pc \vdash \\ \Gamma_{n-2}, \Sigma, \Lambda\{\mathtt{e_{n-1}}\}\Gamma_{n-1} = \Gamma_{n-2}(r)[n-1 \mapsto \tau_{n-1}], \Sigma, \Lambda,\\ r_a \notin dom(H_{n-1}),\\ H' = H_{n-1}\left[r_a \mapsto \left[\begin{array}{c}_proto_ \mapsto [\,],\\ i \mapsto va_i\end{array}\right]\right],\\ \Gamma' = \Gamma_{n-1}\left[r_a \mapsto \left[\begin{array}{c}_proto_ \mapsto pc,\\ i \mapsto pc \sqcup \tau_i\end{array}\right]\right],\\ \Sigma' = \Sigma\left[r_a \mapsto pc \sqcup \left(\bigsqcup_{i=0,1,..,n-1} \Gamma'(r_a)(i)\right)\right]\end{array}}{\langle \Gamma, \Sigma, \Lambda, pc, H, r, [\vec{e}\,]\rangle \triangleright_{ArrayLit} \langle \Gamma', \Sigma', \Lambda, pc, H', r\rangle}$$

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

Definition 5. *New Function Literal Relation* $\triangleright_{FuncLit}$ *is defined as follows:*

$$\frac{\begin{array}{c} r_f \notin dom(H),\\ H' = H\left[r_f \mapsto \left[\begin{array}{ll} prototype \mapsto \{\}, & @code \mapsto P,\\ @fscope \mapsto r, & @fparam \mapsto \vec{x}\end{array}\right]\right],\\ \Gamma' = \Gamma\left[r_f \mapsto \left[\begin{array}{ll} prototype \mapsto pc, & @code \mapsto pc,\\ @fscope \mapsto pc, & @fparam \mapsto pc\end{array}\right]\right],\\ \Sigma' = \Sigma\,[r_f \mapsto pc]\end{array}}{\langle \Gamma, \Sigma, \Lambda, pc, H, r, \mathtt{function}(\vec{x})\{\mathtt{P}\}\rangle \triangleright_{FuncLit} \langle \Gamma', \Sigma', \Lambda, pc, H', r\rangle}$$

The $\triangleright_{FuncLit}$ relation stores the function body `P` in an internal property @*code* and stores the formal parameters (if any) in @*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 $@fscope$ internal property which points to where the function is defined. Every function object stores a *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 $\triangleright_{FuncCall}$ which models the invocation of a function with arguments "$\mathtt{e}(\vec{\mathtt{e}})$" and tracks its security typing.

In any function invocation the first step is to evaluate the `e` expression in $\mathtt{e}(\vec{\mathtt{e}})$ to a function reference r_f. The function reference is used to retrieve the formal parameters of the function being invoked. Every expression $\mathtt{e_i}$ in the expression vector $\vec{\mathtt{e}}$ is evaluated to a pure value va_i (recall from Table 3.1 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 τ_i in the typed environment Γ. 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 Γ and the structure security typing Σ. A new security context entry is pushed into the security context stack Λ taking the least upper bound of the security levels of the current security context pc and the scope where this function was defined. The $@this$ internal property of the new scope object is assigned the $\#Global$ value since the function is invoked on the global scope.

Definition 6. *Function Call Relation $\triangleright_{FuncCall}$ is defined as follows:*

$$\frac{\begin{array}{c}
\langle \Gamma, \Sigma, \Lambda, pc, H, r, \mathtt{e} \rangle \rightarrow \langle \Gamma_0, \Sigma_0, \Lambda, pc, H_0, r_f \rangle \\
\vec{x} = H(r_f)(@fparam),\ \vec{x} = \{x_1, x_2, ...x_n\} \\
P = H(r_f)(@code),\ r_f^s = H(r_f)(@fscope),\ \vec{e} = \{e_1, e_2, ...e_n\} \\
\langle H_0, r, \mathtt{e_1} \rangle \rightarrow \langle H_1 = H_0(r)[x_1 \mapsto va_1] \rangle, \\
\vdots \\
\langle H_{n-1}, r, \mathtt{e_n} \rangle \rightarrow \langle H_n = H_{n-1}(r)[x_n \mapsto va_n], r \rangle, \\
\tau_0 = H_0, r, pc \vdash \Gamma, \Sigma, \Lambda\{\mathtt{e_1}\}\Gamma_0 = \Gamma(r)[x_1 \mapsto \tau_1], \Sigma, \Lambda, \\
\vdots \\
\tau_n = H_{n-1}, r, pc \vdash \\
\Gamma_{n-1}, \Sigma, \Lambda\{\mathtt{e_n}\}\Gamma_n = \Gamma_{n-1}(r)[x_n \mapsto \tau_n], \Sigma, \Lambda, \\
r' \notin dom(H_n),\ \Lambda', pc' = \mathtt{push}(\langle pc.\ell \sqcup \Gamma(r_f)(@fscope) \sqcup \Sigma(r_f), \mathtt{FUNC} \rangle) \\
H' = H_n \left[r' \mapsto \left[\begin{array}{c} @scope \mapsto r_f^s, \\ @this \mapsto \#Global, \\ x_1 \mapsto va, ..., x_n \mapsto va_n \end{array} \right] \right], \\
\Gamma' = \Gamma_n \left[r' \mapsto \left[\begin{array}{c} @scope \mapsto pc', \\ @this \mapsto \Gamma(\#Global), \\ x_1 \mapsto \tau_1, ..., x_n \mapsto \tau_n \end{array} \right] \right], \\
\Sigma' = \Sigma\,[r' \mapsto pc'] \\
\langle \Gamma', \Sigma', \Lambda', pc', H', r', P \rangle \rightarrow \langle \hat{\Gamma}, \hat{\Sigma}, \Lambda', pc', \hat{H}, \hat{r}, va \rangle
\end{array}}{\langle \Gamma, \Sigma, \Lambda, pc, H, r, \mathtt{e(\vec{e})} \rangle \triangleright_{FuncCall} \langle \hat{\Gamma}, \hat{\Sigma}, \Lambda', pc', \hat{H}, va \rangle}$$

Definition 7 defines relation $\triangleright_{MethodCall}$ which models the calling of a property method with arguments "`e[e'](e⃗)`" 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 "@*this*" is pointing to in the new scope object. As shown previously in Definition 6, the @*this* is assigned the value #*Global* which means it is pointing to the global object, however when the function is invoked as a property method the @*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* $\rhd_{MethodCall}$ *is defined as follows:*

$$\frac{\begin{array}{c}
\langle \Gamma, \Sigma, \Lambda, pc, H, r, \mathtt{e} \rangle \rightarrow \langle \Gamma_0, \Sigma_0, \Lambda, pc, H_0, r_o \rangle, r_o \neq null \\
\langle \Gamma_0, \Sigma_0, \Lambda, pc, H_0, r, \mathtt{e'} \rangle \rightarrow \langle \Gamma_1, \Sigma_1, \Lambda, pc, H_1, m \rangle \\
\langle H_1, r_o, m \rangle \ \rhd_{Proto} \ r_m, \ \ r_f = H_1(r_m)(m) \\
\vec{x} = H_1(r_f)(@fparam), \ \vec{x} = \{x_1, x_2, ...x_n\} \\
P = H_1(r_f)(@code), \ r_f^s = H_1(r_f)(@fscope), \ \vec{e} = \{e_1, e_2, ...e_n\} \\
\langle H_1, r, \mathtt{e_1} \rangle \rightarrow \langle H_2 = H_1(r)[x_1 \mapsto va_1] \rangle, \\
\vdots \\
\langle H_n, r, \mathtt{e_n} \rangle \rightarrow \langle H_{n+1} = H_n(r)[x_n \mapsto va_n] \rangle, \\
\tau_0 = H_1, r, pc \vdash \Gamma_1, \Sigma, \Lambda\{\mathtt{e_1}\}\Gamma_2 = \Gamma_1(r)[x_1 \mapsto \tau_1], \Sigma, \Lambda, \\
\vdots \\
\tau_n = H_n, pc \vdash \\
\Gamma_n, \Sigma, \Lambda\{\mathtt{e_n}\}\Gamma_{n+1} = \Gamma_n(r)[x_n \mapsto \tau_n], \Sigma, \Lambda, \\
r' \notin dom(H_{n+1}), \\
\Lambda', pc' = \mathtt{push}(\Lambda, \langle pc.\ell \sqcup \Gamma_{n+1}(r_f)(@fscope) \sqcup \Sigma(r_m) \sqcup \Sigma(r_o), \mathtt{FUNC} \rangle), \\
H' = H_{n+1}\left[r' \mapsto \left[\begin{array}{c} @scope \mapsto r_f^s, \\ @this \mapsto r_o, \\ x_1 \mapsto va, ..., x_n \mapsto va_n \end{array} \right] \right], \\
\Gamma' = \Gamma_{n+1}\left[r' \mapsto \left[\begin{array}{c} @scope \mapsto pc', \\ @this \mapsto \Gamma(r_o), \\ x_1 \mapsto \tau_1, ..., x_n \mapsto \tau_n \end{array} \right] \right], \\
\Sigma' = \Sigma_{n+1}\left[r' \mapsto pc' \right] \\
\langle \Gamma', \Sigma', \Lambda', pc', H', r', P \rangle \rightarrow \langle \hat{\Gamma}, \hat{\Sigma}, \Lambda', pc', \hat{H}, va \rangle
\end{array}}{\langle \Gamma, \Sigma, \Lambda, pc, H, r, \mathtt{e[e'](\vec{e})} \rangle \rhd_{MethodCall} \langle \hat{\Gamma}, \hat{\Sigma}, \Lambda', pc', \hat{H}, va \rangle}$$

Definition 8 defines relation $\rhd_{ConstrFunc}$ which models the instantiation of an object using a constructor function with arguments "`new e(`$\vec{\mathtt{e}}$`)`" and tracks its security typing.

Definition 8. *Constructor Function Call Relation* $\triangleright_{ConstrFunc}$ *is defined as follows:*

$$\frac{\begin{array}{c}
\langle \Gamma, \Sigma, \Lambda, pc, H, r, \mathtt{e}\rangle \rightarrow \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^s = H(r_f)(@fscope),\ \vec{e} = \{e_1, e_2, ...e_n\} \\
\langle H_0, r, \mathtt{e_1}\rangle \rightarrow \langle H_1 = H_0(r)[x_1 \mapsto va_1]\rangle, \\
\vdots \\
\langle H_{n-1}, r, \mathtt{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\{\mathtt{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\{\mathtt{e_n}\}\Gamma_n = \Gamma_{n-1}(r)[x_n \mapsto \tau_n], \Sigma, \Lambda, \\
r', r_o \notin dom(H_n), \\
\Lambda', pc' = \mathtt{push}(\langle pc.\ell \sqcup \Gamma_n(r_f)(@fscope) \sqcup \Sigma(r_f), \mathtt{FUNC}\rangle), \\
H' = H_n \left[r' \mapsto \left[\begin{array}{c} @scope \mapsto r_f^s, \\ @this \mapsto r_o, \\ x_1 \mapsto va, ..., x_n \mapsto va_n \end{array} \right] \right], \\
\Gamma' = \Gamma_n \left[r' \mapsto \left[\begin{array}{c} @scope \mapsto pc', \\ @this \mapsto pc', \\ x_1 \mapsto \tau_1, ..., x_n \mapsto \tau_n \end{array} \right] \right], \\
\Sigma' = \Sigma\left[r' \mapsto pc'\right] \\
\langle \Gamma', \Sigma', \Lambda', pc', H', r', P\rangle \rightarrow \langle \hat{\Gamma}, \hat{\Sigma}, \Lambda', pc', \hat{H}, \hat{r}, va\rangle \\
\tau = pc'.\ell,\ \Lambda, pc = \mathtt{pop}(\Lambda'), \\
\hat{v} = \begin{cases} va,\ \mathtt{isObj}(va) = true \\ r_o,\ otherwise \end{cases} \quad \tau_{\hat{v}} = \begin{cases} \hat{\Sigma}(va) \sqcup \tau, \mathtt{isObj}(va) = true \\ \hat{\Sigma}(r_o),\ otherwise \end{cases} \\
\Gamma'' = \hat{\Gamma}(r)[\hat{v} \mapsto [_proto_ \mapsto \tau_{\hat{v}} \sqcup \Gamma(r_f)(prototype), ...]] \\
\Sigma'' = \hat{\Sigma}[\hat{v} \mapsto \tau_{\hat{v}}],\ H'' = \hat{H}(r)[\hat{v} \mapsto [_proto_ \mapsto H(r_f)(prototype), ...]]
\end{array}}{\langle \Gamma, \Sigma, \Lambda, pc, H, r, \mathtt{new\ e(\vec{e})}\rangle \triangleright_{ConstrFunc} \langle \Gamma'', \Sigma'', \Lambda, pc, H'', \hat{v}\rangle}$$

Constructor functions are used in JavaScript to instantiate objects and to implement inheritance. When a function is invoked with the `new` operator it is called a constructor function and the result of the evaluation of the `new e(e⃗)` expression is

a reference to a newly created object in memory. If the function's return value [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 `new e(e⃗)` expression. The $\triangleright_{ConstrFunc}$ relation accounts for this fact by checking the type of the return value va of the constructor function using the `isObj()` function (recall from Table 3.1 that va is either a primitive type or a reference to an object). The `isObj()` function returns *true* when the va is pointing to any type that is not a primitive type, formally, $isObj(va) = true$, iff $va \notin \{n, m, b, null, undefined\}$.

Definition 9 defines relation $\triangleright_{ConMeth}$ which models the instantiation of an object using a constructor method with arguments "`new e[e'](e⃗)`" and tracks its security typing. The relation $\triangleright_{ConMeth}$ defined below operates similarly to relation $\triangleright_{ConstrFunc}$ with minor differences. The main difference is the structure security level $\Sigma(r_m)$ of the object that owns the constructor method r_f is accounted for when taking the least upper bound of the current security context pc and the scope where the function was defined $\Gamma(r_f)(@fscope)$.

In the typing rules we write $\Gamma, \Sigma, \Lambda, H, r \vdash \mathcal{E} : \tau$ to mean that expression $\mathcal{E}$ has type τ assuming type environment Γ, structure security Σ, security context stack Λ, heap memory H, and r is a reference to the current scope object. The type τ is defined as a recursive type relation as outlined in Table 3.6. To simplify the typing rules, we do not distinguish between recursive types and their un-foldings following the *equi-recursive* approach outlined in [43].

---

[2] Every function returns a value in JavaScript even functions with no explicit return statements, in which case the return value is *undefined*.

Definition 9. *Constructor Method Call Relation $\rhd_{ConMeth}$ is defined as follows:*

$$\frac{\begin{array}{c}
\langle \Gamma, \Sigma, \Lambda, pc, H, r, \mathtt{e}\rangle \rightarrow \langle \Gamma_0, \Sigma_0, \Lambda, pc, H_0, r_o\rangle, r_o \neq null \\
\langle \Gamma_0, \Sigma_0, \Lambda, pc, H_0, r, \mathtt{e'}\rangle \rightarrow \langle \Gamma_1, \Sigma_1, \Lambda, pc, H_1, m\rangle \\
\langle H_1, r_o, m\rangle \ \rhd_{Proto}\ r_m, \ \ r_f = H_1(r_m)(m) \\
\vec{x} = H_1(r_f)(@fparam),\ \vec{x} = \{x_1, x_2, ...x_n\} \\
P = H_1(r_f)(@code),\ r_f^s = H_1(r_f)(@fscope),\ \vec{e} = \{e_1, e_2, ...e_n\} \\
\langle H_1, r, \mathtt{e_1}\rangle \rightarrow \langle H_2 = H_1(r)[x_1 \mapsto va_1]\rangle, \\
\vdots \\
\langle H_n, r, \mathtt{e_n}\rangle \rightarrow \langle H_{n+1} = H_n(r)[x_n \mapsto va_n]\rangle, \\
\tau_0 = H_1, r, pc \vdash \Gamma_1, \Sigma_1, \Lambda\{\mathtt{e_1}\}\Gamma_2 = \Gamma_1(r)[x_1 \mapsto \tau_1], \Sigma_1, \Lambda, \\
\vdots \\
\tau_n = H_n, pc \vdash \\
\Gamma_n, \Sigma_1, \Lambda\{\mathtt{e_n}\}\Gamma_{n+1} = \Gamma_n(r)[x_n \mapsto \tau_n], \Sigma_1, \Lambda, \\
r', r_o \notin dom(H_{n+1}), \\
\Lambda', pc' = \\
\mathtt{push}(\langle pc.\ell \sqcup \Gamma_{n+1}(r_f)(@fscope) \sqcup \Sigma_1(r_m) \sqcup \Sigma_1(r_o), \mathtt{FUNC}\rangle), \\
H' = H_{n+1}\left[r' \mapsto \left[\begin{array}{c} @scope \mapsto r_f^s, \\ @this \mapsto r_o, \\ x_1 \mapsto va, ..., x_n \mapsto va_n \end{array}\right]\right], \\
\Gamma' = \Gamma_{n+1}\left[r' \mapsto \left[\begin{array}{c} @scope \mapsto pc', \\ @this \mapsto pc', \\ x_1 \mapsto \tau_1, ..., x_n \mapsto \tau_n \end{array}\right]\right], \\
\Sigma' = \Sigma_1\,[r' \mapsto pc'] \\
\langle \Gamma', \Sigma', \Lambda', pc', H', r', P\rangle \rightarrow \langle \hat{\Gamma}, \hat{\Sigma}, \Lambda', pc', \hat{H}, \hat{r}, va\rangle \\
\tau = pc'.\ell,\ \Lambda, pc = \mathtt{pop}(\Lambda'), \\
\hat{v} = \left\{\begin{array}{l} va, \mathtt{isObj}(va) = true \\ r_o,\ otherwise \end{array}\right. \quad \tau_{\hat{v}} = \left\{\begin{array}{l} \hat{\Sigma}(va) \sqcup \tau, \mathtt{isObj}(va) = true \\ \hat{\Sigma}(r_o),\ otherwise \end{array}\right. \\
\Gamma'' = \hat{\Gamma}(r)[\hat{v} \mapsto [_proto_ \mapsto \tau_{\hat{v}} \sqcup \Gamma(r_f)(prototype), ...]] \\
\Sigma'' = \hat{\Sigma}[\hat{v} \mapsto \tau_{\hat{v}}],\ H'' = \hat{H}(r)[\hat{v} \mapsto [_proto_ \mapsto H(r_f)(prototype), ...]]
\end{array}}{\langle \Gamma, \Sigma, \Lambda, pc, H, r, \mathtt{new\ e[e'](\vec{e})}\rangle \rhd_{ConMeth} \langle \Gamma'', \Sigma'', \Lambda, pc, H'', \hat{v}\rangle}$$

The previously defined relations and the typing rules in Tables 3.4 and 3.5 use two auxiliary functions, namely, `push()` and `pop()`. Both functions operate on the security context stack Λ. The $\Lambda', pc' = \texttt{push}(\Lambda, arg_1, arg_2, ...)$ function pushes one or more arguments onto the Λ 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 $\Lambda, x = \texttt{pop}(\Lambda')$ function pops the top-most element from the security context stack Λ returning a reference to the modified stack and the value of the nth element pointed to by x. When the value returned from the `pop()` function is not needed we use the "_" symbol to mean the value is discarded.

Based on the rules in Tables 3.6, 3.4, and 3.5 the type system satisfies a straightforward confidentiality condition. An attacker observing outputs from channel with type τ should only be able to observe inputs with types $\sqsubseteq \tau$. More precisely, given a derivation $\vdash \Lambda, \Sigma, \Gamma\ \{\texttt{output}_\ell(E)\}\ \Lambda, \Sigma, \Gamma$, 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 $lit : \tau_2$, and primitive values $pv : \tau_3$ with initial types $\tau_1 \sqcup \tau_2 \sqcup \tau_3 \sqsubseteq \tau$, where $\sqcup$ is the least upper bound operator.

3.6 Summary and Discussion

In this chapter we started with the meta-notation and syntactic conventions of the JavaScript language, then we briefly introduced the semantics of the language, then we extended the semantics of the language with relations that model the scope chain, prototype chain, and other aspects of the language. At the end of the chapter, we introduced our flow-sensitive type system for JavaScript.

Although our flow-sensitive type system is based on the work of Hunt and Sands in [27] where they proved noninterference of the type system, as stated in [47], it is diffi-

Table 3.6: Expression Typing.

$\Gamma, \Sigma, \Lambda, pc, H, r \vdash E : \tau$
iff $\tau =$ **case** E **of**

;	$\models$	$\bot_{\mathcal{L}}$
$this$	$\models$	$\Gamma(r)(@this) \to \mathcal{L}$
r_o	$\models$	$\Sigma(r_o) \to \mathcal{L}$, iff $r_o \in dom(H(r))$
x	$\models$	$\Gamma(r')(x) \to \mathcal{L}$, iff $\langle H, r, x\rangle \rhd_{Scope} r' \wedge r' \neq null$
pv	$\models$	$\bot_{\mathcal{L}}$
"{ }"	$\models$	$\Sigma'(r_o) \to \mathcal{L}$, iff $\langle ...\Sigma, \{\ \}\rangle \rhd_{ObjectLit} \langle ...\Sigma', r_o\rangle$
"[" "]"	$\models$	$\Sigma'(r_a) \to \mathcal{L}$, iff $\langle ...\Sigma, \texttt{[ ]}\rangle \rhd_{ArrayLit} \langle ...\Sigma', r_a\rangle$
"["$\vec{e}$"]"	$\models$	$\Sigma'(r_a) \to \mathcal{L}$, iff $\langle ...\Sigma,$ "["$\vec{e}$"]"$\rangle \rhd_{ArrayLit} \langle ...\Sigma', r_a\rangle$
$\{\overrightarrow{pn : e}\}$	$\models$	$\Sigma'(r_o) \to \mathcal{L}$, iff $\langle ...\Sigma, \{\overrightarrow{pn : e}\}\rangle \rhd_{ObjectLit} \langle ...\Sigma', r_o\rangle$
"("e")"	$\models$	$\Gamma, \Sigma, \Lambda, H, r \vdash e : \tau$
$x.m$	$\models$	$\Gamma(r')(m) \sqcup \Sigma(r_o) \to \mathcal{L}$, iff $\langle H, r, x\rangle \rhd_{Scope} r_o$ $\wedge \langle H, r_o, m\rangle \rhd_{Proto} r'$ $\wedge\ r' \neq null \wedge\ m \in dom(H(r'))$
$x.m$	$\models$	$\Sigma(r_o) \to \mathcal{L}$, iff $\langle H, r, x\rangle \rhd_{Scope} r_o$ $\wedge \langle H, r_o, m\rangle \rhd_{Proto} r'$ $\wedge\ r' = null$
$e\Diamond_{po}$	$\models$	$\Gamma, \Sigma, \Lambda, H, r \vdash e : \tau$
$\Diamond_{un} e$	$\models$	$\Gamma, \Sigma, \Lambda, H, r \vdash e : \tau$
$e_1 \Diamond_{bin} e_2$	$\models$	$\Gamma, \Sigma, \Lambda, H, r \vdash e_1 : \tau_1, e_2 : \tau_2 \to \tau_1 \sqcup \tau_2$
$e([\vec{e}\,])$	$\models$	$pc'.\ell \to \mathcal{L}$, iff , $\langle ..., pc, \texttt{e(}\vec{\texttt{e}}\texttt{)}\rangle \rhd_{FuncCall} \langle ..., pc', va\rangle$
$e[e']([\vec{e}\,])$	$\models$	$pc'.\ell \to \mathcal{L}$, iff , $\langle ..., pc, \texttt{e[e'](}\vec{\texttt{e}}\texttt{)}\rangle \rhd_{MethodCall} \langle ..., pc', va\rangle$
$new\ e([\vec{e}\,])$	$\models$	$\Sigma''(\hat{v}) \to \mathcal{L}$, iff , $\langle ..., \Sigma, \texttt{new e(}\vec{\texttt{e}}\texttt{)}\rangle \rhd_{ConstrFunc} \langle ..., \Sigma'', \hat{v}\rangle$
$new\ e[e']([\vec{e}\,])$	$\models$	$\Sigma''(\hat{v}) \to \mathcal{L}$, iff , $\langle .., \Sigma, \texttt{new e[e'](}\vec{\texttt{e}}\texttt{)}\rangle \rhd_{ConMeth} \langle .., \Sigma'', \hat{v}\rangle$
$function\ ([\vec{x}])\{[P]\}$	$\models$	$\Sigma'(r_f) \to \mathcal{L}$, iff $\langle ..., \Sigma, ...\rangle \rhd_{FuncLit} \langle ..., \Sigma', r_f\rangle$
$x[e]$	$\models$	$\Gamma(r')(m) \sqcup \Sigma(r_o) \to \mathcal{L}$, iff $\langle ..., H, e\rangle \to \langle ..., H', m\rangle \wedge$ $\langle H', r, x\rangle \rhd_{Scope} r_o \wedge r_o \neq null \wedge$ $\langle H, r_o, m\rangle \rhd_{Proto} r' \wedge\ m \in dom(H(r'))$
$x[e]$	$\models$	$\Sigma(r_o) \to \mathcal{L}$, iff $\langle H, r, x\rangle \rhd_{Scope} r_o$ $\wedge \langle ..., H, e\rangle \to \langle ..., H', m\rangle$ $\wedge \langle H', r_o, m\rangle \rhd_{Proto} r'$ $\wedge\ r' = null$

cult to extend their work to languages with dynamic code evaluation (e.g. JavaScript). Russo and Sabelfeld in [47] stated that dynamic monitoring of program execution is one appropriate method to deal with languages featuring dynamic code evaluation. However, their main contribution is the proof of impossibility of a sound, purely dynamic information flow monitor that accepts programs certified by classical flow-sensitive static analysis like the ones in [27]. To that end, in the next chapter we present our hybrid flow-sensitive information flow monitor where we combine the static analysis of the type system with dynamic flow-sensitive monitoring.

