TÉCNICO LISBOA

Precise Information Flow Control for JavaScript

Francisco João do Vale Lopes e Silva Quinaz

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Prof. Ana Gualdina Almeida Matos

Examination Committee

Chairperson: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Members of the Committee: Prof. Carla Maria Gonçalves Ferreira

September 2021

Dedicated to my family.

Acknowledgments

I want to thank my supervisors, José Fragoso Santos and Ana Almeida Matos, for all their support and availability throughout this work. Your accompaniment was essential for this project, as well as for shaping my thinking to solve many problems that my academic-professional life can show me.

I want to thank all my friends, for accompany me and for giving me the right dose of fun, fellowship, love, and belief.

Finally, and not least, want to thank all my family, for all the opportunities and motivation they gave me to become someone as professional and as human as them. I consider it to be the biggest luck of my life.

Abstract

Nowadays, information flow control is particularly important on the Web. *JavaScript* programs that run in the browser can include scripts from different providers, which are often unknown to the user and execute in the context of the main web page with access to all of the user's resources. This raises important security concerns, which can be solved through the use of language-based mechanisms, such as information flow control.

JavaScript poses two fundamental problems to information flow analyses. On the one hand, the dynamic nature of the language makes it a difficult target for static analyses, resulting in too coarse over approximations with large numbers of false positives. On the other hand, the complexity of the language semantics renders the direct development of precise program analyses for *JavaScript* a challenging task. To counter these issues, we propose a new dynamic analysis for securing information flow in *JavaScript* that works by first compiling the given *JavaScript* program to a novel intermediate language for JavaScript analysis and specification called *ECMA-SL*.

This thesis is part of a larger project, whose goal is to build a tool-suit for *JavaScript* analysis based on *ECMA-SL*. Here, we contribute to the overarching *ECMA-SL* project in three different ways: first, we define the formal semantics of *ECMA-SL* and describe its interpreter; second, we design a new information flow monitor and inlining compiler for *ECMA-SL*; finally, we develop two distinct embedders for running *ECMA-SL* in *JavaScript*. By combining the various elements of the constructed infrastructure, we obtain a precise information flow monitor inlining compiler for *JavaScript*, which we thoroughly tested against *Test262*, *JavaScript*'s official test suite.

Keywords: Security Monitor, Information Flow Control, JavaScript, Program Instrumentation, Intermediate Languages, Formal Semantics

Resumo

Atualmente, o controlo de fluxos de informação é particularmente importante na web. Os programas escritos em JavaScript que correm no navegador podem incluir scripts de diferentes origens, normalmente desconhecidas pelo utilizador, executados no contexto da página principal, com acesso a todos os recursos do utilizador. Isto cria preocupações ao nível da segurança que podem ser resolvidas através do uso de mecanismos baseados em linguagens, como o controlo de fluxos de informação.

A linguagem JavaScript apresenta dois problemas fundamentais relativamente à análise de fluxos de informação. Por um lado, a natureza dinâmica da linguagem faz com que esta seja um alvo difícil para análises estáticas, resultando em aproximações incorretas e num grande número de falsos positivos. Por outro lado, a complexidade da semântica da linguagem torna o desenvolvimento de análises precisas para JavaScript uma tarefa desafiadora. Para combater estes problemas, propomos uma nova análise dinâmica para garantir a segurança de fluxos de informação em JavaScript. Esta análise compila o programa JavaScript fornecido para uma nova linguagem intermediária projetada para análise e especificação de JavaScript chamada ECMA-SL.

Esta tese faz parte de um projeto maior, cujo objetivo é construir um conjunto de ferramentas para análise de JavaScript baseadas em ECMA-SL. Aqui, contribuímos para o projeto ECMA-SL de três maneiras diferentes: primeiro, definimos a semântica formal do ECMA-SL e descrevemos o seu interpretador; em segundo lugar, projetámos um novo monitor de fluxos de informação e o seu compilador-em-linha para ECMA-SL; finalmente, desenvolvemos dois *embedders* distintos para executar ECMA-SL em JavaScript. Ao combinar os vários elementos da infraestrutura construída, obtemos um compilador-em-linha de um monitor de fluxos de informação preciso para JavaScript, que testamos exaustivamente contra a Test262, a bateria de testes oficial do JavaScript.