Chapter 4

Hybrid Flow-Sensitive Security Monitor For JavaScript

In this chapter, we present a sound flow-sensitive monitor by combining static analysis of the type system introduced in the previous chapter 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.

4.1 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 3.3 with the statements outlined in Table 4.1. The new statements (commands) are similar to the ones defined in [7]:

- $\square$: statement with no transition; it is used to serve as a terminal configuration in the semantics.

- $\rtimes$: generates "join-of-branch" event j that makes the VM monitor synchronize with joint points in the control-flow of the running program.

- $\checkmark$: generates "step" event t. The $\checkmark$ 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 4.3 and 4.4 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 4.2. 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 $o_\ell(e, va)$ event which generates an observable output event w as previously discussed in subsection 3.4. The transition events outlined in Table 4.2 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:

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

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 α, updates its state, and generates γ which is either an observable output event w or nothing.

$s ::=$	s	% previously presented in Table 3.3
	$\mid\ output_\ell(e)$	% output statement
	$\mid\ \square\ \mid\ \rtimes\ \mid\ \checkmark$	% additional statements

Table 4.1: Additional statements for monitored execution.

Statements events	β	$::=$	$\alpha \mid t$
	α	$::=$	$\epsilon \mid o_\ell(e, va)$
	ϵ	$::=$	$a_v(x, e) \mid a_p(x.m, e)$
			$\mid a_i(x[m], e) \mid k(lbl)$
			$\mid b(e, S) \mid b(\vec{e}, S)$
			$\mid sk \mid j$
Obervations	w	$::=$	$o_\ell(va)$

Table 4.2: Statements transitions events.

All the symbols in the VM monitor configuration have been introduced before with the exception of the Δ symbol, which is a stack of pairs $\langle \vec{x}, \ell \rangle$. The $\vec{x}$ is a list of variables and ℓ is a security level. The *control-flow stack* Δ is used to capture the security level ℓ 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 ℓ is the security level of the conditional expression of the control-flow statement.

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

$$\frac{\langle H, r, s \rangle \xrightarrow{\alpha} \langle H', r, s' \rangle, \quad \langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r \rangle \rangle \xrightarrow{\alpha}_{\gamma} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc \rangle, \langle H, r \rangle \rangle}{\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, s \rangle \rangle \xrightarrow{\gamma} \langle \langle \Gamma', \Sigma', \Lambda', \Delta', pc \rangle, \langle H', r, s' \rangle \rangle}$$

Similar to [7] 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

$$\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r,s\rangle\rangle \rightarrow^{*} \xrightarrow{w} \rightarrow^{*} \langle\langle\Gamma',\Sigma',\Lambda',\Delta',pc\rangle,\langle H',r,s'\rangle\rangle$$

and we write

$$\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r,s\rangle\rangle \stackrel{\vec{w}}{\Longrightarrow} \langle\langle\Gamma',\Sigma',\Lambda',\Delta',pc\rangle,\langle H',r,s'\rangle\rangle$$

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

Table 4.5 presents the VM monitor transitions. The labeled transitions of the VM monitor make use of the following functions:

- $\vec{x} = \texttt{collect}(S)$: 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 4.6. The collect function uses a helper function `colFunc(e)` that collects function calls in expression e. The collect function is used in rule **M-Branch** outlined in Table 4.5.

- $\Gamma',\Sigma' = \texttt{upgrade}(\vec{x},\ell,\Gamma,\Sigma,H,r)$: upgrades variables in list $\vec{x}$ to the security level ℓ. The upgrade function is used in rule **M-Join** outlined in Table 4.5. The upgrade function is defined as follows:

```
proc upgrade(x⃗, ℓ, Γ, Σ, H, r)  ≡
  ⟨x⃗ = {x_1, x_2, .., x_n}, ⟨H, r, x_i⟩ ▷_scope r_x⟩;
  begin
    for (i := 1 to n step 1) do
      if (isFunc(x_i) = true)
        Γ(r_x)[x_i ↦ [@fscope ↦ Γ(r_x)(@fscope) ⊔ ℓ]];
        Σ(r_x)[x_i ↦ Σ(r_x)(x_i) ⊔ ℓ];
     else if (isObj(x_i) = true)
          Σ(r_x)[x_i ↦ Σ(r_x)(x_i) ⊔ ℓ];
      else
          Γ(r_x)[x_i ↦ Γ(r_x)(x_i) ⊔ ℓ];
        fi
      fi
    end
  end.
```

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 4.1 which is an adaptation of the flow-sensitivity attack example in [47] . 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

4.3. When the VM (running program) executes the *else* branch and generates the *join* event as per rule **End** in Figure 4.4, the VM monitor will upgrade the variables (and the functions' "@fscope" property) that have been collected previously, hence, upgrading the security class of variable $flag$ (or function f() "@fscope" 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 $pc.\ell$ 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.

The same scenario will happen with the second If-Statement in Figure 4.1, 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.

- $\Lambda' = \texttt{comp}_{\Lambda}(lbl, \ell)$: 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 Λ with security level ℓ. The updating starts from the inner most context, until the context above the one labeled *lbl*. The $\texttt{comp}_{\Lambda}(lbl, \ell)$ function is defined as follows:

```
1  // Global variables
2  var secret=?,low=true,flag=false;
3
4      if(secret)
5        flag = true;
6      else
7        skip;
8
9
10     if(!flag)
11       low = false;
12     else
13       skip;
14
15     output(low);
16
17
18
19
20
21
```

```
// Global variables
var secret=?,low=true,flag=false;
   function f() {
      flag = true;
   }

   function g() {
      low = false;
   }

   if(secret)
      f();
   else
      skip;

   if(!flag)
      g();
   else
      skip;

   output(low);
```

Figure 4.1: Two examples of the flow-sensitivity attack that demonstrate how the `collect()` and `upgrade()` functions can be used.

$\mathsf{proc\ comp}_\Lambda(lbl, \ell) \ \equiv$

$\quad \mathsf{var}\ \langle \mathsf{i} := |\Lambda| \rangle;$

$\quad\quad \mathsf{begin}$

$\quad\quad\quad \mathsf{while}\ (\mathsf{i} > 1\ \wedge\ \Lambda(\mathsf{i}).id \neq lbl)\ \mathsf{do}$

$\quad\quad\quad\quad\quad \mathsf{i} := \mathsf{i} - 1;$

$\quad\quad\quad\quad\quad \Lambda(\mathsf{i}).\ell := \Lambda(i).\ell \sqcup \ell;$

$\quad\quad\quad \mathsf{end}$

$\quad\quad\quad \mathsf{i} := \mathsf{i} - 1;$

$\quad\quad\quad \Lambda(\mathsf{i}).\ell := \Lambda(i).\ell \sqcup \ell;$

$\quad\quad \mathsf{end.}$

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 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 4.2 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 *high* to the lower security level variable *low*. In Figure 4.2, 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. [1]

The VM monitor supports three different policies, namely, *OutputFailStop, OutputSuppress, and OutputDefault*, when it comes to how the VM monitor deals with

[1]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.

```
1  // Global variables
2  var high=?, low = true;
3  function f() {
4    if(high)
5      return;
6    low = false;
7  }
8  // Calling function "f"
9  f();
10 output(low);
```

```
// Global variables
var high=?, low = true;
function g() {
  if(high)
    throw true;
  low = false;
}
// Calling function "g"
try { g(); output(low); }
catch(ex) {output(low); }
```

Figure 4.2: Implicit information flow using block structured control-flow in the left side and using non-block structured control-flow in the right side.

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 4.5. 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 4.5. The *OutputDefault* policy means that the monitor either will allow the output command to execute or will output a fixed literal value D, this is outlined in rule **M-OutputDefault** in Table 4.5.

4.2 Attack Model and Security Property

Our attack model is similar to [47] and [7]. 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 ℓ. 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 (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.

Table 4.3: JavaScript operational semantics with events Part (A).

$$\textbf{Skip}\frac{}{\langle H, r, ;\ \rangle \xrightarrow{sk} \langle H, \square\rangle} \quad \textbf{Seq-A}\frac{\langle H, r, \texttt{S}_1\rangle \xrightarrow{\alpha} \langle H', r', \square\rangle}{\langle H, r, \texttt{S}_1;\ \texttt{S}_2\rangle \xrightarrow{\alpha} \langle H', r', \texttt{S}_2\rangle} \quad \textbf{Seq-B}\frac{\langle H, r, \texttt{S}_1\rangle \xrightarrow{\alpha} \langle H', r', \texttt{S'}_1\rangle \quad \texttt{S'}_1 \neq \square}{\langle H, r, \texttt{S}_1;\ \texttt{S}_2\rangle \xrightarrow{\alpha} \langle H', r', \texttt{S'}_1;\ \texttt{S}_2\rangle}$$

$$\textbf{Assign-Var}\frac{\langle H, r, x\rangle \rhd_{Scope} r_x,\ \langle H, r, e\rangle \rightarrow \langle H', r, va\rangle}{\langle H, r, \texttt{x=e}\rangle \xrightarrow{a_v(x,e)} \langle H'(r_x)[x \mapsto va], \square\rangle}$$

$$\textbf{Assign-Obj-Prop}\frac{\langle H, r, x\rangle \rhd_{Scope} r_o, r_o \neq null,\ \langle H, r, e\rangle \rightarrow \langle H', r, va\rangle}{\langle H, r, \texttt{x.y=e}\rangle \xrightarrow{a_p(x.y,e)} \langle H'(r_o)[x \mapsto [y \mapsto va]], \square\rangle}$$

$$\textbf{Assign-Obj-Prop}\frac{\langle H, r, x\rangle \rhd_{Scope} r_o, r_o \neq null,\ \ \langle ..., \texttt{e}_1\rangle \rightarrow \langle H', r, m\rangle,\ \langle H', r, \texttt{e}_2\rangle \rightarrow \langle H'', r, va\rangle}{\langle H, r, \texttt{x[e}_1\texttt{]=e}_2\rangle \xrightarrow{a_p(x.m,e_1,e_2)} \langle H(r_o)[x \mapsto [m \mapsto va]], \square\rangle}$$

$$\textbf{Assign-Array-Index}\frac{\langle H, r, x\rangle \rhd_{Scope} r_o, r_o \neq null,\ \langle ..., \texttt{e}_1\rangle \rightarrow \langle H', r, m\rangle,\ \langle H', r, \texttt{e}_2\rangle \rightarrow \langle H'', r, va\rangle}{\langle H, r, \texttt{x[e}_1\texttt{]=e}_2\rangle \xrightarrow{a_i(x[m],e_1,e_2)} \langle H(r_a)[x \mapsto [m \mapsto va]], \square\rangle}$$

$$\textbf{If-Then}\frac{\langle H, r, e\rangle \rightarrow \langle H', r, va\rangle,\ va \notin \{\texttt{0}, \texttt{null}, \texttt{undefined}, \texttt{false}\}}{\langle H, r, \texttt{if(e) S}_1\texttt{ else S}_2\rangle \xrightarrow{b(e,S_2)} \langle H', \rtimes;\ S_1\rangle}$$

$$\textbf{If-Else}\frac{\langle H, r, e\rangle \rightarrow \langle H', r, va\rangle,\ va \in \{\texttt{0}, \texttt{null}, \texttt{undefined}, \texttt{false}\}}{\langle H, r, \texttt{if(e) S}_1\texttt{ else S}_2\rangle \xrightarrow{b(e,S_1)} \langle H', \rtimes;\ S_2\rangle}$$

$$\textbf{While-Loop}\frac{\langle H, r, e\rangle \rightarrow \langle H', r, va\rangle,\ va \notin \{\texttt{0}, \texttt{null}, \texttt{undefined}, \texttt{false}\}}{\langle H, r, \texttt{while(e) S}\rangle \xrightarrow{b(e,skip)} \langle H', r, S;\ \rtimes;\ \texttt{while(e) S}\rangle}$$

$$\textbf{While-Skip}\frac{\langle H, r, e\rangle \rightarrow \langle H', r, va\rangle,\ va \in \{\texttt{0}, \texttt{null}, \texttt{undefined}, \texttt{false}\}}{\langle H, r, \texttt{while(e) S}\rangle \xrightarrow{b(e,S)} \langle H', r, \rtimes\rangle}$$

$$\textbf{For-Loop}\frac{\langle H, r, e_2\rangle \rightarrow \langle H', r, va\rangle,\ va \notin \{\texttt{0}, \texttt{null}, \texttt{undefined}, \texttt{false}\}}{\langle H, r, \texttt{for(e}_1\texttt{; e}_2\texttt{; e}_3\texttt{) S}\rangle \xrightarrow{b(e_2,skip)} \langle H', S;\ \rtimes;\ \texttt{for(e}_1\texttt{; e}_2\texttt{; e}_3\texttt{) S}\rangle}$$

$$\textbf{For-Skip}\frac{\langle H, r, e_2\rangle \rightarrow \langle H', r, va\rangle,\ va \in \{\texttt{0}, \texttt{null}, \texttt{undefined}, \texttt{false}\}}{\langle H, r, \texttt{for(e}_1\texttt{; e}_2\texttt{; e}_3\texttt{) S}\rangle \xrightarrow{b(e_2,S)} \langle H', \rtimes;\ \rangle}$$

$$\textbf{For-var-in-Loop}\ \frac{\langle H, r, e_2\rangle \rightarrow \langle H', r_o\rangle,\ \text{isObj}(r_o) = true}{\langle H, r, \texttt{for(var x=e}_1\texttt{ in e}_2\texttt{) S}\rangle \xrightarrow{b(\vec{e}=\{e_1,e_2\},skip)} \langle H', S;\ \rtimes;\ \texttt{for(x in e}_2\texttt{) S}\rangle}$$

$$\textbf{For-var-in-Skip}\frac{\langle H, r, e_2\rangle \rightarrow \langle H', r_o\rangle,\ \text{isObj}(r_o) = false}{\langle H, r, \texttt{for(var x=e}_1\texttt{ in e}_2\texttt{) S}\rangle \xrightarrow{b(\vec{e}=\{e_1,e_2\},S)} \langle H', \rtimes\rangle}$$

$$\textbf{For-in-Loop}\frac{\langle H, r, e\rangle \rightarrow \langle H', r_o\rangle,\ \text{isObj}(r_o) = true}{\langle H, r, \texttt{for(x in e) S}\rangle \xrightarrow{b(e,skip)} \langle H', S;\ \rtimes;\ \texttt{for(x in e) S}\rangle}$$

Table 4.4: JavaScript operational semantics with events Part (B).

$$\textbf{For-in-Skip}\ \frac{\langle H, r, e\rangle \rightarrow \langle H', r_o\rangle,\ \text{isObj}(r_o) = false}{\langle H, r, \texttt{for(x in e) S}\rangle \xrightarrow{b(e,S)} \langle H', \rtimes\rangle}$$

$$\textbf{Func-Call}\ \frac{\langle H, r, \texttt{e}(\vec{\texttt{e}})\rangle \rhd_{FuncCall} \langle \hat{H}, va\rangle, \langle \hat{H}, r, x\rangle \rhd_{Scope} r_x}{\langle H, r, \texttt{x=e}(\vec{\texttt{e}})\rangle \xrightarrow{sk} \langle \hat{H}(r_x)[x \mapsto va], \Box\rangle}$$

$$\textbf{Meth-Call}\ \frac{\langle H, r, \texttt{e[e']}(\vec{\texttt{e}})\rangle \rhd_{MethCall} \langle \hat{H}, va\rangle, \langle \hat{H}, r, x\rangle \rhd_{Scope} r_x}{\langle H, r, \texttt{x=e[e']}(\vec{\texttt{e}})\rangle \xrightarrow{sk} \langle \hat{H}(r_x)[x \mapsto va], \Box\rangle}$$

$$\textbf{Return}\ \frac{\langle H, r, e\rangle \rightarrow \langle H', r, va\rangle}{\langle H, r, \texttt{return e}\rangle \xrightarrow{k(\texttt{FUNC})} \langle H', \Box\rangle} \quad \textbf{Label-Stmt}\ \frac{lbl = m}{\langle H, r, \texttt{lbl:S}\rangle \xrightarrow{sk} \langle H, r, \texttt{S}\rangle}$$

$$\textbf{Continue-lbl}\ \frac{lbl = m}{\langle H, r, \texttt{continue lbl;}\ \rangle \xrightarrow{k(\texttt{LBL})} \langle H, \Box\rangle} \quad \textbf{Continue}\ \frac{}{\langle H, r, \texttt{continue;}\ \rangle \xrightarrow{k(\texttt{LOOP})} \langle H, \Box\rangle}$$

$$\textbf{Break-lbl}\ \frac{lbl = m}{\langle H, r, \texttt{break lbl;}\ \rangle \xrightarrow{k(\texttt{LBL})} \langle H, \Box\rangle} \quad \textbf{Break}\ \frac{}{\langle H, r, \texttt{break;}\ \rangle \xrightarrow{k(\texttt{LOOP})} \langle H, \Box\rangle}$$

$$\textbf{Throw}\ \frac{\langle H, r, e\rangle \rightarrow \langle H', r, va\rangle}{\langle H, r, \texttt{throw e}\rangle \xrightarrow{k(\texttt{TRY or FUNC})} \langle H', \Box\rangle}$$

$$\textbf{With}\ \frac{\langle H, r, e\rangle \rightarrow \langle H', r, r_o\rangle, \text{isObj}(r_o) = true,\ H' = H\left[r'_o \mapsto \left[\begin{array}{c} @scope \mapsto r, \\ @this \mapsto H(r)(@this), \\ \ldots \end{array}\right]\right]}{\langle H, r, \texttt{with(e) \{S\}}\rangle \xrightarrow{sk} \langle H', r'_o, \texttt{S}\rangle}$$

$$\textbf{Try}\ \frac{}{\langle H, r, \texttt{try\{S\}}\rangle \xrightarrow{sk} \langle H, r, \texttt{S}\rangle} \quad \textbf{Catch}\ \frac{}{\langle H, r, \texttt{catch(x)\{S\}}\rangle \xrightarrow{sk} \langle H, r, \texttt{S}\rangle}$$

$$\textbf{Finally}\ \frac{}{\langle H, r, \texttt{finally\{S\}}\rangle \xrightarrow{sk} \langle H, r, \texttt{S}\rangle}$$

$$\textbf{Output}\ \frac{\langle H, r, e\rangle \rightarrow \langle H', va\rangle}{\langle H, r, \texttt{output}_\ell\texttt{(e)}\rangle \xrightarrow{o_\ell(e,va)} \langle H', \Box\rangle} \quad \textbf{End}\ \frac{}{\langle H, r, \rtimes\rangle \xrightarrow{j} \langle H, \Box\rangle}$$

Table 4.5: VM Monitor Transitions.

$$\textbf{M-Skip}\ \frac{}{\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle \xrightarrow{sk} \langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle}$$

$$\textbf{M-Assign-Var}\ \frac{\Gamma,\Sigma,\Lambda,pc,H,r\vdash e:\tau,\quad \langle H,r,x\rangle \rhd_{Scope} r_x,\quad \Gamma'=\Gamma(r_x)[x\mapsto pc.\ell\sqcup\tau]}{\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle \xrightarrow{a_v(x,e)} \langle\langle\Gamma',\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle}$$

$$\textbf{M-Assign-Obj-Prop}\ \frac{\begin{array}{c}\Gamma,\Sigma,\Lambda,pc,H,r\vdash e:\tau,\quad \langle H,r,x\rangle \rhd_{Scope} r_x,\\ \Gamma'=\Gamma(r_x)[x\mapsto[y\mapsto pc.\ell\sqcup\tau]],\ \Sigma'=\Sigma(r_x)[x\mapsto pc.\ell\sqcup\tau\sqcup\Sigma(r_x)(x)]\end{array}}{\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle \xrightarrow{a_p(x.y,e)} \langle\langle\Gamma',\Sigma',\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle}$$

$$\textbf{M-Assign-Obj-Prop}\ \frac{\begin{array}{c}\Gamma,\Sigma,\Lambda,pc,H,r\vdash e_1:\tau_1,e_2:\tau_2,\langle H,r,x\rangle \rhd_{Scope} r_x,\\ \Gamma'=\Gamma(r_x)[x\mapsto[m\mapsto pc.\ell\sqcup\tau_1\sqcup\tau_2]],\Sigma'=\Sigma(r_x)[x\mapsto pc.\ell\sqcup\tau_1\sqcup\tau_2\sqcup\Sigma(r_x)(x)]\end{array}}{\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle \xrightarrow{a_i(x[m],e_1,e_2)} \langle\langle\Gamma',\Sigma',\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle}$$

$$\textbf{M-Assign-Array-Index}\ \frac{\begin{array}{c}\Gamma,\Sigma,\Lambda,pc,H,r\vdash e_1:\tau_1,e_2:\tau_2,\quad \langle H,r,x\rangle \rhd_{Scope} r_x,\\ \Gamma'=\Gamma(r_x)[x\mapsto[m\mapsto pc.\ell\sqcup\tau_1\sqcup\tau_2]],\ \Sigma'=\Sigma(r_x)[x\mapsto pc.\ell\sqcup\tau_1\sqcup\tau_2\sqcup\Sigma(r_x)(x)]\end{array}}{\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle \xrightarrow{a_i(x[m],e_1,e_2)} \langle\langle\Gamma',\Sigma',\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle}$$

$$\textbf{M-Branch}\ \frac{\Gamma,\Sigma,\Lambda,pc,H,r\vdash e:\tau,\quad \vec{x}=\texttt{collect}(S),\ \Delta'=\texttt{push}(\Delta,\langle\vec{x},\tau\sqcup pc.\ell\rangle)}{\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle \xrightarrow{b(e,S)} \langle\langle\Gamma,\Sigma,\Lambda,\Delta',pc\rangle,\langle H,r\rangle\rangle}$$

$$\textbf{M-Branch}\ \frac{\Gamma,\Sigma,\Lambda,pc,H,r\vdash e_1:\tau_1,...,e_n:\tau_n,\quad \vec{x}=\texttt{collect}(S),\ \Delta'=\texttt{push}(\Delta,\langle\vec{x},\tau_1\sqcup...\sqcup\tau_n\sqcup pc.\ell\rangle)}{\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle \xrightarrow{b(\vec{e},S)} \langle\langle\Gamma,\Sigma,\Lambda,\Delta',pc\rangle,\langle H,r\rangle\rangle}$$

$$\textbf{M-Join}\ \frac{\Delta,\langle\vec{x},\tau\rangle=\text{pop}(\Delta'),\ \Gamma',\Sigma'=\texttt{upgrade}(\vec{x},\tau\sqcup pc.\ell,\Gamma,\Sigma,H,r)}{\langle\langle\Gamma,\Sigma,\Lambda,\Delta',pc\rangle,\langle H,r\rangle\rangle \xrightarrow{j} \langle\langle\Gamma',\Sigma',\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle}$$

$$\textbf{M-Compensate}\ \frac{\Lambda'=\texttt{comp}_\Lambda(lbl,pc.\ell)}{\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle \xrightarrow{k(lbl)} \langle\langle\Gamma,\Sigma,\Lambda',\Delta,pc\rangle,\langle H,r\rangle\rangle}$$

$$\textbf{M-OutputFailStop}\ \frac{\Gamma,\Sigma,\Lambda,pc,H,r\vdash e:\tau,\ pc.\ell\sqcup\tau\sqsubseteq\ell}{\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle \xrightarrow{o_\ell(e,va)}_{o_\ell(va)} \langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle}$$

$$\textbf{M-OutputSuppress}\ \frac{\Gamma,\Sigma,\Lambda,pc,H,r\vdash e:\tau,\ \gamma=\begin{cases}o_\ell(va) & \text{if } (pc.\ell\sqcup\tau)\sqsubseteq\ell\\ \text{nothing} & \text{if } (pc.\ell\sqcup\tau)\not\sqsubseteq\ell\end{cases}}{\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle \xrightarrow{o_\ell(e,va)}_{\gamma} \langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle}$$

$$\textbf{M-OutputDefault}\ \frac{\Gamma,\Sigma,\Lambda,pc,H,r\vdash e:\tau,\ \gamma=\begin{cases}o_\ell(va) & \text{if } (pc.\ell\sqcup\tau)\sqsubseteq\ell\\ o_\ell(D) & \text{if } (pc.\ell\sqcup\tau)\not\sqsubseteq\ell\end{cases}, \text{where D is a default value}}{\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle \xrightarrow{o_\ell(e,va)}_{\gamma} \langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r\rangle\rangle}$$

Table 4.6: Collecting variables and functions from a statement S.

$\texttt{collect}(S)$ ::=		**case** S **:**
$\{S^*\}$	$\Rightarrow$	$\texttt{collect}(S_1) \cup ... \cup \texttt{collect}(S_n)$
$\texttt{if}(e)\ S_1\ \texttt{else}\ S_2$	$\Rightarrow$	$\texttt{collect}(S_1, S_2) \cup \texttt{colFunc}(e)$
$x = e$	$\Rightarrow$	$\{x\} \cup \texttt{colFunc}(e)$
$x.y = e$	$\Rightarrow$	$\{x\} \cup \texttt{colFunc}(e)$
$x[m] = e$	$\Rightarrow$	$\{x\} \cup \texttt{colFunc}(e)$
$\texttt{while}(e)\ S$	$\Rightarrow$	$\texttt{collect}(S) \cup \texttt{colFunc}(e)$
$\texttt{for}(e_1;\ e_2;\ e_3)\ S$	$\Rightarrow$	$\texttt{collect}(S) \cup \texttt{colFunc}(e_1, e_2, e_3)$
$\texttt{for}(x\ \texttt{in}\ e)\ S$	$\Rightarrow$	$\texttt{collect}(S) \cup \texttt{colFunc}(e)$
$\texttt{for}(\texttt{var}\ x = e_1\ \texttt{in}\ e_2)\ S$	$\Rightarrow$	$\texttt{collect}(S) \cup \texttt{colFunc}(e_1, e_2)$
$\texttt{return}\ \ e$	$\Rightarrow$	$\texttt{colFunc}(e)$
$\texttt{throw}\ \ e$	$\Rightarrow$	$\texttt{colFunc}(e)$
$\texttt{with}(e)\{S\}$	$\Rightarrow$	$\texttt{collect}(S) \cup \texttt{colFunc}(e)$
$\texttt{output}_\ell(e)$	$\Rightarrow$	$\texttt{colFunc}(e)$
otherwise	$\Rightarrow$	$\emptyset$

$\texttt{colFunc}(e)$ ::=		**case** e **:**
$f(\vec{e})$	$\Rightarrow$	$\{f\}$
new $g(\vec{e})$	$\Rightarrow$	$\{g\}$
$x.f()$	$\Rightarrow$	$\{x.f\}$
$x[g]()$	$\Rightarrow$	$\{x[g]\}$
new $x.f()$	$\Rightarrow$	$\{x.f\}$
new $x[g]()$	$\Rightarrow$	$\{x[g]\}$
otherwise	$\Rightarrow$	$\emptyset$

The desired security property in this case is *termination insensitive non-interference* (TINI) [47], [7]. Non-interference means that an attacker who can supply ℓ-level inputs and can observe ℓ-level outputs cannot observe outputs whose security levels are higher than ℓ. 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 $\sim^{\ell}_{\Gamma,\Sigma}$) *Given two heap memories H_1 and H_2 with the same domain, typing environment Γ, and a structure security environment Σ, we define that the two heaps are level-ℓ equivalent denoted $H_1 \sim^{\ell}_{\Gamma,\Sigma} H_2$, iff for all the variables $x \in dom(H_1)$ and $dom(H_2)$, $\Gamma(x) \sqsubseteq \ell$ and objects $o \in dom(H_1)$ and $dom(H_2)$, $\Sigma(o) \sqsubseteq \ell$.*

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

$$\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H_1, r, s\rangle\rangle \overset{\vec{w}}{\Longrightarrow} \langle\langle \Gamma', \Sigma', \Lambda', \Delta', pc\rangle, \langle H_1', r, s'\rangle\rangle$$

then $\exists\ H_2'$ and $\vec{w'}$ such that

$$\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H_2, r, s\rangle\rangle \overset{\vec{w'}}{\Longrightarrow} \langle\langle \Gamma', \Sigma', \Lambda', \Delta', pc\rangle, \langle H_2', r, s'\rangle\rangle$$

and either

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

2. or $\vec{w'}$ is a prefix of $\vec{w}$ and monitor configuration $\langle\Gamma', \Sigma', \Lambda', \Delta', pc\rangle, \langle H_2', r, s'\rangle\rangle$ is in ℓ'-level state where $\ell' \not\sqsubseteq \ell$ and outputs in level ℓ' 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 [47] 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. ■

4.3 Summary

In this chapter we introduced our hybrid flow-sensitive security monitor. We started by augmenting the JavaScript operational semantics with events that are only visible to the VM monitor, then we described the peculiarities of the VM security monitor. We also briefly described the attack model and the desired security property of the VM monitor. Lastly, we sketched the idea of the proof of termination insensitive non-interference (TINI) property of the VM monitor. The full proof is presented in the Appendix.

In the next chapter we will present our "IF-Transpiler" a source-to-source compiler that inlines the VM security monitor introduced in this chapter and we sketch the proof idea of the observational equivalence of the inlined monitor to the VM monitor.

Chapter 5

IF-Transpiler: Inlining of Hybrid Flow-Sensitive Security Monitor For JavaScript

In this chapter 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, $\mathcal{IF}\text{-}\mathcal{T}ranspiler(\mathcal{S}) = (\mathcal{I} \circ \mathcal{T})(\mathcal{S}) = \hat{\mathcal{S}}$, which means the composition of the inlining function $\mathcal{I}{:}\mathcal{S}' \mapsto \hat{\mathcal{S}}$ with transformation function $\mathcal{T}{:}\mathcal{S} \mapsto \mathcal{S}'$ on JavaScript Statement $\mathcal{S}$.

5.1 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 $\mathcal{T}: \mathcal{S} \mapsto \mathcal{S}'$ is defined in Table 5.1 and uses the string concatenation operator "+" to concatenate statements and a helper function $\mathcal{H}oist: \mathcal{S} \mapsto \mathcal{S}'$ 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 $\mathcal{H}oist$ 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 5.1 when the case is an anonymous function invocation.

5.2 Inlining Stage

The inlining stage operates in a similar fashion to the transformation stage. The input is a JavaScript statement $\mathcal{S}'$ and the output is an instrumented version $\hat{\mathcal{S}}$, formally, $\mathcal{I}: \mathcal{S}' \mapsto \hat{\mathcal{S}}$. The $\mathcal{S}'$ is expected to be a statement that has been converted through

Table 5.1: Transformation function $\mathcal{T}(\mathcal{S})$.

$\mathcal{T}(S)$ ::=		**case** S **:**
$x = e;$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(x = e;\ )$
$x.y = e;$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(x.y = e;\ )$
$e_1.x = e_2;$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + \mathcal{H}oist(\$t_1.x = e_2;\ )$
$e_1[e_2] = e_3;$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + \mathcal{H}oist(\$t_2 = e_2;\ ) + \mathcal{H}oist(\$t_1[\$t_2] = e_3;\ )$
$(\texttt{function}(\vec{x})\{S\})(\vec{e});$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$tmp = (\texttt{function}(\vec{x})\{S\})(\vec{e});\ )$
$e;$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$tmp = e;\ )$
$(e);$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$tmp = e;\ ) + (\$tmp);$
$\{S^*\}$	$\Rightarrow_{\mathcal{T}}$	$\{\mathcal{T}(S_1) + ... + \mathcal{T}(S_n)\}$
$\texttt{if}(e)\ S$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$tmp = e;\ ) + \texttt{if}(\$tmp)\ \{\mathcal{T}(S)\ \}\ \texttt{else}\ \{\}$
$\texttt{if}(e)\ S_1\ \texttt{else}\ S_2$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$tmp = e;\ ) + \texttt{if}(\$tmp)\ \{\mathcal{T}(S_1)\ \}\ \texttt{else}\ \{\mathcal{T}(S_2)\ \}$
$\texttt{while}(e)\ S$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$tmp = e;\ ) + \texttt{while}(\$tmp)\{\mathcal{T}(S)\}$
$\texttt{do}\{S\}\texttt{while}(e);$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$tmp = e;\ ) + \texttt{do}\{\mathcal{T}(S)\}\texttt{while}(\$tmp);$
$\texttt{for}(e_1;\ e_2;\ e_3)\ S$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + \mathcal{H}oist(\$t_2 = e_2;\ )+$ $\texttt{for}(\ ;\ \$t2;\ )\ \{\mathcal{T}(S) + \mathcal{H}oist(\$t_3 = e_3;\ ) + \mathcal{H}oist(\$t_2 = e_2;\ )\}$
$\texttt{for}(x\ \texttt{in}\ e)\ S$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$tmp = e;\ ) + \texttt{for}(x\ \texttt{in}\ \$tmp)\ \{\mathcal{T}(S)\}$
$\texttt{for}(\texttt{var}\ x = e_1\ \texttt{in}\ e_2)\ S$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\texttt{var}\ x = e_1;\ ) + \mathcal{H}oist(\$tmp = e_2;\ )+$ $\texttt{for}(x\ \texttt{in}\ \$tmp)\ \{\mathcal{T}(S)\}$
$\texttt{return}\ \ e;$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$tmp = e;\ ) + \texttt{return}\ \ \$tmp;$
$\texttt{throw}\ \ e;$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$tmp = e;\ ) + \texttt{throw}\ \ \$tmp;$
$\texttt{with}(e)\{S\}$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$tmp = e;\ ) + \texttt{with}(\$tmp)\{\mathcal{T}(S)\}$
$id : S$	$\Rightarrow_{\mathcal{T}}$	$id : \{\mathcal{T}(S)\}$
$\texttt{try}\{S\}$	$\Rightarrow_{\mathcal{T}}$	$\texttt{try}\{\mathcal{T}(S)\}$
$\texttt{catch}(x)\{S\}$	$\Rightarrow_{\mathcal{T}}$	$\texttt{catch}(x)\{\mathcal{T}(S)\}$
$\texttt{finally}\{S\}$	$\Rightarrow_{\mathcal{T}}$	$\texttt{finally}\{\mathcal{T}(S)\}$
$\texttt{output}_\ell(e)$	$\Rightarrow_{\mathcal{T}}$	$\mathcal{H}oist(\$tmp = e;\ ) + \texttt{output}_\ell(\$tmp)$
otherwise	$\Rightarrow_{\mathcal{T}}$	S

$\mathcal{H}oist(S)$ ::=		**case** S **:**
$x = y[e]$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$tmp = e;\ ) + x = y[\$tmp];$
$x = e_1.e_2$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + \mathcal{H}oist(x = \$t_1.e_2;\ )$
$x = e_1[e_2]$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + \mathcal{H}oist(\$t_2 = e_2;\ ) + x = \$t_1[\$t_2];$
$x = e_1[e_2](\vec{e})$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + \mathcal{H}oist(\$t_2 = e_2;\ ) + \mathcal{H}oist(x = \$t_1[\$t_2](\vec{e});\ )$
$x = e.y$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$tmp = e;\ ) + x = \$tmp.y;$
$x = e_1 ?\ e_2 : e_3$	$\Rightarrow_{\mathcal{H}}$	$\texttt{if}(e_1)\ \{x = e_1;\ \ \}\ \texttt{else}\ \{x = e_2;\ \ \}$
$x = y.f(\vec{e} = \{e_1, .., e_n\})$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + .. + \mathcal{H}oist(\$t_n = e_n;\ ) + x = y.f(\$t_1, .., \$t_n);$
$x = y[f](\vec{e} = \{e_1, .., e_n\})$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + .. + \mathcal{H}oist(\$t_n = e_n;\ ) + x = y[f](\$t_1, .., \$t_n);$
$x = f(\vec{e} = \{e_1, .., e_n\})$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + .. + \mathcal{H}oist(\$t_n = e_n;\ ) + x = f(\$t_1, .., \$t_n);$
$x = \texttt{new}\ \ y.f(\vec{e} = \{e_1, .., e_n\})$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + .. + \mathcal{H}oist(\$t_n = e_n;\ ) + x = \texttt{new}\ \ y.f(\$t_1, .., \$t_n);$
$x = \texttt{new}\ \ y[f](\vec{e} = \{e_1, .., e_n\})$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + .. + \mathcal{H}oist(\$t_n = e_n;\ ) + x = \texttt{new}\ \ y[f](\$t_1, .., \$t_n);$
$x = \texttt{new}\ \ f(\vec{e} = \{e_1, .., e_n\})$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + .. + \mathcal{H}oist(\$t_n = e_n;\ ) + x = \texttt{new}\ \ f(\$t_1, .., \$t_n);$
$x = [e_1, .., e_n]$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + .. + \mathcal{H}oist(\$t_n = e_n;\ ) + x = [\$t_1, .., \$t_n];$
$x = \{pn_1 : e_1, .., pn_n : e_n\}$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + .. + \mathcal{H}oist(\$t_n = e_n;\ ) + x = \{pn_1 : \$t_1, .., pn_n : \$t_n\};$
$x = (\texttt{function}(\vec{x})\{S\})(\vec{e});$	$\Rightarrow_{\mathcal{H}}$	$\$\lambda = \texttt{function}(\vec{x})\{\mathcal{T}(S)\};\ +\mathcal{H}oist(x = \$\lambda(\vec{e});\ )$
$x = e_1 \lozenge_{bin}\ e_2;$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$t_1 = e_1;\ ) + \mathcal{H}oist(\$t_2 = e_2;\ ) + x = \$t_1 \lozenge_{bin} \$t_2;$
$x = \lozenge_{un}\ e;$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$tmp = e;\ ) + x = \lozenge_{un} \$tmp;$
$x = e\ \lozenge_{po};$	$\Rightarrow_{\mathcal{H}}$	$\mathcal{H}oist(\$tmp = e;\ ) + x = \$tmp\ \lozenge_{un};$
otherwise	$\Rightarrow_{\mathcal{H}}$	S

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 Γ and Σ. Therefore, the inlining process will have the form $\Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in} \vdash \mathit{I}(\mathcal{S}') = \hat{\mathcal{S}}$. The inlining function make use of $sec_lvl(e)$ function defined in Table 5.2 to inline the security 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 Γ. For example, if the current scope is the global scope the $\$\$CS$ symbol will be replaced by Γ`['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 Γ`['Global']['foo']` etc. The inlining function also makes use of a $\$scope()$ [1] and a $\$proto()$ functions. The two functions correspond to the $\triangleright_{Scope}$ and $\triangleright_{Proto}$ relations defined previously in subseciotn 3.4. The $\$scope(\$\$CS, var)$ function takes two arguments, the first is the current scope object $\$\$CS$ and the second argument is the variable var to locate. The $\$proto(obj, prop, \$\$CS)$ 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 5.3 to 5.5 . It is important to note that the

[1] 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 5.2: Security level of expression e.

$sec_lvl(e)$ ::=		**case** e **:**
x	$\Rightarrow$	$\$scope(\$\$CS,'x')['x'];$
$this$	$\Rightarrow$	$\$\$CS.this$
pv	$\Rightarrow$	$\bot_\ell$
$e_1 \Diamond_{bin} e_2$	$\Rightarrow$	$sec_lvl(e_1) \sqcup sec_lvl(e_2)$
$\Diamond_{un} x$	$\Rightarrow$	$sec_lvl(x)$
$x \Diamond_{po}$	$\Rightarrow$	$sec_lvl(x)$
$y[z] \mid y.z$	$\Rightarrow$	$\$prop('y','z',\$\$CS)$

inlining function introduces the ✓ and □ statements, which are only used for our proof of correctness and do not appear in the instrumented code.

5.3 Soundness of the Inlined Security Monitor

Similar to [7], 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* $\Gamma_{in} = \Gamma \uplus \Sigma, \Lambda_{in} = \Lambda, \Delta_{in} = \Delta,$ *and* $pc_{in} = pc$.

The instrumented program should start with configuration $\langle H_0 \uplus mst_0, r_0, \hat{\mathcal{S}} \rangle$, where H_0 is the initial heap memory, mst_0 is the initial monitor state, and the initial scope reference $r_0 = \#Global$ is pointing to the address of the global object. The initial monitor state is defined as $mst_0 = (\Gamma_0 = \{'Global' : \{scope : \texttt{null}, \Sigma : \bot_\mathcal{L}\}\}, \Lambda_0 = [\{\ell : \bot_\mathcal{L}, id :' Global'\}], \Delta_0 = [], pc_0 = \Lambda[0])$, which means the only information that both Γ and Λ initially contain is about the global scope, Δ initially is empty, and pc

Table 5.3: Inlining function *I(S) (Part A)*.

$\Gamma, \Lambda, \Delta, pc \vdash I(S') ::=$		**case** S' **:**
`Skip;`	$\Rightarrow_I$	`Skip;` $+ \checkmark$
`function foo(`$\vec{x}$`)`$\{S\}$; | `foo = function(`$\vec{x}$`)`$\{S\}$;	$\Rightarrow_I$	`function foo(`$\vec{x}$`)`$\{I(S)\}$; $+\$\$CS['foo'] =$ $\{x_1 : \$pc.\ell, ..., x_n = \$pc.\ell, \$fscope : \$pc.\ell, prototype : \{\Sigma : \$pc.\ell\}, \Sigma : \$pc.\ell\};\ \checkmark$
$obj = \{pn_1 : e_1, .., pn_n : e_n\};$	$\Rightarrow_I$	$obj = \{pn_1 : e_1, .., pn_n : e_n\};\ +\$\$CS['obj'] =$ $\{pn1 : sec_lvl(e_1) \sqcup \$pc.\ell, ..., pn_n : sec_lvl(e_n) \sqcup \$pc.\ell, __proto__ : \texttt{null},$ $\Sigma : \$pc.\ell \sqcup sec_lvl(e_1) ... \sqcup sec_lvl(e_n)\};\ \checkmark$
$x = [e_0, e_1, ..., e_{n-1}]$	$\Rightarrow_I$	$x = [e_0, e_1, ..., e_{n-1}];\ +\$\$CS['x'] =$ $\{0 : sec_lvl(e_0) \sqcup \$pc.\ell, ..., n-1 : sec_lvl(e_{n-1}) \sqcup \$pc.\ell, __proto__ : \texttt{null},$ $\Sigma : \$pc.\ell \sqcup sec_lvl(e_1) ... \sqcup sec_lvl(e_n)\};\ \checkmark$
$x = f(\vec{e});$	$\Rightarrow_I$	$\$rf = \$scope(\$\$CS, 'f')['f'];\ +\$rf.scope = \$\$CS;$ $+\$rf.this = \$\Gamma['Global'];\ +\$rf['x'_i] = sec_lvl(e_i) \sqcup \$pc.\ell;\ +$ $\$\Lambda.push(\{\ell : \$rf.\$fscope \sqcup \$pc.\ell \sqcup \$rf.\Sigma, id : '\texttt{FUNC}'\});\ +x = f(\vec{e});$ $+\$scope(\$\$CS, 'x')['x'] = \$\Lambda.pop().\ell;\ +$ $(isObj(\$scope(\$\$CS, 'x')['x']))?\$scope(\$\$CS, 'x')['x'].\Sigma =$ $\$scope(\$\$CS, 'x')['x'].\Sigma \sqcup \$pc.\ell$ $: \$scope(\$\$CS, 'x')['x'] = \$scope(\$\$CS, 'x')['x'] \sqcup \$pc.\ell;\ \checkmark$
$x = \texttt{new}\ \ f(\vec{e});$	$\Rightarrow_I$	$\$rf = \$scope(\$\$CS, 'f')['f'];\ +\$rf.scope = \$\$CS;\ +$ $\$rf.this = \$scope(\$\$CS, 'x')['x'] = \{\Sigma : \$pc.\ell, __proto__ : \$rf.prototype\}+$ $\$rf['x'_i] = sec_levl(e_i);\ +\$rf.InvokedAsConstr = true;\ +$ $\$\Lambda.push(\{\ell : \$rf.\$fscope \sqcup \$pc.\ell \sqcup \$rf.\Sigma, id : '\texttt{FUNC}'\});\ +$ $x = \texttt{new}\ \ f(\vec{e});\ +\$scope(\$\$CS, 'x')['x'] = \$\Lambda.pop().\ell;\ +$ $(isObj(\$scope(\$\$CS, 'x')['x']))?\$scope(\$\$CS, 'x')['x'].\Sigma =$ $\$scope(\$\$CS, 'x')['x'].\Sigma \sqcup \$pc.\ell$ $: \$scope(\$\$CS, 'x')['x'] = \$scope(\$\$CS, 'x')['x'] \sqcup \$pc.\ell;\ \checkmark$
$x = y.f(\vec{e})$ | $x = y[f](\vec{e})$	$\Rightarrow_I$	$\$rf = \$prop('y', 'f', \$\$CS);\ +\$rf.scope = \$\$CS;\ +$ $\$rf.this = \$scope(\$\$CS, 'y')['y'];\ +\$rf['x'_i] = sec_lvl(e_i);\ +$ $\$\Lambda.push(\{\ell : \$rf.\$fscope \sqcup \$pc.\ell \sqcup \$rf.\Sigma, id : '\texttt{FUNC}'\});\ +$ $x = y.f(\vec{e});\ +\$scope(\$\$CS, 'x')['x'] = \$\Lambda.pop().\ell;\ +$ $(isObj(\$scope(\$\$CS, 'x')['x']))?\$scope(\$\$CS, 'x')['x'].\Sigma =$ $\$scope(\$\$CS, 'x')['x'].\Sigma \sqcup \$pc.\ell$ $: \$scope(\$\$CS, 'x')['x'] = \$scope(\$\$CS, 'x')['x'] \sqcup \$pc.\ell;\ \checkmark$
$x = \texttt{new}\ \ y.f(\vec{e});$ | $x = \texttt{new}\ \ y[f](\vec{e});$	$\Rightarrow_I$	$\$rf = \$prop('y', 'f', \$\$CS);\ +$ $\$rf.this = \$scope(\$\$CS, 'x')['x'] = \{\Sigma : \$pc.\ell, __proto__ : \$rf.prototype\}+$ $\$rf['x'_i] = sec_lvl(e_i);\ +\$rf.InvokedAsConstr = true;\ +$ $\$\Lambda.push(\{\ell : \$rf.\$fscope \sqcup \$pc.\ell \sqcup \$rf.\Sigma, id : '\texttt{FUNC}'\});\ +$ $x = \texttt{new}\ \ y.f(\vec{e});\ +\$scope(\$\$CS, 'x')['x'] = \$\Lambda.pop().\ell;\ +$ $(isObj(\$scope(\$\$CS, 'x')['x']))?\$scope(\$\$CS, 'x')['x'].\Sigma =$ $\$scope(\$\$CS, 'x')['x'].\Sigma \sqcup \$pc.\ell$ $: \$scope(\$\$CS, 'x')['x'] = \$scope(\$\$CS, 'x')['x'] \sqcup \$pc.\ell;\ \checkmark$
$x = e;$	$\Rightarrow_I$	$x = e;\ +\$scope(\$\$CS, 'x')['x'] = sec_lvl(e);\ +$ $(isObj(\$scope(\$\$CS, 'x')['x']))?\$scope(\$\$CS, 'x')['x'].\Sigma =$ $\$scope(\$\$CS, 'x')['x'].\Sigma \sqcup \$pc.\ell$ $: \$scope(\$\$CS, 'x')['x'] = \$scope(\$\$CS, 'x')['x'] \sqcup \$pc.\ell;\ \checkmark$

Table 5.4: Inlining function *I(S) (Part B)*.

$x.y = e;$ $\Rightarrow_I$

$x.y = e;\ +\$scope(\$\$CS,'x')['x']['y'] = sec_lvl(e);\ +$
$(isObj(\$prop('x','y',\$\$CS))?$
$\$prop('x','y',\$\$CS).\Sigma = \$prop('x','y',\$\$CS).\Sigma \sqcup \$pc.\ell +$
$: \$scope(\$\$CS,'x')['x']['y'] = \$prop('x','y',\$\$CS) \sqcup \$pc.\ell;$
$\$scope(\$\$CS,'x')['x'].\Sigma = sec_lvl(x.y) \sqcup sec_lvl(x);\ \checkmark$

$x[y] = e;$ $\Rightarrow_I$

$x[y] = e;\ +\$scope(\$\$CS,'x')['x'][y] = sec_lvl(e);\ +$
$\$tmp = (isObj(sec_lvl(y))?sec_lvl(y).\Sigma : sec_lvl(y);\ +$
$(isObj(\$prop('x',y,\$\$CS))?\$prop('x',y,\$\$CS).\Sigma =$
$\$prop('x',y,\$\$CS).\Sigma \sqcup \$tmp \sqcup \$pc.\ell +$
$: \$scope(\$\$CS,'x')['x'][y] = \$prop('x',y,\$\$CS) \sqcup \$tmp \sqcup \$pc.\ell;$
$\$scope(\$\$CS,'x')['x'].\Sigma = sec_lvl(x[y]) \sqcup sec_lvl(x);\ \checkmark$

`return;` $\Rightarrow_I$

$\$old_pc = \$pc;\ +\texttt{while}(\$pc.id\ \texttt{!==}'\texttt{FUNC}')\ \$\Lambda.pop();\ +$
$\texttt{if}(\$\$CS.InvokedAsContr)\{+$
$\quad \$\$CS.this.\Sigma = \$\$CS.this.\Sigma \sqcup \$old_pc.\ell;\ +$
$\quad \$\Lambda[\$\Lambda.len - 1] = \{\ell : \$\$CS.this;\ \};\ +$
$\}\ \texttt{else}\ \{$
$\quad \$\Lambda[\$\Lambda.len - 1] = \{\ell : \$old_pc.\ell\};$
$\}+$
$\texttt{return;}\checkmark$
$\texttt{RET}\ \{'lbl' :' \texttt{FUNC}'\}$

`return x;` $\Rightarrow_I$

$\$old_pc = \$pc;\ +\texttt{while}(\$pc.id\ \texttt{!==}'\texttt{FUNC}')\ \$\Lambda.pop();\ +$
$\$rx = \$scope(\$\$CS,'x')['x'];\ +$
$\texttt{if}(isObj(\$rx))\{+$
$\quad \$rx.\Sigma = \$rx.\Sigma \sqcup \$old_pc.\ell;\ +$
$\quad \$\Lambda[\$\Lambda.len - 1] = \{\ell : \$rx\};\ +$
$\}\ \texttt{else if}(\$\$CS.InvokedAsContr)\{+$
$\quad \$\$CS.this.\Sigma = \$\$CS.this.\Sigma \sqcup \$old_pc.\ell;\ +$
$\quad \$\Lambda[\$\Lambda.len - 1] = \{\ell : \$\$CS.this;\ \};\ +$
$\}\ \texttt{else}\ \{\ \$\Lambda[\$\Lambda.len - 1] = \{\ell : \$rx \sqcup \$old_pc.\ell\};\ \}+$
$\texttt{return x;}\checkmark$
$\texttt{RET}\ \{'lbl' :' \texttt{FUNC}'\}$

`break; | continue;` $\Rightarrow_I$

$\$old_pc = \$pc;\ +\texttt{while}(\$pc.id\ \texttt{!==}'\texttt{LOOP}')\ \$\Lambda.pop();\ +$
$\$\Lambda[\$\Lambda.len - 2].\ell = \$\Lambda[\$\Lambda.len - 2].\ell \sqcup \$old_pc.\ell;\ +$
$\texttt{break;}\ |\ \texttt{continue;}\checkmark$
$\texttt{RET}\ \{'lbl' : \texttt{'LOOP'}\}$

`break 'label';`
`| continue 'label';` $\Rightarrow_I$

same as previous case $+ \checkmark$
$\texttt{RET}\ \{'lbl' : \texttt{'label'}\}$

$\texttt{if}(e)\ \{S_1\}\ \texttt{else}\ \{S_2\}$ $\Rightarrow_I$

$\$xs_1 = \$Collect(S_1);\ +\$xs_2 = \$Collect(S_2);\ +$
$\$\Lambda.push(\{\ell : \$pc.\ell \sqcup sec_lvl(e), id :' \texttt{IF}'\});\ +$
$\texttt{if}(e)\ \{\$upgrade(\$xs_2, \$pc.\ell);\ +\checkmark + \texttt{var}\ \$should_comp = I(S_1);\ \}\ \texttt{else} +$
$\{\$upgrade(\$xs_1, \$pc.\ell);\ +\checkmark + \texttt{var}\ \$should_comp = I(S_2);\ \ \}+$
$\texttt{if}(\$should_comp)\{\$Comp_\Lambda(\$should_comp.lbl, \$pc.\ell);\ \} + \$\Lambda.pop();\ +\checkmark$

Table 5.5: Inlining function *I(S) (Part C)*.

$\texttt{while}(e)\{S\}$	$\Rightarrow_I$	$\$\Lambda.push(\{\ell : \$pc.\ell \sqcup sec_lvl(e), lbl :' \texttt{LOOP}'\});\ +$ $\$\Delta.push(\{xs : \$Collect(S), \ell : \$pc.\ell\});\ +$ $\texttt{while}(e)\{+$ $\checkmark + \texttt{var}\ \$should_comp = I(S);\ +\}$ $\$tmp = \$\Delta.pop();\ +$ $\$upgrade(\$tmp.xs, \$tmp.\ell);\ +$ $\texttt{if}(\$should_comp)\{\$Comp_\Lambda(\$should_comp.lbl, \$pc.\ell);\ \} + \$\Lambda.pop();\ +\checkmark$
$\texttt{for}(;\ e;\ )\{S\}$	$\Rightarrow_I$	$\$\Lambda.push(\{\ell : \$pc.\ell \sqcup sec_lvl(e), lbl :' \texttt{LOOP}'\});\ +\$\Delta.push(\{xs : \$Collect(S), \ell : \$pc.\ell\});\ +$ $\texttt{for}(;\ e;\ )\{+$ $\checkmark + \texttt{var}\ \$should_comp = I(S);\ \}+$ $\$tmp = \$\Delta.pop();\ +$ $\$upgrade(\$tmp.xs, \$tmp.\ell);\ +$ $\texttt{if}(\$should_comp)\{\$Comp_\Lambda(\$should_comp.lbl, \$pc.\ell);\ \} + \$\Lambda.pop();\ +\checkmark$
$\texttt{for}(x\ \texttt{in}\ e)\{S\}$	$\Rightarrow_I$	$\$\Lambda.push(\{\ell : \$pc.\ell \sqcup sec_lvl(e), lbl :' \texttt{LOOP}'\});\ +\$\Delta.push(\{xs : \$Collect(S), \ell : \$pc.\ell\});\ +$ $\texttt{for}(x\ \texttt{in}\ e)\{+$ $\checkmark + \texttt{var}\ \$should_comp = I(S);\ \}+$ $\$tmp = \$\Delta.pop();\ +$ $\$upgrade(\$tmp.xs, \$tmp.\ell);\ +$ $\texttt{if}(\$should_comp)\{\$Comp_\Lambda(\$should_comp.lbl, \$pc.\ell);\ \} + \$\Lambda.pop();\ +\checkmark$
S^*	$\Rightarrow_I$	$\texttt{var}\ \$tmp, \$ret;\ +$ $\$tmp = I_1(S_1);\ +\texttt{if}(!\$ret)\ \$ret = \$tmp;\ +$ $\vdots$ $\$tmp = I_n(S_n);\ +\texttt{if}(!\$ret)\ \$ret = \$tmp;$ $RET\ \$ret$
$id : S$	$\Rightarrow_I$	$\$\Lambda.push(\{\ell : \$pc.\ell, lbl :' \texttt{id}'\});\ +id : \{I(S)\} + \$\Lambda.pop();\ +\checkmark$
$\texttt{throw e;}$	$\Rightarrow_I$	$\$old_pc = \$pc;\ +\texttt{while}(\$pc.id\ \texttt{!== 'FUNC'}\&\&\ \$pc.id\ \texttt{!== 'TRY'})\{\$\Lambda.pop();\ \}+$ $\$\Lambda[\$\Lambda.len - 1] = \{\ell : \$old_pc.\ell \sqcup sec_lvl(e)\};\ +\texttt{throw e;} + \checkmark$ $\texttt{RET}\ \{'lbl' : \texttt{'FUNC'}\ \mid\ \texttt{'TRY'}\}$
$\texttt{with(x)}\{S\}$	$\Rightarrow_I$	$\$\Lambda.push(\{\ell : \$pc.\ell \sqcup sec_lvl(x), lbl : \texttt{'WITH'}\});\ +\texttt{with(x)}\{$ $\$ro = \$scope('x', \$\$CS)['x'];\ +\$ro.scope = \$\$CS;\ +$ $\$ro.this = \$\$CS.this;\ +\$\$CS = \$ro;\ +$ $I(S);\ +$ $\$\$CS = \$ro.scope;\ \}+$ $\$\Lambda.pop();\ +\checkmark$
$\texttt{try}\{S\}$	$\Rightarrow_I$	$\texttt{try}\{\$\Lambda.push(\{\ell : \$pc.\ell, lbl : \texttt{'TRY'}\});\ +I(S);\ +\$\Lambda.pop();\ \} + \checkmark$
$\texttt{catch(x)}\{S\}$	$\Rightarrow_I$	$\texttt{catch(x)}\{I(S);\ +\$\Lambda.pop();\ \} + \checkmark$
$\texttt{finally}\{S\}$	$\Rightarrow_I$	$\texttt{finally}\{I(S);\ \} + \checkmark$
$\checkmark$	$\Rightarrow_I$	$\square$
$\square$	$\Rightarrow_I$	$\square$
$\texttt{output}_\ell(e)$	$\Rightarrow_I$	$\textbf{case}\ \$Policy :$
'OutputFailStop'	:	$if(sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell)\ \{\texttt{output}_\ell(e);\ +\checkmark\}\ \texttt{else}\ \$diverge\$;$
'OutputSuppress'	:	$if(sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell)\ \texttt{output}_\ell(e);\ \texttt{else}\ \text{Skip};\ +\checkmark$
'OutputDefault'	:	$if(sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell)\ \texttt{output}_\ell(e);\ \texttt{else}\ \texttt{output}_\ell(D);\ +\checkmark$
Otherwise	$\Rightarrow_I$	S'

is pointing to the top-most element of security context stack Λ 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* $\langle\langle H_1, r_1, \mathcal{S}\rangle, \langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle\rangle \sim \langle H_2 \uplus mst, r_2, \hat{\mathcal{S}}\rangle$ *if and only if* $H_1 = H_2, r_1 = r_2$, *and* $\exists\ \Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in}$ *such that* $\Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in} \vdash \hat{\mathcal{S}} = (\mathcal{I} \circ \mathcal{T})(\mathcal{S})$ *and* $\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle \cong ((\Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in}) = mst)$.

Similar to [7], we define $\stackrel{\beta}{\Longrightarrow}$ for transitions of statement configurations by

$$\langle H \uplus mst, r, \hat{\mathcal{S}}\rangle \stackrel{\beta}{\Longrightarrow} \langle H' \uplus mst', r, \hat{\mathcal{S}}'\rangle$$

to mean

$$\langle H \uplus mst, r, \hat{\mathcal{S}}\rangle \rightarrow^* \stackrel{\beta}{\rightarrow} \rightarrow^* \langle H' \uplus mst', r, \hat{\mathcal{S}}'\rangle$$

and we write $\stackrel{\vec{\beta}}{\Longrightarrow}$ 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 4.3 and 4.4. Among the possible events is $o_\ell(e, va)$ 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* $\vec{\beta}\ |_{out}\ \vec{w}$ *to hold if and only if* $\vec{w}$ *is obtained by filtering non-output events and changing every* $o_\ell(e, va)$ *to* $o_\ell(va)$.

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

(a) For all $\mathcal{S}', \Gamma', \Sigma', \Lambda', \Delta', pc'$ *if* $\langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \mathcal{S}\rangle\rangle \stackrel{\vec{w}}{\Longrightarrow} \langle\langle\Gamma', \Sigma', \Lambda', \Delta', pc\rangle, \langle H', r, \mathcal{S}'\rangle\rangle$ *then there is* $\vec{\beta}$ *such that* $\vec{\beta}\ |_{out}\ \vec{w}$, $\langle H \uplus mst, r, \hat{\mathcal{S}}\rangle \stackrel{\vec{\beta}}{\Longrightarrow} \langle H' \uplus mst', r, \hat{\mathcal{S}}'\rangle$, *and*

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

(b) For all $\hat{\mathcal{S}}', mst'$ *if* $\langle H \uplus mst, r, \hat{\mathcal{S}}\rangle \stackrel{\vec{\beta}}{\Longrightarrow} \stackrel{t}{\longrightarrow} \langle H' \uplus mst', r, \hat{\mathcal{S}}'\rangle$ *then there is* $\vec{w}$ *such that* $\vec{\beta} \mid_{out} \vec{w}$, $\langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \mathcal{S}\rangle\rangle \stackrel{\vec{w}}{\Longrightarrow} \langle\langle\Gamma', \Sigma', \Lambda', \Delta', pc\rangle, \langle H', r, \mathcal{S}'\rangle\rangle$, *and* $\langle\langle H', r, \mathcal{S}'\rangle, \langle\Gamma', \Sigma', \Lambda', \Delta', pc'\rangle\rangle \sim \langle H' \uplus mst', r, \hat{\mathcal{S}}'\rangle$.

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 $\langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \mathcal{S}\rangle\rangle$ which is assumed to be coupled with the inlined monitor configuration $\langle H \uplus mst, r, \hat{\mathcal{S}}\rangle$ as stated in Theorem 2. For every step performed by the VM monitor, we go by cases on statement $\mathcal{S}$ and compare each step of the VM monitor with one or more steps performed by the inlined monitor on the instrumented statement $\hat{\mathcal{S}}$ finishing with a $t-$transition. In order to prove that the resulting configuration is coupled $\langle\langle\Gamma', \Sigma', \Lambda', \Delta', pc'\rangle, \langle H', r, \mathcal{S}'\rangle\rangle \sim \langle H' \uplus mst', r, \hat{\mathcal{S}}'\rangle$, we have to establish monitor state coupling: $\langle\Gamma', \Sigma', \Lambda', \Delta', pc'\rangle \cong (mst' = (\Gamma'_{in}, \Lambda'_{in}, \Delta'_{in}, pc'_{in}))$. We sketch the case where $\mathcal{S}$ = `x=e;` as an illustration. According to *IF-Transpiler*(`x=e;`) $\Rightarrow \hat{\mathcal{S}} = (x = e;\ +\$rx = \$scope(\$\$CS,'x')['x'];\ +\$rx = sec_lvl(e);\ +(isObj(\$rx))?\$rx.\Sigma = \$rx.\Sigma \sqcup \$pc.\ell : \$rx = \$rx \sqcup \$pc.\ell;\ +\checkmark)$.

Now we show the steps of the VM monitor execution, assuming τ 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):

$$\langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{x=e;}\rangle\rangle$$

$\xrightarrow{a_v(x,e)}$

$$\langle\langle\Gamma(r_x)[x \mapsto pc.\ell \sqcup \tau], \Sigma, \Lambda, \Delta, pc\rangle, \langle H(r_x)[x \mapsto va], r, \square\rangle\rangle \text{ where } \langle H, r, x\rangle \rhd_{Scope} r_x$$

$=$

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

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

$$\langle H \uplus mst, r, "x = e;\ +\$rx = \$scope(\$\$CS,'x')['x'];\ +$$

$$\$rx = sec_lvl(e);\ +\$rx = \$rx \sqcup \$pc.\ell;\ +\checkmark" \rangle$$

$\xrightarrow{\beta_1}$ where $\beta_1 = a_v(x, e)$

$$\langle H(r_x)[x \mapsto va] \uplus mst, r, "\$rx = \$scope(\$\$CS,'x')['x'];\ +$$

$$\$rx = sec_lvl(e);\ +\$rx = \$rx \sqcup \$pc.\ell;\ +\checkmark" \rangle$$

$\xrightarrow{\beta_2}$ where $\beta_2 = a_v(\$rx, \Gamma(r_x)[x])$

$$\langle H(r_x)[x \mapsto va] \uplus mst, r, "\$rx = sec_lvl(e);\ +\$rx = \$rx \sqcup \$pc.\ell;\ +\checkmark" \rangle$$

$\xrightarrow{\beta_3}$ where $\beta_3 = a_v(\Gamma(r_x)[x], \tau)$ and $\tau = sec_lvl(e)$

$$\langle H(r_x)[x \mapsto va] \uplus mst = (\Gamma(r_x)[x \mapsto \tau], \Lambda, \Delta, pc), r, "\$rx = \$rx \sqcup \$pc.\ell;\ +\checkmark" \rangle$$

$\xrightarrow{\beta_4}$ where $\beta_4 = a_v(\Gamma(r_x)[x], \Gamma(r_x)[x] \sqcup \$pc.\ell)$

$$\langle H(r_x)[x \mapsto va] \uplus mst = (\Gamma(r_x)[x \mapsto \tau \sqcup pc.\ell], \Lambda, \Delta, pc), r, "\checkmark" \rangle$$

$\xrightarrow{t}$

$$\langle H(r_x)[x \mapsto va] \uplus mst = (\Gamma(r_x)[x \mapsto \tau \sqcup pc.\ell], \Lambda, \Delta, pc), r, \Box \rangle$$

$=$

$$\langle H' \uplus mst', r, \hat{\mathcal{S}}' \rangle \qquad \text{which define } H', mst', \hat{\mathcal{S}}'$$

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

For clause **(b)** of Theorem 2, the proof goes by the induction on the structure of statement $\mathcal{S}$, the rules outlined in Tables 5.3 to 5.5, and the events that get generated by the language VM as outlined in the operational semantics of the language in Tables 4.3 and 4.4. Based on the rules in Tables 5.3 to 5.5, every instrumented statement $\hat{\mathcal{S}}'$ encompasses the semantics of the original non-instrumented statement $\mathcal{S}$ plus one or

more additional statements, formally, $\hat{\mathcal{S}}' = \mathcal{S}^* + \mathcal{S} + \mathcal{S}^* + \checkmark$, where $\mathcal{S}^*$ means zero or more statements and $+$ is concatenation operator. Which leads us to the following; When $\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \mathcal{S}\rangle\rangle \xRightarrow{\vec{\alpha}} \langle\langle \Gamma', \Sigma', \Lambda', \Delta', pc\rangle, \langle H', r, \mathcal{S}'\rangle\rangle$ and $\langle H \uplus mst, r, \hat{\mathcal{S}}\rangle \xRightarrow{\vec{\beta}}$ $\xrightarrow{t} \langle H' \uplus mst', r, \hat{\mathcal{S}}'\rangle$ then $\vec{\beta}$ will contain the same events generated by $\vec{\alpha}$ and if $\vec{\alpha} \mid_{out} \vec{w_1}$ and $\vec{\beta} \mid_{out} \vec{w_2}$ then $\vec{w_1} = \vec{w_2}$ or both are empty when all the events that got generated are internal events ϵ. 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 $\mid_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$, we get $\vec{\alpha} \mid_{out} \emptyset$ and $\vec{\beta} \mid_{out} \emptyset$, trivially $\emptyset = \emptyset$ since all the events that got generated by the two monitors are internal events ϵ as outlined in Table 4.2 .

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. [2] Every run of the instrumented program satisfies termination-insensitive non-interference (TINI) with respect to the chosen policy. ■

5.4 Summary

In this chapter we presented the inlining of the VM security monitor introduced in the previous chapter. Precisely, we introduced our IF-Transpiler, a source-to-source compiler that instruments vanilla JavaScript code with the security monitor in two stages: Transformation and Inlining. The final output of the transpiler is an instrumented version of the original code that is capable of running on any JavaScript engine. In

[2] Corollary 3 corresponds to Theorem 3.

that regard, in the next chapter, we present our implementation of the IF-Transpiler and evaluate its performance with respect to un-instrumented code and to other implementations in the literature.

Chapter 6

Implementation and Experimental Evaluation

In this chapter we present the implementation of our IF-Transpiler and evaluate its performance with respect to un-instrumented code and other implementations in the literature. We conduct different experiments to study and assess the efficiency and effectiveness of the IF-Transpiler.

6.1 Implementation

In order to implement our framework we used *Esprima* project [1] in conjunction with three tools from the ECMAScript Tooling project [2]. *Esprima* stands for ECMAScript parsing infrastructure for multipurpose analysis. It is a high performance, standard-compliant JavaScript parser written in JavaScript. Esprima parser supports not only ECMAScript 5, but it supports also the most recent version of the standard ECMAScript 6 released in the summer of 2015, which makes our implementation ready

[1] http://esprima.org
[2] https://github.com/estools

to get extended to support ECMAScript 6 in the future.

The Esprima parser generates an Abstract Syntax Tree (AST) format that is standardized by the EStree project [3], which makes the output of the parser usable with other ECMAScript tools that use the same format. As a matter of fact, the Esprima parser is used as a component in many projects that perform static analysis, dynamic tracing, and code transformation [4].

We used the Esprima parser with three tools from the ECMAScript Tooling project, namely, *estraverse*, *esrecurse*, and *escodegen*. As the names of the tools may suggest, the *estraverse* tool provides an iterative ECMAScript traversing functionality, the *esrecurse* tool provides a recursive ECMAScript traversing functionality, and the *escodegen* tool is a code generator that transforms the Abstract Syntax Tree (AST) to text based JavaScript code.

We implemented our IF-Transpiler in a pipeline fashion that is very similar to the one previously described in chapter 5 and it is outlined in Figure 6.1. The Esprima parser is used to generate the Abstract Syntax Tree (AST) of the JavaScript code, then the *estraverse* tool is used to traverse the AST and transform every and each node as previously described in section 5.1. The output of the transformation stage is an AST that is passed to the inlining stage. We modified the *esrecurse* tool to suit the needs of the inlining stage, precisely, we modified the tool to maintain scope information when recursively visiting the nodes of the AST tree and we made every tree node points to its parent node. These two modifications are necessary to inject the additional statements that implement the information flow tracking. The inlining stage instruments the AST with the necessary statements that track the information flow as described in section 5.2. Finally, the output of the inlining stage is passed to the code generation stage where the AST is transformed into source code and get saved

[3]https://github.com/estree/estree
[4]http://esprima.org/demo/index.html

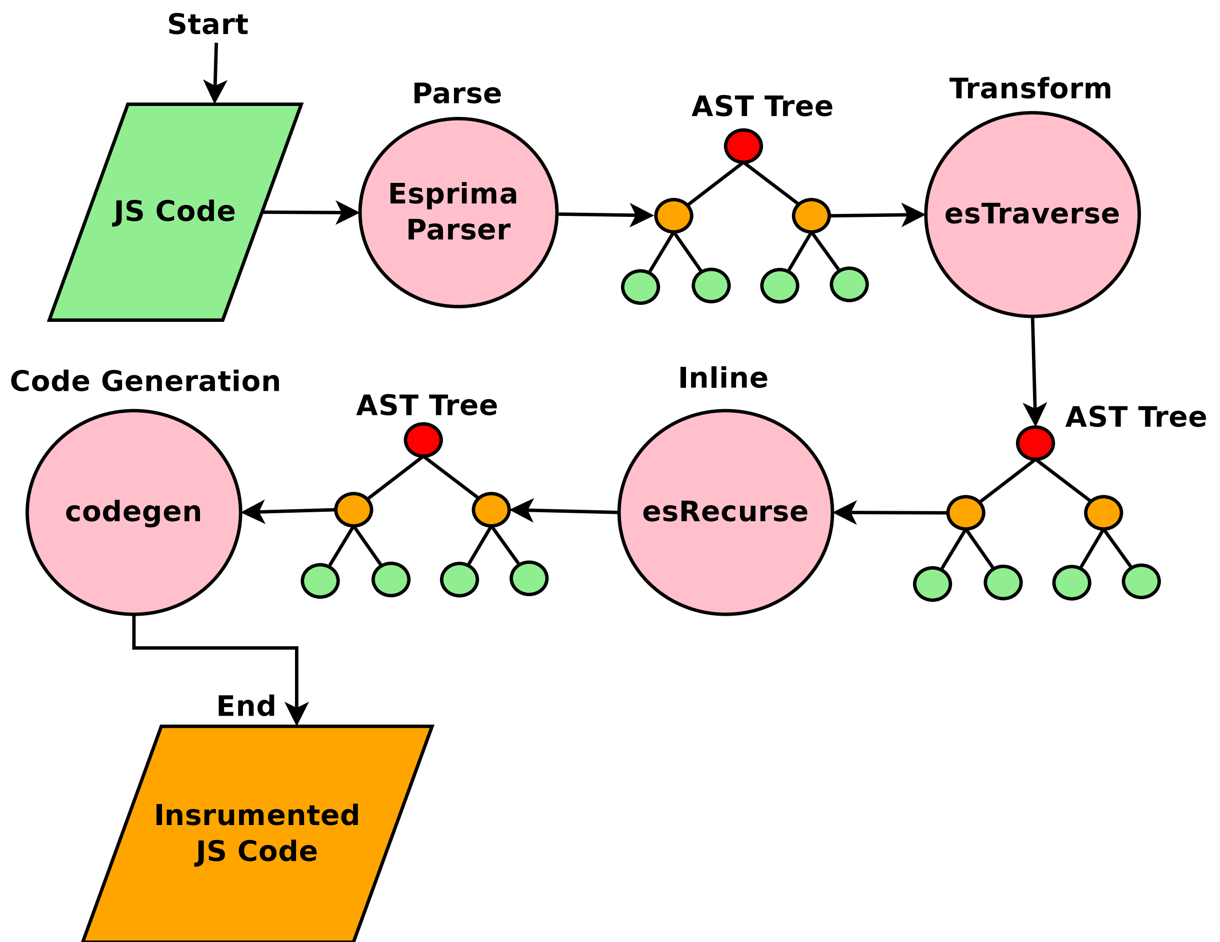

Figure 6.1: IF-Transpiler Pipeline Architecture.

to a file. Our IF-Transpiler depends on some functionality provided by NodeJS [32] to parse command line arguments and to read and write files to the filesystem.

The instrumented version of the code (the generated file as the result of the set of transformations) serves as a replacement to the original file as the semantics of the original code is preserved. The only difference between the original code and the generated file, is the additional information flow tracking statements. In the next section we show the overhead incurred by the additional statements with respect to the original code. The generated code can run on any standard JavaScript engine and is not tied to NodeJS nor V8 engine.

6.2 Experiments and Results

In this section we present the results of three experiments that were conducted to assess the performance of the proposed framework. The first experiment shows the overhead incurred by only the transformation stage. Specifically, we compare the performance of the transformed code to the original code. The second experiment compares the instrumented JavaScript code that contains both, the transformations and the inlining of the information flow tracking statements, to the original code. In particular, we present the overhead in runtime incurred by the injected statements that monitor the information flow throughout the JavaScript code. The third experiment, compares the performance of our work with other work in the literature on information flow security for JavaScript. All the experiments were conducted on a MacBook Pro machine running Mac OS X 10.11 with a dual Core i5 Intel processor running at 2.7 Ghz and 8GB of system RAM.

6.2.1 Performance and Size Overhead

In the three experiments, we used code from the "Rosetta Code" [5] project. As they state on their website, "Rosetta Code is a programming chrestomathy site". The idea is to present solutions to same problem or task with many different programming languages. We used the JavaScript programming language page and picked the implementation of the following algorithms:

- **FFT**: Fast Fourier Transform algorithm.
- **LZW**: Lempel-Ziv-Welch (LZW) lossless data compression algorithm.
- **Knapsack (KS)**: Combinatorial optimization algorithm.
- **Floyd Triangles (FT)**: algorithm that generates Floyd's right-angled triangular array of natural numbers.
- **Hamming Numbers (HN)**: algorithm that generates 5-smooth and 7-smooth of hamming numbers.
- **24 Game (24)**: algorithm that takes as an input four digits, each from one to nine, with repetitions allowed and generates an arithmetic expression that evaluates to 24 using just those four digits, and all of those four digits are used exactly once. The arithmetic expression allows only multiplication, division, addition, and subtraction operators.
- **MD5**: MD5 hash function implementation in JavaScript[6].
- **SHA-256**: SHA-256 hash function implementation in JavaScript[7].

[5] http://rosettacode.org/wiki/Rosetta_Code
[6] http://pajhome.org.uk/crypt/md5/md5.html
[7] http://www.movable-type.co.uk/scripts/sha256.html

- **AES**: Advanced Encryption Standard algorithm implementation in JavaScript[8].

We used the 'time' UNIX command to measure how long each test takes to run. The measurements are in milliseconds. The reported time is user time. We took the average of five consecutive runs of each algorithm after running each algorithm for at least one time to minimize any caching/un-caching effect from the operating system. The variation in the reported time was minimal.

We used NodeJS version 4.1.1 to execute each JavaScript test.

Table 6.1 outlines the performance overhead of the transformation stage. The results show that the transformation stage incurs minimal overhead. This is expected as the transformation stage does not inject too many statements compared to the inlining stage.

Table 6.1: Performance overhead of transformed version of the benchmark JavaScript code.

	FFT	**LZW**	**KS**	**FT**	**HN**	**24**	**MD5**	**SHA**	**AES**
Original	118ms	122ms	118ms	117ms	120ms	121ms	113ms	109ms	114ms
Trans. Ver.	122ms	124ms	123ms	120ms	132ms	127ms	119ms	130ms	120ms

Table 6.2 represents the performance overhead of the instrumented code that contains the output of both the transformation and inlining stage. It is evident that the instrumented JavaScript code is 1.5x to 5x slower to execute than the original version. The main reason of the overhead is the extra statements that track the information flow for each variable in every expression in the code.

Tables 6.3 and 6.4 present the increase in size due to the transformation stage and inlining stage, respectively. The sizes are in number of Line-Of-Code (LOC). It is important to note here that we used a pretty printer to generate the JavaScript code after each stage. The reason was to compare visually the result of the transformation

[8]http://www.movable-type.co.uk/scripts/aes.html

Table 6.2: Performance overhead of instrumented version of the benchmark JavaScript code.

	FFT	**LZW**	**KS**	**FT**	**HN**	**24**	**MD5**	**SHA**	**AES**
Original	118ms	122ms	118ms	117ms	120ms	121ms	113ms	109ms	114ms
Instr. Ver.	211ms	185ms	428ms	178ms	610ms	318ms	265ms	214ms	468ms
Slow Down	1.7x	1.5x	3.6x	1.5x	5x	2.6x	2.3x	1.9x	4.1x

and inlining stage with the original code and to be able to debug it in the case we found any problem. This means that the size in LOC can be reduced if we use a code minifier like ”jscompress” [9]. For example, when we used the ”jscompress” tool to compress the instrumented version of the knapsack algorithm, the file size was reduced from 68 KB to 45 KB. The ”jscompress” tool does not use any compression algorithm (e.g. zip algorithm) instead, most of the file size reduction is achieved by removing comments and extra whitespace characters that are not needed by the JavaScript engines.

Table 6.3: Size overhead of transformed version of the benchmark JavaScript code.

	FFT	**LZW**	**KS**	**FT**	**HN**	**24**	**MD5**	**SHA**	**AES**
Original in LOC	117	82	74	78	84	113	385	183	417
Trans. Ver. in LOC	233	110	327	105	231	237	816	366	1030

Table 6.4: Size overhead of instrumented version of the benchmark JavaScript code.

	FFT	**LZW**	**KS**	**FT**	**HN**	**24**	**MD5**	**SHA**	**AES**
Original in LOC	117	82	74	78	84	113	385	183	417
Instr. Ver. in LOC	874	437	1041	449	782	952	3803	1367	3147
Size Increase	7.4x	5.3x	14x	5.7x	9.3x	8.4x	9.8x	7.4x	7.5x

6.2.2 Performance Comparison

In this experiment, we used the same JavaScript benchmark code used in the previous experiment to compare the performance of our implementation to the performance of

[9] http://jscompress.com

JSFlow version 1.0-2013106 [11] and FacetedValues [5], which are related work that provide information flow security for JavaScript. JSFlow uses NodeJS under the hood, on the other hand, FacetedValues uses Narcissus JavaScript interpreter, which in turn depends on SpiderMonkey JavaScript engine. With respect to the FacetedValues, we used the code that is part of the Firefox plugin ZaphodFacests [10]. We used the 'time' UNIX command to time how long each test takes when we run it using JSFlow and FacetedValues. We took the average of five consecutive runs. Table 6.2 represents the obtained results, which show that our implementation outperforms both JSFlow and FacetedValues in all the tests. The results are in milliseconds. Both, JSFlow and FacetedValues, failed with an error in the AES test with No Base64 encoding error. JSFlow failed in Floyd Triangle test with an error[11].

Table 6.5: Performance comparison of our approach with JSFlow and FacetedValues.

	FFT	**LZW**	**KS**	**FT**	**HN**	**24**	**MD5**	**SHA**	**AES**
Our Approach	211ms	185ms	428ms	178ms	610ms	318ms	265ms	214ms	468ms
JSFlow	468ms	510ms	12,218ms	failed	5798ms	11,144ms	596ms	768ms	failed
FacetedValues	320ms	380ms	14,415ms	407ms	13,362ms	7,395ms	350ms	343ms	failed

6.3 Evaluating Effectiveness

One of the key challenges in this field of research is the lack of public datasets or benchmarks for evaluating the effectiveness of the proposed approaches in identifying and preventing information leakages. Furthermore, virtually all the existing approaches are focused essentially on theoretical analysis (proof checking), and some times on testing efficiency, and study effectiveness only in an ad hoc way using simple illustrative code snippets. To fill this gap, we propose in this work a new security benchmark for

[10] https://github.com/taustin/ZaphodFacets

[11] All the code of our implementation and benchmarks are available online at https://bitbucket.org/bsayed/if-transpiler.

language-based information flow control. The benchmark has been made available publicly for the research community. The source code is available so that others can add to it and extend the benchmark.

6.3.1 Language-Based IFC Benchmark

The language-based IFC benchmark consists of 28 test cases, where in each test case an illicit information flow takes place. The scenario of the test cases is based on a popular JavaScript library that measures the strength of user-generated passwords. [12] The purpose of the website and its library, is to aid end-users in choosing a strong password. The library implements a `chkPassword(pwd)` function that takes the password as an argument and computes a *score* and a *complexity* for the password. The *score* is a numeric value between 0% and 100%, where 0% means very weak and 100% is very strong. The *complexity* is a string value that indicates the complexity level of the password in a text format, e.g. 'Too Short', 'Weak', 'Strong', etc.

The test cases are based on hypothetical scenario, where a benign website wants to use the library to aid its users in creating strong passwords. However, using a third-party library of such kind poses a security risk, where the library could leak information about the passwords created by the users of the website to unknown destination. The question here is, how to use the library, and make sure that the library or any of its functions do not leak information about the passwords being created by the users?

For the sake of simplicity and for the purpose of this experiment, we simplified the `chkPassword(pwd)` function by removing all the code that relates to HTML and CSS elements manipulation and all the code not relevant to IFC, since these are not the focus of the test. Our intent is to make the code of the test cases as readable as possible. In each test case, the password is passed as an argument to the `chkPassword(pwd)`

[12] http://www.passwordmeter.com

function and information about the password or the password itself is leaked using `console.log()` or `print()` statements. The `console.log()` and `print()` are considered the *output* statements that leak information. Each test case is designed to test a particular aspect of the syntax/semantics of the JavaScript language, e.g. implicit flow with If-Statements, handling of non-block structured control flow statements like `return`, implicit flow using exceptions, etc. In Tables 6.6 to 6.8, we list each test case and a short description about what features it tests in the JavaScript language and how the password or information about the password is being leaked. In Table 6.9 we categorized the test cases based on the language feature that is involed in the information leak.

It is important to note here that the 28 test cases are not exhaustive and we did not try to cover all the possible combinations of the different language constructs. However, the main goal is to cover as many features of the language as possible, especially the ones that are widely used in JavaScript applications and can be used by attackers to leak information, e.g. loops, if-Statements, *this* expression in different scopes, exceptions, continue, break, return statements, object properties, prototype-based inheritance, etc. We expect that other researchers will extend the benchmark by adding more test cases.

6.3.2 Experiment Results

In Table 6.10, we outline the results of running the 28 test cases against IF-Transplier (our approach), JSFlow [11] [13], and FacetedValues [5]. We were not able to use the tools proposed in [24] and [49] since both of the tools only supports a smaller subset of the JavaScript language. Although the tool proposed in [49] is wider in scope than the one proposed in [24], both do not model some aspects of the JavaScript language that are used in the test cases, e.g. `return`, `while-loop` `continue`, `break` and `throw`

[13] We used the most recent version at the time of performing this experiment, version `chalmerslbs-jsflow-4f18402991ff`.

statements.

The test cases were customized to be used correctly by each tool. For example, we used the `upg()` function to upgrade the security level of the password in the case of JSFlow, along with the `lprint()` function to print the security labels of variables. In the case of FacetedValues, we used the `cloakDeepak()` function to label the password as a *high* security level variable e.g. `cloakDeepak(pwd,'h')`. In the case of IF-Transplier, we assigned the shadow variable of the password which corresponds to the security level of the password in the typing environment Γ the value "1" for high, e.g. $\Gamma`['global']['pass'] = 1;`. The idea is that public variables and channels are assigned value "0" for low, and higher security variables and channels are assigned integer value "1". Our model is flexible and can be assigned any kind of labeling that an ordering relation can be implemented on, e.g. English alphabet, where 'a' is less than 'b', and 'b' is less than 'c', and so on. Every output statement was wrapped with a function call to our internal $`output()` function, which takes the following arguments:

- The security level of the variable that is going to be sent through the output channel e.g. '1' for high.
- The value of the variable that is going to be sent through the output channel, e.g. 'Temp1234'.
- The security level ℓ of the output channel e.g. '0' for low or public.
- The security policy that should be implemented in the case of illegal information flow, e.g. 'suppress'.
- The level of the current security context represented by $*pc*.

The $`output()` function checks its arguments against the chosen policy and decides either to implement the policy or outputs the value of the variable.

In Table 6.10, the ✓ symbol means the tool was able to prevent the leakage of information correctly. The ✗ symbol means the tool was not able to prevent (missed) the leakage of information. The `Exp` symbol means the tool throws a runtime exception at the point where a possible leak could happen, this is specific for the JSFlow tool. The `Err` symbol means a runtime error was thrown while the program was running which happened the most with the FacetedValues tool as we explain later.

As depicted by Table 6.10, although the FactedValues approach was able to prevent information leakage in the first four test cases, it aborted execution with runtime type error "`TypeError: pwd (type undefined) has no properties`" in the remaining test cases. The indicated error means that the "`pwd`" variable of type "`undefined`" has no properties. The variable "`pwd`" contains the password, however, the Faceted-Value approach assigns the "`undefined`" value to higher security level variables in low security contexts, this made the variable "`pwd`" contain the value "`undefined`". Although this might be correct in some cases and a better approach compared to the secure multi-execution approach proposed in [14] in terms of space and time efficiency as the authors outlined in [5]. On one hand, we think it is an overly restrictive approach as indicated in [47] and it might be unpractical in scenarios where confidentiality is targeted. On the other hand, as we mentioned before, their approach assigns the value "`undefined`" to higher security variables in lower security contexts, this approach is flawed in certain cases based on the semantics of the JavaScript language. As we formally defined it before in rule **IF-Else** in Table 4.3, any value va that is in the set {`0,null,undefined,false`} is considered a "`false`" value in any conditional expression in the language. This means that any value in this set can be replaced with "`undefined`" and the semantics will not change. For example, `var high={false or 0 or null or undefined},low=false; if(high) low=true;` regardless which value from the set the "high" variable will be assigned, the information

will be leaked to the "low" variable, since all the four values have the same semantic in conditional expression of the if-statement. It is also interesting that any variable that has the value "`undefined`", is assigned "`undefined`" as its hidden value, effectively changing nothing.

The JSFlow tool was able to prevent information leakage in 22 cases. In eight of these cases, the tool throws a runtime exception indicating a possibility of information leakage even if later in time the information did not leak. Consider the following example from test case number 09:

```
for (var j = 0; j < 16; j++)
  if (pwd.length == j)
    throw j;
```

The value of the throw-statement will be the length of the password in this case, however, the tool aborted execution prematurely before the throw-statement executed. Although, in this example they prevented a possible information leak, they aborted the execution of the program completely making their approach more restrictive than it needs to be, because the remaining code might not have any statements that leak the value of the throw-statement. We believe that permissiveness is an important factor in the practicality of information flow models, this has been studied in [24].

In the remaining six cases, the JSFlow tool failed to detect or prevent the information leakage, the main reason is that JSFlow enforces the non-interference policy dynamically at runtime with no static analysis. This means that attacks that take advantage of the untaken branches of the control-flow will pass undetected. This is true since the untaken branches can not be explored at runtime. For example, the following code is from test case number 20:

```
1  var pass = upg("temp1234");
2  var pwdLengthGreaterThan10 = true;
3  function foo(pwd) {
4    if(true) {
5      if (true) {
6        if (pwd.length > 10)
7          return;
8      } // end of inner if-statement
9      // JSFlow Failed to detect un-taken branch
10     pwdLengthGreaterThan10 = false;
11   } // end of outer if-statement
12 }
13 foo(pass);
14 lprint(pwdLengthGreaterThan10);
```

The "`pwdLengthGreaterThan10`" variable is assigned "`true`" value initially. When the function "`foo()`" is called with the password passed as an argument, the third if-statement will not be executed since the password length is not greater than 10, and the "`pwdLengthGreaterThan10`" variable will get updated in the body of the first if-statement with "`false`" value, effectively leaking information about the password length. This is in contrast to our approach which implements a hybrid approach that is capable of tracking information flow in the untaken branches. Although the work in [24] proposed a hybrid approach which should address some of the drawbacks of JSFlow, their approach is based on a much smaller subset of the JavaScript language and their approach is not flow-sensitive like ours. As we mentioned previously, we could not run the test cases against their implementation since their approach misses many features that the test cases depend on.

It is clear that our flow-sensitive hybrid approach is more permissive than both of JSFlow and FacetedValues, at the same time it is able to handle more types of attacks,

in particular, the attacks that rely on the untaken branches of the control-flow. This is in addition to the fact that our approach instruments the JavaScript code with the security monitor and does not rely on any specific JavaScript engine.

6.4 Summary and Discussion

In this chapter we presented the implementation and performance results of our IF-Transpiler. We compared the size and performance of the instrumented code to the original code and found that the instrumented code yields 1.5x to 5x performance overhead and 5.3x to 14x increase in LOC size. When we compared our approach to other implementations in the literature, we found that our approach outperformed them in all the benchmarks considered. This confirms the idea of inlining security monitors in source code can benefit from modern JavaScript engine optimizations.

Although our performance results are better than other work in the literature, we believe that there still a number of optimizations that can be implemented that could enhance the performance of our approach. Our focus in the implementation was on correctness rather than optimizations.

In addition we presented a language-based IFC benchmark that evaluates certain aspects of the JavaScript language with respect to information flow control. Our approach outperformed the other approaches, both in terms of permissiveness and accuracy when it came to the information flow attacks that take advantage of the untaken branches of the control-flow.

Table 6.6: Information flow test cases and their description (Part A).

Test Case	Short Description
Test01	Direct information flow and direct leakage of the whole password.
Test02	Usage of If-statement with dead code and direct leakage of the whole password
Test03	Usage of For-loop that loops only one time to leak the password as whole.
Test04	Usage of while-loop that loops only one time to leak the password as whole.
Test05	Usage of For-in-loop that leaks the characters of the password one at a time.
Test06	Usage of Array that leaks the password length by assigning a true boolean value to an element of the array at index "`password.length`". Then the code loops to find the index where value is true, then leaks that index (which is equal to the length of the password).
Test07	Combines a for-loop with a conditional `break` statement. The for-loop loops from 0 to 16 (where 16 is assumed to be max password length allowed by the library) and breaks conditionally when the loop counter equals the password length, then leaks the password counter, effectively leaking the password length.
Test08	Combines three for-loops with a conditional `continue` statement. The first for-loop, loops 16 times over an empty array and assigns a true boolean value for every element (where 16 is assumed to be max password length). The second for-loop loops over the array assigning a false boolean value to all its elements conditionally continuing when the loop counter equals the password length. This will result in only one element that is true in the array. The index of this element is the password length. The third for-loop loops over the array leaking the index of the element that has true boolean value, effectively leaking the password length.
Test09	Combines Try-Catch statement with conditional exception inside a for-loop. The for-loop loops 16 times conditionally throwing an exception when the loop counter equals the password length. The value of the exception is the loop counter. In the catch-clause, the thrown exception is caught and then leaked using a print statement. Effectively leaking the password length.
Test10	Combines a for-loop and a conditional `return` statement from the `chkPassword(pwd)` function. The for-loop loops 16 times conditionally returning from the function when the loop counter is equal to the password length. The return value is the loop counter. Then the result of invoking the `chkPassword(pwd)` function is leaked using a print statement.

Table 6.7: Information flow test cases and their description (Part B).

Test Case	Short Description
Test11	Similar to test case number 10, with the difference of the loop counter being a global variable. So, after invoking the function the loop counter (global variable) is leaked directly with a print statement.
Test12	Similar to the previous test case with the difference of the loop counter being referred to inside the function with the "`this.j`" expression instead of "`j`", at the same time there is a local variable with same name "`j`" defined in the function. The idea is to test if the IFC tool implements global vs. local scoping correctly.
Test13	Usage of object properties to test correct scoping of the "`this`" variable. In this case we invoke a method property on an object which contains a property that is assigned the password length, then the "`this`" variable is passed to a print statement. In this test case the "`this`" variable points to the object instead of the global scope. This is in contrast to the previous test case where the "`this`" variable is pointing to the global scope.
Test14	Similar to the previous test case but instead of passing the "`this`" variable to a print statement, the property that holds the password length is passed.
Test15	Usage of "`new`" operator to hide password length inside a newly created object. The password is passed to a function that is invoked with the "`new`" operator. Inside the function the password length is assigned to a property. This property is leaked using a print statement.
Test16	Similar to the previous case but instead of invoking the function with the "`new`" operator, the function is invoked normally, which makes the assignment of the property happens to the global object. Then the property is leaked through the global object using a print statement.
Test17	Similar to test case number 15, with the difference of the function returns an object. The idea is to try to confuse the IFC tool since invoking functions with the "`new`" operator returns object by default, however, if the function that was invoked contains an explicit return statement with an object as a return value. The returned object value of the return statement takes precedence on the one created by the runtime as the result of the "`new`" operator.
Test18	Combines two functions with one defined inside the other and the `this` variable scoping. The outer function is invoked first along with the password passed as an argument. The inner function is then invoked with the password passed as an argument. In the inner function the "`this`" variable is used to define a property that is assigned the password length. The idea is to confuse the IFC tool and to test correct scoping of the "`this`" variable inside nested functions.
Test19	Similar to case number 17 but instead of returning an object as the return value for the return statement, the function returns a literal (number 5). In this case the object created by the runtime takes precedence over the return statement in the function, since the return value is not an object.
Test20	Combines three nested if-statements with a function that returns conditionally on the size of the password. The idea is to test if the IFC tool supports the analysis of the untaken branches of the control-flow.

Table 6.8: Information flow test cases and their description (Part C).

Test Case	Short Description
Test21	Combines object aliasing and conditional "`delete`" expression statement. The idea is to confuse the IFC tool with the aliasing and using property existence to leak information about the password length.
Test22	Uses object aliasing to leak information about the password length.
Test23	Uses prototypical-based inheritance to leak information about the password length. A function "`foo()`" is defined, then its prototype is assigned a property "`i`" with the password length. Then the "`new`" operator is used to instantiate an instance object and the object's property "`i`" is then leaked using a print statement.
Test24	Similar to the previous test case but instead of defining a primitive property, it defines a property method on the prototype of the "`foo()`" function which is later invoked to leak the password length.
Test25	Defines a function with multiple conditional return statements that none of them get executed. The idea is to test the analysis of the untaken control-flow branches. But in this test case the untaken branches are shallow (i.e. not nested).
Test26	Similar to the previous case but instead of assignment statements in the untaken branches, it contains function calls.
Test27	Similar to the previous two test cases, but instead of assignment and function invocation the untaken branches contain assignment to object properties. The idea is to test if the IFC tool will handle different scenarios of untaken branches.
Test28	Uses with-statement with an object that contains a property that has same variable name of a global variable. Inside the with-statement there is an if-statement that branches on the password length. Information about the password is then leaked with a print statement. The idea is to test if the IFC tool will handle the untaken branch correctly inside the with-statement and if the tool will upgrade the property of the object or the global variable (correct scoping), since both have the same name.

Table 6.9: Language feature used in each test case (TC= Test Case, l=loop, r=return, c=continue/break, thr=throw, arr= array, OProp=Object Property, OProto=Object Prototype, and try=try-catch).

TC	**Language Feature**											
	l	**if**	**r**	**c**	**thr**	**with**	**arr**	**OProp**	**OProto**	**try**	**this**	**new**
TC 1												
TC 2		✓										
TC 3	✓											
TC 4	✓											
TC 5	✓											
TC 6	✓	✓					✓	✓				
TC 7	✓	✓						✓				
TC 8	✓	✓		✓			✓	✓				
TC 9	✓	✓			✓			✓		✓		
TC 10	✓	✓	✓					✓				
TC 11	✓	✓	✓					✓				
TC 12	✓	✓	✓					✓			✓	
TC 13								✓			✓	
TC 14								✓			✓	
TC 15								✓			✓	✓
TC 16								✓			✓	
TC 17								✓			✓	✓
TC 18			✓					✓			✓	
TC 19			✓					✓			✓	✓
TC 20		✓	✓									
TC 21		✓						✓				
TC 22								✓				
TC 23								✓	✓			✓
TC 24			✓					✓	✓		✓	✓
TC 25		✓	✓									
TC 26		✓	✓									
TC 27		✓	✓				✓	✓				
TC 28		✓				✓		✓				

d

Table 6.10: Results of the language-based IFC benchmark.

Test Case	**IF-Transpiler**	**JSFlow**	**FacetedValues**
Test01	✓	✓	✓
Test02	✓	✓	✓
Test03	✓	✓	✓
Test04	✓	✓	✓
Test05	✓	✓	Err
Test06	✓	Exp	Err
Test07	✓	Exp	Err
Test08	✓	Exp	Err
Test09	✓	Exp	Err
Test10	✓	Exp	Err
Test11	✓	Exp	Err
Test12	✓	Exp	Err
Test13	✓	✗	Err
Test14	✓	✓	Err
Test15	✓	✓	Err
Test16	✓	✓	Err
Test17	✓	✓	Err
Test18	✓	✓	Err
Test19	✓	✓	Err
Test20	✓	✗	Err
Test21	✓	Exp	Err
Test22	✓	✓	Err
Test23	✓	✓	Err
Test24	✓	✓	Err
Test25	✓	✗	Err
Test26	✓	✗	Err
Test27	✓	✗	Err
Test28	✓	✗	Err

Chapter 7

Conclusions and Future Work

7.1 Summary of Contributions

Key characteristics of modern web applications are interactivity, reactivity, and service composition which are attributed to the heavy usage of JavaScript on the client-side. Typically, modern web applications use several third-party JavaScript libraries to achieve such level of engagement which raises security concerns. In this research, we proposed IF-Transpiler, an inliner of a hybrid flow-sensitive security monitor that tracks information flow dynamically at runtime. To our knowledge, our framework is the first hybrid flow-sensitive work in the literature that targets the JavaScript language. Unlike other work in the literature, our work covers the full syntax of the language and models its prototype and scope chains. Our framework handles block structured (e.g. if-statements and loops) and non-block structured control-flow (e.g. throw and return statements).

Our framework handles the untaken branches of the control-flow by utilizing static-analysis. It collects and upgrades the variables that could have been modified or the functions that could have been invoked in the untaken branch. To our knowledge, our

framework is the first in the literature to combine static-analysis with a flow-sensitive type system for JavaScript language. In addition, it models JavaScript objects and distinguishes between function, array, and regular objects. One important aspect of our work, is the proper modelling of the *this* variable and what it points to in different scopes. For example, calling a function *foo()* with or without the *new* operator will result in making the *this* variable either point to newly created object or the global object.

Finally, we presented the implementation and compared the performance of our approach to un-instrumented JavaScript code and found that the the amount of overhead due to the information flow tracking statements ranges between 1.5x to 5x. We also compared our approach to other implementation in the literature and found that our approach is faster than the work in the literature with order of magnitude. This proves that the choice of inlining the security monitor is better in the sense of taking advantage of the optimizations implemented by the JavaScript engine running the code.

7.2 Limitations and Future Work

One of the JavaScript language features that is not currently handled by our framework is the "*arguments*" variable that is defined by default in every function by the standard JavaScript Runtime. The "*arguments*" variable behaves like an array object in the sense that it allows the programmer to access the arguments passed to the function using array index notation, e.g. `arguments[0]` means the first argument passed to the function, `arguments[1]` means the second, and so on. Another feature that is provided by the standard Runtime and not currently handled is the "*eval()*" function. The "*eval()*" function parses and evaluate the code passed to it as a string argument. It

is important to note here that the two missing features do not add any new constructs to the JavaScript syntax. Our framework does handle the full syntax of the language. The main reason we omitted them is that we believe they are part of the JavaScript language Runtime rather than part of the core language syntax. As a matter of fact, JavaScript language does come with a relatively rich Runtime API that defines objects and functions that provide basic and utility functionality. For example, the "*Array*" object, which serves as the base object for any array object, defines many functions for array elements manipulation, such as $pop(), push(), shift()$, and $splice()$. The "*Date*" object provides functions that handle date and time functionality, etc.

If the implementation of these functions are in the JavaScript language, our VM monitor will be able to track the information flow handily. However, in many cases, the objects and functions that are part of the standard Runtime API are implemented in a different language (e.g. C or C++ depending on the JavaScript VM implementation), which requires some additions to the events that get generated by the running VM (program) and consequently, additional transitions to the VM monitor. The additions will correspond to the APIs' operational semantics. For example, the $pop()$ function when invoked on an array object, it removes the last element and decrements the *length* property of the array object by one.

In future work, we intend to expand our security monitor to handle the APIs provided by the standard JavaScript Runtime. The same idea is applicable to the APIs provided by web browsers (e.g. for DOM manipulation) and other non-standard JavaScript Runtimes (e.g. NodeJS).

Appendix A

Proof of the Soundness of the Hybrid Flow-Sensitive Security Monitor

Definition 16. ($\rightsquigarrow$) For any two statements d and S such that d is not any of the $\Box$, $\rtimes$, or the *throw* statements, predicate $d \rightsquigarrow \langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, S\rangle\rangle$ holds iff $\exists\ H_0, r_0$ and $\langle\Gamma_0, \Sigma_0, \Lambda_0, \Delta_0, pc_0\rangle$ such that
$\langle\langle\Gamma_0, \Sigma_0, \Lambda_0, \Delta_0, pc_0\rangle, \langle H_0, r_0, d\rangle\rangle \rightarrow^* \langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, S\rangle\rangle$ where $\Lambda_0 = \{\bot_{\mathcal{L}}\}$, $\Delta_0 = \emptyset$, and $r_0 = \#Global$.

Lemma 1 establishes some properties of ($\rightsquigarrow$) predicate.

Lemma 1. *(Reachability).*

- *If* $d \rightsquigarrow \langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, S'\rangle\rangle$ *and* $\langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, S'\rangle\rangle \rightarrow^* \langle\langle\Gamma', \Sigma', \Lambda', \Delta', pc'\rangle, \langle H', r', S''\rangle\rangle$ *and if an exception got thrown, we assume the existence of a try-catch clause in an outer scope, then* $d \rightsquigarrow \langle\Gamma', \Sigma', \Lambda', \Delta', pc'\rangle, \langle H', r', S''\rangle$ *holds.*

- *If* $d \rightsquigarrow \langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{if(e)}\ S_1\ \texttt{else}\ S_2\rangle\rangle$ *holds, then both* S_1 *and* S_2 *do not contain* $\Box$ *or* $\rtimes$ *statements.*

- *If* $d \rightsquigarrow \langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{for(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{) S}\rangle\rangle$ *holds, then* S *does not contain* $\Box$ *or* $\rtimes$ *statements.*

- *If* $d \rightsquigarrow \langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{while(e) S}\rangle\rangle$ *holds, then* S *does not contain* $\Box$ *or* $\rtimes$ *statements.*

- *If* $d \rightsquigarrow \langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{do}\{\texttt{S}\}\texttt{while(e)}\rangle\rangle$ *holds, then* S *does not contain* $\Box$ *or* $\rtimes$ *statements.*

- *If* $d \rightsquigarrow \langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{for(x in e) S}\rangle\rangle$ *holds, then* S *does not contain* $\Box$ *or* $\rtimes$ *statements.*

- *If* $d \rightsquigarrow \langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{for(var x=}e_1\ \texttt{in}\ e_2\texttt{) S}\rangle\rangle$ *holds, then* S *does not contain* $\Box$ *or* $\rtimes$ *statements.*

- *If* $d \rightsquigarrow \langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{with(e)}\{\texttt{S}\}\rangle\rangle$ *holds, then* S *does not contain* $\Box$ *or* $\rtimes$ *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 $\Box$, $\rtimes$, or the *throw* statements.

Lemma 2. *($\sqcup$ on typing environments) Given* H_1*,* H_2 *and* H_3*, typing environments* Γ*,* Γ'*, and* Γ''*, and structure security environments* Σ*,* Σ'*, and* Σ''*, it holds that*

- $\Gamma \sqcup \Gamma' \sqcup \Gamma'' = \Gamma \sqcup \Gamma'' \sqcup \Gamma'$

- $\Sigma \sqcup \Sigma' \sqcup \Sigma'' = \Sigma \sqcup \Sigma'' \sqcup \Sigma'$

- *If* $H_1 \sim^{\ell}_{\Gamma,\Sigma} H_2$ *then* $H_1 \sim^{\ell}_{\Gamma\sqcup\Gamma',\Sigma\sqcup\Sigma'} H_2$

- *If* $H_1 \sim^{\ell}_{\Gamma,\Sigma} H_2$ *and* $H_2 \sim^{\ell}_{\Gamma\sqcup\Gamma',\Sigma\sqcup\Sigma'} H_3$ *then* $H_1 \sim^{\ell}_{\Gamma\sqcup\Gamma',\Sigma\sqcup\Sigma'} H_3$

Lemma 3. *(Behaviour of the VM Monitor when transitioning in arbitrary security level ℓ) 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. $H' \sim^{\ell}_{\Gamma',\Sigma'} H$

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 $\Lambda_0 = \{\bot_\mathcal{L}\}$, by applying M-Assign-Var in Table 4.5 then $\Gamma' = \Gamma(r_x)[x \mapsto pc.\ell \sqcup \tau]$, Σ is not changed $\Sigma' = \Sigma$, then $H' \sim^{\ell\sqcup\tau}_{\Gamma',\Sigma'} H$ 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 4.5 then $\Gamma' = \Gamma[x \mapsto [y \mapsto pc.\ell \sqcup \tau_e]]$, and $\Sigma' = \Sigma[x \mapsto pc.\ell \sqcup \tau_e \sqcup \Sigma(r_x)(x)]$, let $\ell' = pc.\ell \sqcup \tau_e \sqcup \Sigma(r_x)(x)$ and by Lemma 2 item 2 holds with respect to ℓ'.

- `x[`e_1`]=`e_2`;`) by reachability property (definition 1), item 1 holds, by applying M-Assign-Obj-Prop in Table 4.5 then $\Gamma' = \Gamma(r_x)[x \mapsto [m \mapsto pc.\ell \sqcup \tau_1 \sqcup \tau_2]$, and $\Sigma' = \Sigma(r_x)[x \mapsto pc.\ell \sqcup \tau_e \sqcup \Sigma(r_x)(x)]$, let $\ell' = pc.\ell \sqcup \tau_1 \sqcup \tau_2 \sqcup \Sigma(r_x)(x)$ and by Lemma 2 item 2 holds with respect to ℓ'.

- `x[`e_1`]=`e_2`;`) same as the previous case with the assumption that x is an array-object.

- `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 state, consequently $\Gamma' = \Gamma, \Sigma' = \Sigma$ and item 2 holds.

- `if(e)` S_1 `else` S_2) by reachability property (definition 1) item 1 holds, we assume "e" evaluates to a $va \notin \{0, null, undefined, false\}$ (the proof when "e" evaluating to a $va \in \{0, null, undefined, false\}$ proceeds similarly) by T-If-Stmt rule in Table 3.4, If-Then rule in Table 4.3, and M-Branch rule in Table 4.5 we have
  $\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{if(e)} S_1 \ \texttt{else} \ S_2\rangle\rangle \xrightarrow{b(e,S_2)} \langle\langle \Gamma', \Sigma', \Lambda', \Delta', pc'\rangle, \langle H', r, \rtimes;\ S_1\rangle\rangle$
  when M-Branch rule is applied, the Δ' *control-flow stack* will contain all the variables and functions in the untaken branch S_2 and the security level ℓ of the e conditional expression, $\Delta' = \texttt{push}(\Delta, \langle \vec{x}, \tau\rangle)$ where $\vec{x} = collect(S_2)$
  when T-If-Stmt rule is applied, the Λ' *security-context stack* will contain the higher of the security context where the If-Statement called and security level of expression e, $\Lambda' = \texttt{push}(\Lambda, \langle pc.\ell \sqcup \tau\rangle)$, then End rule in Table 4.4 and the M-Join rule in Table 4.5 will get applied, we get
  $\langle\langle \Gamma', \Sigma', \Lambda', \Delta', pc'\rangle, \langle H', r, \rtimes\rangle\rangle \xrightarrow{j} \langle\langle \Gamma'', \Sigma'', \Lambda', \Delta, pc'\rangle, \langle H'', r, \square\rangle\rangle$ 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 $H''' \sim^{\ell'}_{\Gamma''', \Sigma'''} H$ where $\ell' = pc.\ell \sqcup \tau$ and τ is the security level of the conditional expression "e" and $\Gamma''' = \Gamma \sqcup \Gamma' \sqcup \Gamma'', \Sigma''' = \Sigma \sqcup \Sigma' \sqcup \Sigma''$ hence item 2 holds.

- `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 $b(e, skip)$ the monitor performs the join transition without upgrading any variables or functions since $collect(skip) = \emptyset$, 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_2, $b(e_2, S)$.

- `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, $b(e, S)$.

- `x=e(`$\vec{e}$`);`) by reachability property (definition 1) item 1 holds, by applying rule T-Func-Call in Table 3.4, $\Gamma'' = \Gamma \sqcup \hat{\Gamma}[x \mapsto \tau_{ret} \sqcup pc.\ell], \Sigma'' = \Sigma \sqcup \hat{\Sigma}, H'' = \hat{H}[x \mapsto va], H \sim^{\ell}_{\Gamma'',\Sigma''} H''$, 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 3.5.

- `return e;`) by reachability property (definition 1) item 1 holds, by applying T-Return rule in Table 3.5, Return rule in Table 4.4, the monitor will apply M-Compensate rule in Table 4.5 to upgrade the entries of the security context stack Λ with $pc.\ell$ until one level above the label lbl,
$\langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \ \texttt{e}\rangle\rangle \rightarrow \langle\langle\Gamma', \Sigma', \Lambda, \Delta, pc\rangle, \langle H', r, va\rangle\rangle$,
$\langle\langle\Gamma', \Sigma', \Lambda, \Delta, pc\rangle, \langle H', r, \texttt{return} \ \ \texttt{va}\rangle\rangle \xrightarrow{k(\texttt{FUNC})} \langle\langle\Gamma'', \Sigma'', \Lambda', \Delta, pc\rangle, \langle H'', r, \Box\rangle\rangle$ where $\Lambda' = comp_{\Lambda}(\texttt{FUNC}, \ell)$, $\ell = pc.\ell \sqcup \tau$ and τ is the security type of expression e, where $\Gamma'' = \Gamma \sqcup \Gamma', \Sigma'' = \Sigma \sqcup \Sigma'$ by Lemma 2 then $H'' \sim^{\ell}_{\Gamma'',\Sigma''} H$ 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 $comp_{\Lambda}(\texttt{TRY} \mid \texttt{FUNC}, \ell)$.

- `break;`) by reachability property (definition 1) item 1 holds, by applying T-Break rule in Table 3.5 and Break rule in Table 4.4, the monitor will apply

M-Compensate rule in Table 4.5 which modifies only security context stack Λ hence, $\Gamma' = \Gamma, \Sigma' = \Sigma$, therefore $H \sim^{\ell}_{\Gamma,\Sigma} H'$ 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 Λ to the label "lbl" $comp_{\Lambda}(\texttt{lbl}, pc.\ell)$.
- `continue lbl;`) same as previous case.
- `try{S}`) by reachability property (definition 1) item 1 holds, by applying T-Try rule in Table 3.5 and Try rule in Table 4.4, the monitor steps over try-statements. The only change is in Λ, 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 3.5 and Catch rule in Table 4.4, the monitor steps over catch-statements. The only change is in Λ, 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 3.5 and Finally rule in Table 4.4, 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 $\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, S\rangle\rangle$ is secure according to Definition 12 (TINI).*

proof: By induction over the number of the ℓ-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:

$\langle\langle\Gamma_1, \Sigma_1, \Lambda_1, \Delta_1, pc_1\rangle, \langle H_1, r, s\rangle\rangle \overset{\vec{w_1}}{\Longrightarrow} \langle\langle\Gamma'_1, \Sigma'_1, \Lambda'_1, \Delta'_1, pc_1\rangle, \langle H'_1, r, \square\rangle\rangle$

and

$\langle\langle\Gamma_2, \Sigma_2, \Lambda_2, \Delta_2, pc_2\rangle, \langle H_2, r, s\rangle\rangle \overset{\vec{w_2}}{\Longrightarrow} \langle\langle\Gamma'_2, \Sigma'_2, \Lambda'_2, \Delta'_2, pc_2\rangle, \langle H'_2, r, \square\rangle\rangle$ then it holds that $\vec{w_1} = \vec{w_2}$, with respect to security level ℓ, 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 $\langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \mathcal{S}\rangle\rangle \sim \langle H \uplus mst, r, \hat{\mathcal{S}}\rangle$ 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 $\hat{\mathcal{S}}$ is the instrumented version of statement $\mathcal{S}$ as defined by . We show that every execution of the VM monitor ending with a transition to the $\square$ statement, has an equivalent execution of the inlined monitor ending with a t-transition to the $\square$ statement. By applying rule $\Gamma', \Lambda, \Delta, pc \vdash \mathcal{I}(\square) \Rightarrow_{\mathcal{I}} \square$ we always obtain a valid transformation relation between $\mathcal{S}'$ and $\hat{\mathcal{S}}'$ 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 $|_{out}$ 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 $\mathcal{S}$.

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 visiable to any observer (attacker).

proof: By rule induction $\longrightarrow$ on the different cases :

We will start by the most interesting cases first.

Case $\texttt{if}(e)\ \{S_1\}\ \texttt{else}\ \{S_2\}$:

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

$$\langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{if}(e)\ \{\rtimes;\ S_1\}\ \texttt{else}\ \{\rtimes;\ S_2\}\rangle\rangle$$

By rule T-If-Stmt in Table 3.4

$$\langle\langle\Gamma, \Sigma, \Lambda', \Delta, pc'\rangle, \langle H, r, \texttt{if}(e)\ \{\rtimes;\ S_1\}\ \texttt{else}\ \{\rtimes;\ S_2\}\rangle\rangle$$

$\xrightarrow{\alpha_1}$ where $\alpha_1 = b(e, S_2)$

$$\langle\langle\Gamma, \Sigma, \Lambda', \Delta', pc'\rangle, \langle H, r, \rtimes;\ S_1\rangle\rangle$$

$\xrightarrow{j}$

$$\langle\langle\Gamma', \Sigma', \Lambda', \Delta, pc'\rangle, \langle H, r, \Box;\ S_1\rangle\rangle$$

$=$

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

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

$$\langle H \uplus mst, r, \text{"}\$xs = \$Collect(S_2);\ ...\text{"}\rangle$$

$\xrightarrow{\beta_1}$ where $\beta_1 = b(e, S_2)$

$$\langle H \uplus mst = (..., \Delta', ...), r, \text{"}\$\Lambda.push(\{\ell : \$pc.\ell \sqcup sec_lvl(e), id :' \texttt{IF}'\});\ ...\text{"}\rangle$$

$\xrightarrow{sk}$ Assuming $e \notin \{\texttt{0}, \texttt{null}, \texttt{undefined}, \texttt{false}\}$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta', pc'), r, \text{"}\$upgrade(\$xs, \$pc.\ell);\ ...\text{"}\rangle$$

$\xrightarrow{j}$

$$\langle H \uplus mst = (\Gamma', \Lambda', \Delta, pc'), r, \text{"}\checkmark;\ \hat{S}_1'\text{"}\rangle$$

$\xrightarrow{t}$

$$\langle H \uplus mst = (\Gamma', \Lambda', \Delta, pc'), r, \square;\ \hat{S}_1'\rangle$$

$=$

$$\langle H \uplus mst', r, \hat{\mathcal{S}}'\rangle \quad \text{which define } mst', \hat{S}' = \square$$

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

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 ϵ as outlined in Table 4.2.

Case $\texttt{while}(e)\ \{S\}$:

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

$$\langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{while}(e)\ \{S\}\rangle\rangle$$

- By rule T-While in Table 3.4

$$\langle\langle\Gamma, \Sigma, \Lambda', \Delta, pc'\rangle, \langle H, r, \texttt{while}(e)\ \{S\}\rangle\rangle$$

$\xrightarrow{\alpha_1}$ where $\alpha_1 = b(e, S)$

$$\langle\langle\Gamma, \Sigma, \Lambda', \Delta', pc'\rangle, \langle H, r, \rtimes\rangle\rangle$$

$\xrightarrow{j}$

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

- By rule T-While in Table 3.4

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

$=$

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

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

$$\langle H \uplus mst, r, \text{"}\$\Lambda.push(\{\ell : \$pc.\ell \sqcup sec_lvl(e), lbl :' \texttt{LOOP}'\});\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (..., \Lambda', ..., pc'), r, \text{"}\$\Delta.push(\{xs : \$Collect(S), \ell : \$pc.\ell\});\ ...\text{"}\rangle$$

$$\xrightarrow{b(e,S)} \quad \text{Assuming } e \in \{\texttt{0}, \texttt{null}, \texttt{undefined}, \texttt{false}\}$$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta', pc'), r, \text{"}\$tmp = \$\Delta.pop();\ \$upgrade(\$tmp.xs, \$tmp.\ell);\ ...\text{"}\rangle$$

$$\xrightarrow{j}$$

$$\langle H \uplus mst = (\Gamma', \Lambda', \Delta, pc'), r, \text{"}\texttt{if}(\$should_comp)\{\$Comp_{\Lambda}(\$should_comp.lbl, \$pc.\ell);\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma', \Lambda', \Delta, pc'), r, \$\Lambda.pop();\ +\checkmark\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma', \Lambda, \Delta, pc), r, \checkmark\rangle$$

$$\xrightarrow{t}$$

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

$$=$$

$$\langle H \uplus mst', r, \hat{\mathcal{S}}'\rangle \qquad \text{which define } mst', \hat{S}' = \square$$

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

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 ϵ as outlined in Table 4.2.

Case `for`$(;\ e;\ )\{S\}$:

Similar to while loop case.

Case $\texttt{for}(x \texttt{ in } e)\{S\}$:

Similar to while loop case.

Case `throw e`:

We show the VM monitor execution first.

$$\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{throw e}\rangle\rangle$$

- By rule T-Throw in Table 3.5

$$\xrightarrow{\alpha_1} \quad \langle\langle \Gamma, \Sigma, \Lambda', \Delta, pc'\rangle, \langle H, r, \texttt{throw e}\rangle\rangle$$

where $\alpha_1 = k(\texttt{TRY} \mid \texttt{FUNC})$

$$= \quad \langle\langle \Gamma', \Sigma', \Lambda'', \Delta, pc''\rangle, \langle H, r, \Box\rangle\rangle$$

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

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

$$\langle H \uplus mst, r, \text{"}\$old_pc = \$pc;\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst, r, \text{"}\texttt{while}(\$pc.id\ \texttt{!== 'FUNC'}\&\&\ \$pc.id\ \texttt{!== 'TRY'})\{\$\Lambda.pop();\ \};\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, \text{"}\$\Lambda[\$\Lambda.len - 1] = \{\ell : \$old_pc.\ell \sqcup sec_lvl(e)\};\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, \text{"}\texttt{throw e;}\checkmark\text{"}\rangle$$

$$\xrightarrow{k(\texttt{TRY}|\texttt{FUNC})}$$

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

$$\xrightarrow{t}$$

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

$$=$$

$$\langle H \uplus mst', r, \hat{\mathcal{S}}'\rangle \quad \text{which define } mst', \hat{S}' = \square$$

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

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 ϵ as outlined in Table 4.2.

Case `return;`

We show the VM monitor execution first.

$$\langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{return;}\rangle\rangle$$

- By rule T-Return in Table 3.5

$$\xrightarrow{\alpha_1} \quad \langle\langle\Gamma, \Sigma, \Lambda', \Delta, pc'\rangle, \langle H, r, \texttt{return;}\rangle\rangle \quad \text{where } \alpha_1 = k(\texttt{FUNC})$$

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

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

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

$$\langle H \uplus mst, r, \text{"}\$old_pc = \$pc;\ \ldots\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst, r, \text{"}\texttt{while}(\$pc.id\ \texttt{!== 'FUNC'})\{\$\Lambda.pop();\ \};\ \ldots\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, \text{"}\texttt{if}(\$\$CS.InvokedAsContr)\{$$

$$\$\$CS.this.\Sigma = \$\$CS.this.\Sigma \sqcup \$old_pc.\ell;\ \$\Lambda[\$\Lambda.len - 1] = \{\ell : \$\$CS.this;\ \};$$

$$\}\ \texttt{else}\ \{$$

$$\$\Lambda[\$\Lambda.len - 1] = \{\ell : \$old_pc.\ell\}\};\ \ldots\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, \text{"}\texttt{return;}\checkmark\text{"}\rangle$$

$$\xrightarrow{k(\texttt{FUNC})}$$

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

$$\xrightarrow{t}$$

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

$$=$$

$$\langle H \uplus mst', r, \hat{\mathcal{S}}'\rangle \qquad \text{which define } mst', \hat{S}' = \square$$

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

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 ϵ as outlined in Table 4.2.

Case `return x;`

We show the VM monitor execution first.

$$\langle\langle\Gamma,\Sigma,\Lambda,\Delta,pc\rangle,\langle H,r,\texttt{return x;}\rangle\rangle$$

- By rule T-Return in Table 3.5

$$\langle\langle\Gamma,\Sigma,\Lambda',\Delta,pc'\rangle,\langle H,r,\texttt{return x;}\rangle\rangle$$

$\xrightarrow{\alpha_1}$ where $\alpha_1 = k(\texttt{FUNC})$

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

$=$

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

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

$$\langle H \uplus mst, r, "\$old_pc = \$pc;\ ..." \rangle$$

$\xrightarrow{sk}$

$$\langle H \uplus mst, r, "\texttt{while}(\$pc.id\ \texttt{!== 'FUNC'})\{\$\Lambda.pop();\ \};\ ..." \rangle$$

$\xrightarrow{sk}$

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

$\xrightarrow{sk}$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, "\texttt{if}(isObj(\$rx))\{$$

$$\$rx.\Sigma = \$rx.\Sigma \sqcup \$old_pc.\ell;$$

$$\$\Lambda[\$\Lambda.len - 1] = \{\ell : \$rx\};$$

$$\}\texttt{else if}(\$\$CS.InvokedAsContr)\{$$

$$\$\$CS.this.\Sigma = \$\$CS.this.\Sigma \sqcup \$old_pc.\ell;\ \$\Lambda[\$\Lambda.len - 1] = \{\ell : \$\$CS.this;\ \};$$

$$\}\ \texttt{else}\ \{$$

$$\$\Lambda[\$\Lambda.len - 1] = \{\ell : \$rx \sqcup \$old_pc.\ell\}\};\ ..." \rangle$$

$\xrightarrow{sk}$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, "\texttt{return x;}\checkmark" \rangle$$

$\xrightarrow{k(\texttt{FUNC})}$

$$\langle H \uplus mst = (\Gamma, \Lambda'', \Delta, pc''), r, "\checkmark" \rangle$$

$\xrightarrow{t}$

$$\langle H \uplus mst = (\Gamma, \Lambda'', \Delta, pc''), r, \Box \rangle$$

$=$

$$\langle H \uplus mst', r, \hat{\mathcal{S}}' \rangle \qquad \text{which define } mst', \hat{S}' = \Box$$

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

$mst', r, \hat{\mathcal{S}}'\rangle$.

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 ϵ as outlined in Table 4.2.

Case `continue;`:

We show the VM monitor execution first.

$$\langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{continue;}\rangle\rangle$$

By rule T-Continue in Table 3.5

$$\langle\langle\Gamma, \Sigma, \Lambda', \Delta, pc'\rangle, \langle H, r, \texttt{continue;}\rangle\rangle$$

$\xrightarrow{\alpha_1}$ where $\alpha_1 = k(\texttt{LOOP})$

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

$=$

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

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

$$\langle H \uplus mst, r, \text{"}\$old_pc = \$pc;\ \ldots\text{"}\rangle$$

$\xrightarrow{sk}$

$$\langle H \uplus mst, r, \text{"}\texttt{while}(\$pc.id\ \texttt{!== 'LOOP'})\{\$\Lambda.pop();\ \};\ \ldots\text{"}\rangle$$

$\xrightarrow{sk}$

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

$\xrightarrow{sk}$

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

$\xrightarrow{k(\texttt{LOOP})}$

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

$\xrightarrow{t}$

$$\langle H \uplus mst = (\Gamma, \Lambda'', \Delta, pc''), r, \Box\rangle$$

$=$

$$\langle H \uplus mst', r, \hat{\mathcal{S}}'\rangle \qquad \text{which define } mst', \hat{S}' = \Box$$

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

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 ϵ as outlined in Table 4.2.

Case `break;`:

Same as previous case.

Case `continue label;`:

We show the VM monitor execution first.

$$\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{continue label;}\rangle\rangle$$

- By rule T-Continue in Table 3.5

$$\langle\langle \Gamma, \Sigma, \Lambda', \Delta, pc'\rangle, \langle H, r, \texttt{continue label;}\rangle\rangle$$

$\xrightarrow{\alpha_1}$ where $\alpha_1 = k(\texttt{label})$

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

$=$

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

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

$$\langle H \uplus mst, r, \text{"}\$old_pc = \$pc;\ \ldots\text{"}\rangle$$

$$\xrightarrow{sk}$$

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

$$\xrightarrow{sk}$$

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

$$\xrightarrow{sk}$$

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

$$\xrightarrow{k(\texttt{label})}$$

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

$$\xrightarrow{t}$$

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

$$=$$

$$\langle H \uplus mst', r, \hat{\mathcal{S}}'\rangle \qquad \text{which define } mst', \hat{S}' = \square$$

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

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 ϵ as outlined in Table 4.2.

Case `break label;`:

Same as the previous case.

Case $\texttt{with}(x)\{S\}$:

We show the VM monitor execution first.

$$\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{with}(x)\{S\}\rangle\rangle$$

- By rule T-With in Table 3.5

$$\langle\langle \Gamma, \Sigma, \Lambda', \Delta, pc'\rangle, \langle H, r, \texttt{with}(x)\{S\}\rangle\rangle$$

$\xrightarrow{sk}$ assuming expression x points to an object-reference $\mathbf{r_o}$,

otherwise, it is a runtime error.

$$\langle\langle \Gamma, \Sigma, \Lambda', \Delta, pc'\rangle, \langle H, \mathbf{r_o}, S\rangle\rangle$$

$\xrightarrow{\vec{w_1}}$

$$\langle\langle \Gamma', \Sigma', \Lambda', \Delta, pc'\rangle, \langle H, \mathbf{r_o}, \square\rangle\rangle$$

- By rule T-With in Table 3.5

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

$=$

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

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

$$\langle H \uplus mst, r, \text{"}\$\Lambda.push(\{\ell : \$pc.\ell \sqcup sec_lvl(x), lbl : \texttt{'WITH'}\});\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

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

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, \text{"}\$ro.this = \$\$CS.this;\ \$\$CS = \$ro;\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, \text{"}S;\ \$\Lambda.pop();\ \checkmark\text{"}\rangle$$

$$\xrightarrow{\vec{w_2}}$$

$$\langle H \uplus mst = (\Gamma', \Lambda', \Delta, pc'), r, \text{"}\$\Lambda.pop();\ \checkmark\text{"}\rangle$$

$$\xrightarrow{sk}$$

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

$$\xrightarrow{t}$$

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

$$=$$

$$\langle H \uplus mst', r, \hat{\mathcal{S}}'\rangle \quad \text{which define } mst', \hat{S}' = \square$$

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

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 $|_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$ we get $\vec{\alpha}\ |_{out}\ \vec{w_1}$ and $\vec{\beta}\ |_{out}\ \vec{w_2}$ 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 $\texttt{try}\{S\}$:

We show the VM monitor execution first.

$$\langle\langle\Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{try}\{S\}\rangle\rangle$$

- By rule T-Try in Table 3.5

$$\langle\langle\Gamma, \Sigma, \Lambda', \Delta, pc'\rangle, \langle H, r, \texttt{try}\{S\}\rangle\rangle$$

$\xrightarrow{sk}$

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

$\xrightarrow{\vec{w_1}}$

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

- By rule T-With in Table 3.5

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

$=$

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

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

$$\langle H \uplus mst, r, \text{"}\texttt{try}\{\$\Lambda.push(\{\ell : \$pc.\ell, lbl : \texttt{'TRY'}\});\ S;\ \$\Lambda.pop();\ \}\checkmark\text{"}\rangle$$

$\xrightarrow{sk}$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, \text{"}S;\ \$\Lambda.pop();\ \}\checkmark\text{"}\rangle$$

$\xrightarrow{\vec{w_2}}$

$$\langle H \uplus mst = (\Gamma', \Lambda', \Delta, pc'), r, \text{"}\$\Lambda.pop();\ \}\checkmark\text{"}\rangle$$

$\xrightarrow{sk}$

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

$\xrightarrow{t}$

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

$=$

$$\langle H \uplus mst', r, \hat{\mathcal{S}}'\rangle \qquad \text{which define } mst', \hat{S}' = \square$$

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

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 $|_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$ we get $\vec{\alpha}\ |_{out}\ \vec{w_1}$ and $\vec{\beta}\ |_{out}\ \vec{w_2}$ 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 $(\Gamma, \Lambda', \Delta, pc')$ for the inlined monitor.

We show the VM monitor execution first.

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

$$\xrightarrow{sk}$$

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

$$\xrightarrow{\vec{w_1}}$$

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

- By rule T-Catch in Table 3.5

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

$$=$$

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

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

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

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, \text{"}S;\ \$\Lambda.pop();\ \}\checkmark\text{"}\rangle$$

$$\xrightarrow{\vec{w_2}}$$

$$\langle H \uplus mst = (\Gamma', \Lambda', \Delta, pc'), r, \text{"}\$\Lambda.pop();\ \}\checkmark\text{"}\rangle$$

$$\xrightarrow{sk}$$

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

$$\xrightarrow{t}$$

$$\langle H \uplus mst = (\Gamma', \Lambda, \Delta, pc), r, \Box\rangle$$

$$=$$

$$\langle H \uplus mst', r, \hat{\mathcal{S}}'\rangle \qquad \text{which define } mst', \hat{S}' = \Box$$

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

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 $|_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$ we get $\vec{\alpha} \mid_{out} \vec{w_1}$ and $\vec{\beta} \mid_{out} \vec{w_2}$ 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 $(\Gamma, \Lambda', \Delta, pc')$ for the inlined monitor.

We show the VM monitor execution first.

$$\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \texttt{finally}\{S;\ \} \rangle \rangle$$
$$\xrightarrow{sk}$$
$$\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, S \rangle \rangle$$
$$\xrightarrow{\vec{w_1}}$$
$$\langle \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H, r, \square \rangle \rangle$$
$$=$$
$$\langle \langle \Gamma', \Sigma', \Lambda, \Delta, pc \rangle, \langle H, r, S' \rangle \rangle \qquad \text{which defines } \Gamma', \Sigma', S' = \square$$

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

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, ``\texttt{catch(x)}\{S;\ \}\checkmark" \rangle$$
$$\xrightarrow{sk}$$
$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, ``S;\ \}\checkmark" \rangle$$
$$\xrightarrow{\vec{w_2}}$$
$$\xrightarrow{sk}$$
$$\langle H \uplus mst = (\Gamma', \Lambda, \Delta, pc), r, ``\checkmark" \rangle$$
$$\xrightarrow{t}$$
$$\langle H \uplus mst = (\Gamma', \Lambda, \Delta, pc), r, \square \rangle$$
$$=$$
$$\langle H \uplus mst', r, \hat{\mathcal{S}}' \rangle \qquad \text{which define } mst', \hat{S}' = \square$$

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

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 $|_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$ we get $\vec{\alpha}\ |_{out}\ \vec{w_1}$ and $\vec{\beta}\ |_{out}\ \vec{w_2}$ 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 $\texttt{output}_\ell(e)$:

The VM monitor execution will depend on the chosen policy:

Case OutputFailStop:

$$\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \texttt{output}_\ell(e) \rangle \rangle$$

$\xrightarrow{o_\ell(e,va)}_{o_\ell(va)}$ in case $\tau \sqcup pc.\ell \sqsubseteq \ell$, where τ is the security level of e

$$\langle \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle, \langle H, r, \Box \rangle \rangle$$

$=$

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

Otherwise:

The VM monitor will get stuck as outlined in Table 4.5 rule M-OutputFailStop

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

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, "if(sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell)\ \{\texttt{output}_\ell(e);\ \checkmark\}$$

$$\texttt{else}\ \$diverge\$;\ "\rangle$$

$\xrightarrow{sk}$

$$\text{in case where } sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell$$

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, "\texttt{output}_\ell(e);\ \checkmark"\rangle$$

$\xrightarrow{o_\ell(e,va)}$

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, "\checkmark"\rangle$$

$\xrightarrow{t}$

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

$=$

$$\langle H \uplus mst, r, \hat{\mathcal{S}}'\rangle \qquad \text{which define } \hat{S}' = \square$$

Otherwise:

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \$diverge\$\rangle,$$

$$\text{where } \$diverge\$ \text{ means stuck or lack of progress.}$$

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

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

In the case where $sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell$ is **false**, both monitors are stuck.

Case OutputSupress:

$$\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{output}_\ell(e)\rangle\rangle$$

$\xrightarrow{o_\ell(e,va)}_{o_\ell(va)}$ in case $\tau \sqcup pc.\ell \sqsubseteq \ell$, where τ is the security level of e

$$\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \Box\rangle\rangle$$

$=$

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

Otherwise:

$$\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{output}_\ell(e)\rangle\rangle$$

$\xrightarrow{o_\ell(e,va)}_{nothing}$ in case $\tau \sqcup pc.\ell \not\sqsubseteq \ell$, where τ is the security level of e

$$\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \Box\rangle\rangle$$

$=$

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

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

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"}if(sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell)\ \{\texttt{output}_{\ell}(e);\ \}\ \texttt{else}\ \ Skip;\ \checkmark\text{"}\rangle$$

$$\xrightarrow{sk} \quad \text{in case where } sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell$$

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"}\texttt{output}_{\ell}(e);\ \checkmark\text{"}\rangle$$

$$\xrightarrow{o_{\ell}(e,va)}$$

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

$$\xrightarrow{t}$$

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

$$=$$

$$\langle H \uplus mst, r, \hat{\mathcal{S}}'\rangle \qquad \text{which define } \hat{S}' = \square$$

Otherwise:

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"}if(sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell)\ \{\texttt{output}_{\ell}(e);\ \}\ \texttt{else}\ \ Skip;\ \checkmark\text{"}\rangle$$

$$\xrightarrow{sk} \quad \text{in case where } sec_lvl(e) \sqcup \$pc.\ell \not\sqsubseteq \ell$$

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

$$\xrightarrow{sk}$$

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

$$\xrightarrow{t}$$

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

$$=$$

$$\langle H \uplus mst, r, \hat{\mathcal{S}}'\rangle \qquad \text{which define } \hat{S}' = \square$$

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

If we apply the output relation $|_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$ we either get $\vec{\alpha}\ |_{out}\ o_{\ell}(va)$

and $\vec{\beta} \mid_{out} o_\ell(va)$ when $sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell$ is true, or we get $\vec{\alpha} \mid_{out} \emptyset$ and $\vec{\beta} \mid_{out} \emptyset$ when $sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell$ is false. Trivially, in the first case $o_\ell(va) = o_\ell(va)$ and in the second case $\emptyset = \emptyset$, hence, clause **(b)** of Theorem 2 holds.

Case OutputDefault:

$$\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{output}_\ell(e)\rangle\rangle$$

$$\xrightarrow{o_\ell(e,va)}_{o_\ell(va)} \quad \text{in case } \tau \sqcup pc.\ell \sqsubseteq \ell, \text{where } \tau \text{ is the security level of } e$$

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

$$=$$

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

Otherwise:

$$\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{output}_\ell(e)\rangle\rangle$$

$$\xrightarrow{o_\ell(e,va)}_{o_\ell(D)} \quad \text{in case } \tau \sqcup pc.\ell \not\sqsubseteq \ell, \text{where } \tau \text{ is the security level of } e$$

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

$$=$$

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

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

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, "if(sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell)\ \{\texttt{output}_\ell(e);\ \}$$
$$\texttt{else output}_\ell(D);\ \checkmark" \rangle$$

$\xrightarrow{sk}$ in case where $sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell$

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, "\texttt{output}_\ell(e);\ \checkmark" \rangle$$

$\xrightarrow{o_\ell(e,va)}$

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, "\checkmark" \rangle$$

$\xrightarrow{t}$

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \Box \rangle$$

$=$

$$\langle H \uplus mst, r, \hat{\mathcal{S}}' \rangle \qquad \text{which define } \hat{S}' = \Box$$

Otherwise:

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, "if(sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell)\ \{\texttt{output}_\ell(e);\ \}$$
$$\texttt{else output}_\ell(D);\ \checkmark" \rangle$$

$\xrightarrow{sk}$ in case where $sec_lvl(e) \sqcup \$pc.\ell \not\sqsubseteq \ell$

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, "\texttt{output}_\ell(D);\ \checkmark" \rangle$$

$\xrightarrow{o_\ell(e,D)}$

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, "\checkmark" \rangle$$

$\xrightarrow{t}$

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \Box \rangle$$

$=$

$$\langle H \uplus mst, r, \hat{\mathcal{S}}' \rangle \qquad \text{which define } \hat{S}' = \Box$$

In case where $sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell$ is true or false, if we apply rule $\Gamma, \Lambda, \Delta, pc \vdash I(\Box) \Rightarrow_I \Box$ we obtain a valid transformation relation between $\mathcal{S}'$ and $\hat{\mathcal{S}}'$ and we get $\langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle \cong (mst = (\Gamma_{in}, \Lambda_{in}, \Delta_{in}, pc_{in}))$, consequently we get $\langle \langle H, r, \mathcal{S}' \rangle, \langle \Gamma, \Sigma, \Lambda, \Delta, pc \rangle \rangle \sim$

$\langle H \uplus mst, r, \hat{\mathcal{S}}'\rangle$ hence, clause **(a)** of Theorem 2 holds .

If we apply the output relation $|_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$ we either get $\vec{\alpha} \mid_{out} o_\ell(va)$ and $\vec{\beta} \mid_{out} o_\ell(va)$ when $sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell$ is true, or we get $\vec{\alpha} \mid_{out} o_\ell(D)$ and $\vec{\beta} \mid_{out} o_\ell(D)$ when $sec_lvl(e) \sqcup \$pc.\ell \sqsubseteq \ell$ is false. Trivially, in the first case $o_\ell(va) = o_\ell(va)$ and in the second case $o_\ell(D) = o_\ell(D)$, hence, clause **(b)** of Theorem 2 holds.

Case $\texttt{foo = function}(\vec{x})\{S\};$ $\mid$ $\texttt{function foo}(\vec{x})\{S\};$:

VM monitor execution:

$$\langle\langle \Gamma, \Sigma, \Lambda, \Delta, pc\rangle, \langle H, r, \texttt{function foo}(\vec{x})\{S\}\rangle\rangle$$

by Definition 5 $\qquad \rhd_{FuncLit}$

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

$=$

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

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

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"}\texttt{function foo}(\vec{x})\{\mathcal{I}(S')\};\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H' \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"}\$\$CS['foo'] = \{x_1 : \$pc.\ell, ..., x_n = \$pc.\ell,$$

$$\$fscope : \$pc.\ell, prototype : \{\Sigma : \$pc.\ell\}, \Sigma : \$pc.\ell\};\ \checkmark\text{"}\rangle$$

$$\xrightarrow{sk}$$

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

$$\xrightarrow{t}$$

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

$$=$$

$$\langle H' \uplus mst', r, \hat{\mathcal{S}}'\rangle \quad \text{which defines } H', mst', \hat{S}' = \square$$

If we apply rule $\Gamma', \Lambda, \Delta, pc \vdash \mathcal{I}(\square) \Rightarrow_{\mathcal{I}} \square$ we obtain a valid transformation relation between $\mathcal{S}'$ and $\hat{\mathcal{S}}'$ and we get $\langle \Gamma', \Sigma', \Lambda, \Delta, pc\rangle \cong (mst' = (\Gamma'_{in}, \Lambda_{in}, \Delta_{in}, pc_{in}))$, consequently we get $\langle \Gamma', \Sigma', \Lambda, \Delta, pc\rangle, \langle\langle H', r, \mathcal{S}'\rangle\rangle \sim \langle H' \uplus mst', r, \hat{\mathcal{S}}'\rangle$ hence, clause **(a)** of Theorem 2 holds .

If we apply the output relation $|_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$ we get $\vec{\alpha}\ |_{out}\ \emptyset$ and $\vec{\beta}\ |_{out}\ \emptyset$, 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:

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

by Definition 3 $\quad \rhd_{ObjectLit}$

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

$=$

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

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

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

$\xrightarrow{sk}$

$$\langle H'\uplus mst=(\Gamma,\Lambda,\Delta,pc),r,\text{"}\$\$CS['foo']=\{pn1:sec_lvl(e_1)\sqcup\$pc.\ell,...,$$
$$pn_n:sec_lvl(e_n)\sqcup\$pc.\ell,__proto__:\texttt{null},\Sigma:\$pc.\ell\sqcup sec_lvl(e_1)...\sqcup sec_lvl(e_n)\};\ \checkmark\text{"}\rangle$$

$\xrightarrow{sk}$

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

$\xrightarrow{t}$

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

$=$

$$\langle H'\uplus mst',r,\hat{\mathcal{S}}'\rangle \quad \text{which defines } H',mst',\hat{S}'=\square$$

By applying rule $\Gamma',\Lambda,\Delta,pc\vdash\ \mathcal{I}(\square)\Rightarrow_{\mathcal{I}}\ \square$ we obtain a valid transformation relation between $\mathcal{S}'$ and by taking $\Gamma'_{in}=\Gamma'\uplus\Sigma'$, $\hat{\mathcal{S}}'$ and we get $\langle\Gamma',\Sigma',\Lambda,\Delta,pc\rangle\cong(mst'=(\Gamma'_{in},\Lambda_{in},\Delta_{in},pc_{in}))$, consequently we get $\langle\Gamma',\Sigma',\Lambda,\Delta,pc\rangle,\langle\langle H',r,\mathcal{S}'\rangle\rangle\sim\langle H'\uplus mst',r,\hat{\mathcal{S}}'\rangle$ hence, clause **(a)** of Theorem 2 holds .

If we apply the output relation $|_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$ we get $\vec{\alpha}\ |_{out}\ \emptyset$ and $\vec{\beta}\ |_{out}\ \emptyset$, 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 $\triangleright_{ArrayLit}$ outlined in Definition 4.

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

VM monitor execution:

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

by Definition 6 $\triangleright_{FuncCall}$

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

$\xrightarrow{a_v(x,va)}$

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

where $\tau_{ret} = \Lambda'.pop()$ and r_x is a reference to variable x.

$=$

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

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

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"}\$rf = \$scope(\$\$CS,'f')['f'];\ \$rf.scope = \$\$CS;\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"}\$rf.this = \$\Gamma['Global'];\ \$rf['x_i'] = sec_lvl(e_i) \sqcup \$pc.\ell;\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"}\$\Lambda.push(\{\ell : \$rf.\$fscope \sqcup \$pc.\ell \sqcup \$rf.\Sigma, id :' \texttt{FUNC}'\});\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma, \Lambda', \Delta, pc'), r, \text{"}x = f(\vec{e});\ \$rx = \$scope(\$\$CS,'x')['x'];\ ...\text{"}\rangle$$

$$\xrightarrow{a_v(x,va)}$$

$$\langle \hat{H}(r_x)[x \mapsto va] \uplus mst = (\hat{\Gamma}, \Lambda', \Delta, pc'), r, \text{"}\$rx = \$\Lambda.pop().\ell;\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle \hat{H}' \uplus mst = (\hat{\Gamma}(r_x)[x \mapsto \$\Lambda.pop().\ell], \Lambda, \Delta, pc), r, \text{"}(isObj(\$rx))?\$rx.\Sigma = \$rx.\Sigma \sqcup \$pc.\ell : \$rx = \$rx \sqcup \$pc.\ell;\ \checkmark\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle \hat{H}' \uplus mst = (\hat{\Gamma}(r_x)[x \mapsto pc.\ell \sqcup \Lambda.pop().\ell], \Lambda, \Delta, pc), r, \text{"}\checkmark\text{"}\rangle$$

$$\xrightarrow{t}$$

$$\langle \hat{H}' \uplus mst = (\hat{\Gamma}', \Lambda, \Delta, pc), r, \square\rangle$$

$$=$$

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

By applying rule $\hat{\Gamma}', \Lambda, \Delta, pc \vdash I(\square) \Rightarrow_I \square$ we obtain a valid transformation relation between $\mathcal{S}'$ and $\hat{\mathcal{S}}'$ and by taking $\hat{\Gamma}'_{in} = \hat{\Gamma}' \uplus \hat{\Sigma}'$, we get $\langle \hat{\Gamma}', \hat{\Sigma}', \Lambda, \Delta, pc\rangle \cong (\hat{mst}' = (\hat{\Gamma}'_{in}, \Lambda_{in}, \Delta_{in}, pc_{in}))$, consequently we get $\langle \hat{\Gamma}', \hat{\Sigma}', \Lambda, \Delta, pc\rangle, \langle\langle \hat{H}', r, \mathcal{S}'\rangle\rangle \sim \langle \hat{H}' \uplus \hat{mst}', r, \hat{\mathcal{S}}'\rangle$ hence, clause **(a)** of Theorem 2 holds .

If we apply the output relation $|_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$ we get $\vec{\alpha}\ |_{out}\ \emptyset$ and $\vec{\beta}\ |_{out}\ \emptyset$, trivially $\emptyset = \emptyset$ since the two monitors did not generate any observable events hence,

clause **(b)** of Theorem 2 holds.

Case `x = y.f(`$\vec{e}$`);` | `x = y['f'](`$\vec{e}$`);` :

Similar to the previous case with the difference of the variable `this` will be pointing to object y instead of the global variable and the usage of the property lookup relation $\triangleright_{Proto}$ 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 5.3.

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

VM monitor execution:

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

by Definition 8 $\triangleright_{ConstrFunc}$

$$\langle\langle\Gamma'', \Sigma'', \Lambda, \Delta, pc\rangle, \langle H'', r, x = \hat{v}\rangle\rangle$$

$\xrightarrow{a_v(x,\hat{v})}$

$$\langle\langle\Gamma''(r_x)[x \mapsto \Gamma''(r)(\hat{v})], \Sigma''(r_x)[x \mapsto \Sigma''(r)(\hat{v})], \Lambda, \Delta, pc\rangle, \langle H''(r_x)[x \mapsto \hat{v}], r, \square\rangle\rangle$$

where $\tau_{\hat{v}}$ is the security type of $\hat{v}$ and r_x is a reference to variable x.

$=$

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

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

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"}\$rf = \$scope(\$\$CS,' f')['f'];\ \$rf.scope = \$\$CS;\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma, \Lambda, \Delta, pc), r, \text{"}\$rf.this = \$scope(\$\$CS,' x')['x'] = \{\Sigma : \$pc.\ell,$$

$$__proto__ : \$rf.prototype\};\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma(r_x)[x \mapsto \{\Sigma : \$pc.\ell, ...\}], \Lambda, \Delta, pc), r, \text{"}\$rf['x'_i] = sec_levl(e_i);$$

$$\$rf.InvokedAsConstr = true;\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma', \Lambda, \Delta, pc), r, \text{"}\$\Lambda.push(\{\ell : \$rf.\$fscope \sqcup \$pc.\ell \sqcup \$rf.\Sigma, id :' \texttt{FUNC}'\});\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H \uplus mst = (\Gamma', \Lambda', \Delta, pc'), r, \text{"}x = \texttt{new}\ \ f(\vec{e});\ \$rx = \$scope(\$\$CS,' x')['x'];\ ...\text{"}$$

$$\xrightarrow{a_v(x,\hat{v})}$$

$$\langle H''(r_x)[x \mapsto \hat{v}] \uplus mst = (\Gamma', \Lambda', \Delta, pc'), r, \text{"}\$rx = \$\Lambda.pop().\ell;\ ...\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H''' \uplus mst = (\Gamma'(r_x)[x \mapsto \$\Lambda.pop().\ell], \Lambda, \Delta, pc), r, \text{"}(isObj(\$rx))?\$rx.\Sigma =$$

$$\$rx.\Sigma \sqcup \$pc.\ell : \$rx = \$rx \sqcup \$pc.\ell;\ \checkmark\text{"}\rangle$$

$$\xrightarrow{sk}$$

$$\langle H''' \uplus mst = (\Gamma''(r_x)[x \mapsto pc.\ell \sqcup \Lambda.pop().\ell], \Lambda, \Delta, pc), r, \text{"}\checkmark\text{"}\rangle$$

$$\xrightarrow{t}$$

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

$$=$$

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

By applying rule $\Gamma''', \Lambda, \Delta, pc \vdash \ \mathcal{I}(\square) \Rightarrow_{\mathcal{I}} \ \square$ we obtain a valid transformation relation between $\mathcal{S}'$ and $\hat{\mathcal{S}}'$ and by taking $\Gamma'''_{in} = \Gamma''' \uplus \Sigma'''$, we get $\langle \Gamma''', \Sigma''', \Lambda, \Delta, pc\rangle \cong$

$(mst''' = (\Gamma'''_{in}, \Lambda_{in}, \Delta_{in}, pc_{in}))$, consequently we get $\langle\Gamma''', \Sigma''', \Lambda, \Delta, pc\rangle, \langle\langle H''', r, \mathcal{S}'\rangle\rangle \sim \langle H''' \uplus mst''', r, \hat{\mathcal{S}}'\rangle$ hence, clause **(a)** of Theorem 2 holds .

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

Case `x = new y.f(`$\vec{e}$`);` | `x = new y['f'](`$\vec{e}$`);` :

Similar to the previous case with the difference of using property lookup relation $\rhd_{Proto}$ 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 5.3.

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

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

$$\xrightarrow{a_p(x.y,e)}$$

$$\langle\langle\Gamma(r_x)[x \mapsto [y \mapsto pc.\ell \sqcup \tau]], \Sigma(r_x)[x \mapsto pc.\ell \sqcup \tau \sqcup \Sigma(r_x)(x)], \Lambda, \Delta, pc\rangle,$$
$$\langle H(r_x)[x \mapsto [y \mapsto va]], r, \square\rangle\rangle$$

where $\langle H, r, x\rangle \rhd_{Scope} r_x$ and τ is the security level of expression e.

$$=$$

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

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

$$\langle H \uplus mst, r, \text{"}\texttt{x.y = e};\ \$ry = \$scope(\$\$CS,'x')['y'] = sec_lvl(e);\ ...\text{"}\rangle$$

$$\xrightarrow{a_p(x.y,e)}$$

$$\langle H(r_x)[x \mapsto [y \mapsto va]] \uplus mst, r, \text{"}\$ry = \$scope(\$\$CS,'x')['y'] =$$

$$sec_lvl(e);\ ...\text{"}\rangle$$

$$\xrightarrow{sk} \quad \tau = sec_lvl(e)$$

$$\langle H' \uplus mst = (\Gamma(r_x)[x \mapsto [y \mapsto \tau]], \Lambda, \Delta, pc), r, \text{"}(isObj(\$ry))?\$ry.\Sigma = \$ry.\Sigma \sqcup \$pc.\ell :$$

$$\$ry = \$ry \sqcup \$pc.\ell;\ \checkmark\text{"}\rangle$$

$$\xrightarrow{sk} \quad \text{assuming } isObj(\$ry) \text{ is true}$$

$$\langle H' \uplus mst = (\Gamma(r_x)[x \mapsto [y \mapsto [\Sigma \mapsto \Sigma \sqcup pc.\ell]]], \Lambda, \Delta, pc), r, \text{"}\checkmark\text{"}\rangle$$

$$\xrightarrow{t}$$

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

$$=$$

$$\langle H' \uplus mst', r, \hat{\mathcal{S}}'\rangle \qquad \text{which define } H', mst', \hat{\mathcal{S}}'$$

By applying rule $\Gamma', \Lambda, \Delta, pc \vdash I(\square) \Rightarrow_I \square$ we obtain a valid transformation relation between $\mathcal{S}'$ and $\hat{\mathcal{S}}'$ and by taking $\Gamma'_{in} = \Gamma' \uplus \Sigma'$ and $H' = H'$, we get $\langle \Gamma', \Sigma', \Lambda, \Delta, pc\rangle \cong (mst' = (\Gamma'_{in}, \Lambda_{in}, \Delta_{in}, pc_{in}))$, consequently we get $\langle\langle \Gamma', \Sigma', \Lambda, \Delta, pc\rangle, \langle H', r, \mathcal{S}'\rangle\rangle \sim \langle H' \uplus mst', r, \hat{\mathcal{S}}'\rangle$ hence, clause **(a)** of Theorem 2 holds.

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

Case `x[y] = e;` :

VM monitor execution:

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

$$\xrightarrow{a_i(x.y,e)}$$

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

where $\langle H,r,x\rangle \rhd_{Scope} r_x$, $\tau_1 = sec_lvl(y)$ and τ_2 is the security level of expression e.

$$=$$

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

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

$$\langle H\uplus mst,r,\text{"}\texttt{x[y] = e;}\ \$ry=\$scope(\$\$CS,'x')[y]=sec_lvl(e);\ ...\text{"}\rangle$$

$$\xrightarrow{a_i(x[y],y,e)}$$

$$\langle H(r_x)[x\mapsto[y\mapsto va]]\uplus mst,r,\text{"}\$ry=\$scope(\$\$CS,'x')[y]=sec_lvl(e);$$
$$\$tmp=(isObj(sec_lvl(y))?sec_lvl(y).\Sigma:sec_lvl(y);\ ...\text{"}\rangle$$

$$\xrightarrow{sk} \quad \tau_1=\$tmp \text{ and } \tau_2=sec_lvl(e)$$

$$\langle H'\uplus mst=(\Gamma(r_x)[x\mapsto[y\mapsto\tau_2]],\Lambda,\Delta,pc),r,\text{"}(isObj(\$ry))?\$ry.\Sigma=\$ry.\Sigma\sqcup\$tmp\sqcup$$
$$\$pc.\ell:\$ry=\$ry\sqcup\$tmp\sqcup\$pc.\ell;\ \checkmark\text{"}\rangle$$

$$\xrightarrow{sk} \quad \text{assuming } isObj(\$ry) \text{ is true}$$

$$\langle H'\uplus mst=(\Gamma(r_x)[x\mapsto[y\mapsto[\Sigma\mapsto\Sigma\sqcup pc.\ell\sqcup\$tmp]]],\Lambda,\Delta,pc),r,\text{"}\checkmark\text{"}\rangle$$

$$\xrightarrow{t}$$

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

$$=$$

$$\langle H'\uplus mst',r,\hat{\mathcal{S}}'\rangle \quad \text{which define } H',mst',\hat{\mathcal{S}}'$$

By applying rule $\Gamma',\Lambda,\Delta,pc\vdash I(\square)\Rightarrow_I\square$ we obtain a valid transformation relation between $\mathcal{S}'$ and $\hat{\mathcal{S}}'$ and by taking $\Gamma'_{in}=\Gamma'\uplus\Sigma'$ and $H'=H'$, we get $\langle\Gamma',\Sigma',\Lambda,\Delta,pc\rangle\cong$

$(mst' = (\Gamma'_{in}, \Lambda_{in}, \Delta_{in}, pc_{in}))$, consequently we get $\langle\langle\Gamma', \Sigma', \Lambda, \Delta, pc\rangle, \langle H', r, \mathcal{S}'\rangle\rangle \sim \langle H' \uplus mst', r, \hat{\mathcal{S}}'\rangle$ hence, clause **(a)** of Theorem 2 holds.

If we apply the output relation $|_{out}$ on both of $\vec{\alpha}$ and $\vec{\beta}$ we get $\vec{\alpha} \mid_{out} \emptyset$ and $\vec{\beta} \mid_{out} \emptyset$, 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. ■

Bibliography

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

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

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

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

[5] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages, POPL '12, pages 165–178, New York, NY, USA, 2012. ACM.

[6] Luciano Bello and Eduardo Bonelli. On-the-fly inlining of dynamic dependency monitors for secure information flow. In Gilles Barthe, Anupam Datta, and Sandro Etalle, editors, *Formal Aspects of Security and Trust*, volume 7140 of *Lecture Notes in Computer Science*, pages 55–69. Springer Berlin Heidelberg, 2012.

[7] A Chudnov and D.A Naumann. Information flow monitor inlining. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 200–214, July 2010.

[8] CoffeeScript.org. Coffeescript. 2013.

[9] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th international conference on World wide web - WWW '10*, page 281, New York, New York, USA, 2010. ACM Press.

[10] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: fast and precise in-browser javascript malware detection. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[11] Luciano Bello Daniel Hedin, Arnar Birgisson and Andrei Sabelfeld. Jsflow: Tracking information flow in javascript and its apis. In *ACM Symposium on Applied Computing (SAC)*, 14 March 2014.

[12] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[13] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[14] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society.

[15] Facebook. Javascript sdk. 2013.

[16] Matthew Finifter, Joel Weinberger, and Adam Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, 2010.

[17] OWASP Top 10 for 2013. https://www.owasp.org/category:owasp_top_ten_project. 2014.

[18] James Garrett. Ajax: A new approach to web applications. *experiencezencom*, pages 1–5, 2005.

[19] Google. Chromium with the dart vm. 2013.

[20] Google. Google caja. 2013.

[21] Arjun Guha, Matthew Fredrikson, and Benjamin Livshits. Verified security for browser extensions. *SP '11 Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 115–130, May 2011.

[22] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP 2010–Object-Oriented . . .* , pages 1–25, 2010.

[23] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[24] D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive hybrid information flow control for a javascript-like language. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, pages 351–365, July 2015.

[25] D. Hedin and A Sabelfeld. Information-flow security for a core of javascript. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 3–18, June 2012.

[26] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of javascript. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 3–18, Washington, DC, USA, 2012. IEEE Computer Society.

[27] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM.

[28] Google Inc. Traceur compiler https://github.com/google/traceur-compiler.

[29] Google Inc. Dart language. 2013.

[30] Google Inc. V8 google's open source high-performance javascript engine. 2015.

[31] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *Proceedings of the 18th International Conference on Advanced Information Networking and Applications - Volume 2*, AINA '04, pages 145–, Washington, DC, USA, 2004. IEEE Computer Society.

[32] Joynet. Node js.

[33] Babel JS. https://babeljs.io.

[34] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes : A client-side solution for mitigating cross-site scripting attacks. *Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337, 2006.

[35] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, Oct 1973.

[36] P. Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pages 47–54, 2009.

[37] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 307–325, Berlin, Heidelberg, 2008. Springer-Verlag.

[38] Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 31(7):827 – 843, 2012. IFIP/SEC 2010 Security & Privacy Silver Linings in the Cloud.

[39] NPP Mavrommatis and MARF Monrose. All your iframes point to us. In *USENIX Security Symposium*, pages 1–16, 2008.

[40] Alexander Moshchuk and Tanya Bragin. A Crawler-based Study of Spyware on the Web. In *Network and Distributed Systems Security Symposium*, 2006.

[41] Alexander Moshchuk, Tanya Bragin, and Damien Deville. Spyproxy: Execution-based detection of malicious web content. In *USENIX Security Symposium*, pages 1–16, 2007.

[42] David Nolen. Clojurescript https://www.github.com/clojure/clojurescript. 2013.

[43] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[44] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *SEC'11 Proceedings of the 20th USENIX conference on Security*, 2011.

[45] Charles Reis, J Dunagan, HJ Wang, and O Dubrovsky. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web (TWEB)*, 1(3), 2007.

[46] Mozilla Rhino. https://developer.mozilla.org/en/docs/rhino.

[47] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 186–199, Washington, DC, USA, 2010. IEEE Computer Society.

[48] S. Saad, I. Traore, A. Ghorbani, B. Sayed, D. Zhao, Wei Lu, J. Felix, and P. Hakimian. Detecting p2p botnets through network behavior analysis and machine learning. In *Privacy, Security and Trust (PST), 2011 Ninth Annual International Conference on*, pages 174–180, July 2011.

[49] JosFragoso Santos and Tamara Rezk. An information flow monitor-inlining compiler for securing a core of javascript. In Nora Cuppens-Boulahia, Frdric Cuppens, Sushil Jajodia, Anas Abou El Kalam, and Thierry Sans, editors, *ICT Systems Security and Privacy Protection*, volume 428 of *IFIP Advances in Information and Communication Technology*, pages 278–292. Springer Berlin Heidelberg, 2014.

[50] B. Sayed, I. Traore, and A. Abdelhalim. Detection and mitigation of malicious javascript using information flow control. In *Privacy, Security and Trust (PST), 2014 Twelfth Annual International Conference on*, pages 264–273, July 2014.

[51] Bassam Sayed and Issa Traoré. Protection against web 2.0 client-side web attacks using information flow control. In *Proceedings of the 2014 28th International Conference on Advanced Information Networking and Applications Workshops*, WAINA '14, pages 261–268, Washington, DC, USA, 2014. IEEE Computer Society.

[52] Bassam Sayed, Issa Traore, and Amany Abdelhalim. Hybrid flow-sensitive security monitor for javascript (under review). *ACM Trans. Program. Lang. Syst.*

[53] Bassam Sayed, Issa Traore, and Amany Abdelhalim. If-transpiler: Inlining of hybrid flow-sensitive security monitor for javascript (to be submitted). *Computers & Security.*

[54] Paritosh Shroff, Scott Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 203–217, Washington, DC, USA, 2007. IEEE Computer Society.

[55] Paritosh Shroff, Scott F. Smith, and Mark Thober. Securing information flow via dynamic capture of dependencies. *J. Comput. Secur.*, 16(5):637–688, December 2008.

[56] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated analysis of security-critical javascript apis. In *2011 IEEE Symposium on Security and Privacy*, pages 363–378. Ieee, May 2011.

[57] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07*, 2007.

[58] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.

[59] YM Wang, Doug Beck, and Xuxian Jiang. Automated Web Patrol with Strider HoneyMonkeys. *Network and Distributed Systems Security Symposium*, pages 35–49, 2006.

[60] Wikipedia. Mikeyy computer worm. 2013.

[61] Wikipedia. Xmlhttprequest. 2013.

[62] Wikipedia.com. Javascript object notation. 2013.

[63] Wikipedia.com. Samy computer worm. 2013.

[64] David Zhao, Issa Traore, Ali Ghorbani, Bassam Sayed, Sherif Saad, and Wei Lu. Peer to peer botnet detection based on flow intervals. In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *Information Security and Privacy Research*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 87–102. Springer Berlin Heidelberg, 2012.

[65] David Zhao, Issa Traore, Bassam Sayed, Wei Lu, Sherif Saad, Ali Ghorbani, and Dan Garant. Botnet detection based on traffic behavior analysis and flow intervals. *Comput. Secur.*, 39:2–16, November 2013.