Keywords: Monitores de Segurança, Controlo de Fluxos de Informação, JavaScript, Instrumentação de Programas, Linguagens Intermediárias, Semântica Formal

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software security is a primary concern in modern software development. Hence, there are plenty of mechanisms for enforcing different types of security properties such as access control, secure information flow, and availability. However, security mechanisms in practice are not sufficiently robust to protect modern systems from security attacks, which may have a significant economic impact. Examples of relevant attacks in the last few years are: the *Spectre* (1), the *Meltdown* (2), and the *Foreshadow* (3), which took advantage of vulnerabilities found in microprocessors; and the Heartbleed Bug (4), a memory related vulnerability found in the OpenSSL's (version 1.0.1) (5) Heartbleed extension. In this project, we are specifically interested in security vulnerabilities that can be tackled using language-based mechanisms, such as program analysis (e.g. (6; 7; 8)) and instrumentation (e.g. (9; 10; 11; 12; 13; 14)). For this reason, the Heartbleed Bug is of special interest to us, as it could have been prevented using either static or dynamic language-based mechanisms.

Non-interference, defined in (15), is a mathematical property used to reason about how the execution of a program propagates dependencies between the resources that it manipulates, that is, how information flows between resources during the execution of a program. Informally, we say that a program is non-interferent if it is information flow secure, that is, if its execution does not generate illegal dependencies between the program's resources. Since the late 90s, the research community has proposed an extensive amount of mechanisms for enforcing non-interference such as: type systems, (e.g. (16; 17); information flow monitors, (e.g. (9; 10; 11; 12; 13; 14)); program logics, (e.g. (18; 19; 20; 21)); abstract interpreters, (e.g. (22)), amongst others, (e.g. (23)) However, none of the proposed mechanisms has been widely adopted in practice. As pointed out by Steve Zdancewic in (24):

"Despite their long history and appealing strengths, information-flow-based enforcement mechanisms have not been widely (or even narrowly!) used."

Nowadays, information flow control is particularly important on the Web. *JavaScript* programs that run in the browser can include scripts from different providers, which are often unknown to the user and execute in the context of the main web page, having access to all of its resources. In addition, these scripts can dynamically load more scripts; a common example is the download of advertisements and their banners through Ad Servers.

JavaScript poses two fundamental problems to information flow analyses. On the one hand, the dynamic nature of the language makes it a difficult target for static analyses, resulting in too coarse over approximations with large numbers of false positives. On the other hand, the complexity of the language semantics, whose standard has about 1000 pages, renders the direct development of precise program analyses a challenging task. Indeed, even the simplest language constructs, such as a variable

assignment, might trigger numerous implicit program behaviors.

Due to the dynamicity of the language, most information flow analyses for *JavaScript* proposed so far are based on dynamic approaches. More concretely, they are all based on information flow monitors, which instrument the execution of the original program with the tracking of security labels. Amongst the existing literature, we highlight the following three dynamic analyses:

- The inlining compiler proposed in (25) was the first compiler to inline an information flow monitor in a small core of *JavaScript* (ES3). This core includes some main features of the language, such as extensible objects, prototypical inheritance, and closures. However, it ignores most of the complexity of the *JavaScript* semantics and its implicit behaviors; for instance, it does not support getters and setters, implicit coercions, and none of the *JavaScript* built-in objects.

- The JEST compiler (26) inlines an information flow monitor in a subset of *JavaScript* (ES5). This compiler improves on the work of Fragoso Santos and Rezk (25) in that it supports the entire *JavaScript* syntax and a much wider range of its implicit behaviors, including implicit type coercions and property attributes.

- The JSFlow engine (27) is a security-enhanced JavaScript interpreter for fine-grained tracking of information flow in *JavaScript* programs (ES5). JSFlow supports both a large fragment of *JavaScript* and some APIs provided by the browser to client-side *JavaScript* programs, including part of the DOM API (28).

Importantly, none of the information flow analyses discussed above supports the entire *JavaScript* language. Furthermore, they offer no guarantees of capturing all of *JavaScript*'s implicit information flows. While the first inlining compiler (25) comes with a proof of soundness, it only targets a very small core of the language. In contrast, both JEST and JSFlow come with no proof of soundness.

We believe that the complexity of the *JavaScript* language makes it impossible to design a reasonably precise information flow control mechanism directly on *JavaScript*. As for other types of program analyses, an alternative approach is to compile the program to be analyzed to a simpler intermediate representation and perform the information flow analysis on that intermediate representation. This approach has been used by both the JaVerT tool-chain (29) for verifying and testing *JavaScript* programs, and the ADsafety tool (30) for verifying isolation properties of *JavaScript* applications. While JaVerT works by first compiling the given *JavaScript* program to a simple goto language called *JSIL*, ADsafety works by compiling the given program to *Lamba-JS* (31), a lambda-calculus specifically designed for *JavaScript* analysis and implementation.

This thesis is part of a larger project, developed at Instituto Superior Técnico, whose goal is to build a tool-suit for *JavaScript* analysis based on a new intermediate language called *ECMA-SL*. This language was specifically designed for specifying the *JavaScript* standard and analyzing *JavaScript* programs. The main advantage of the *ECMA-SL* compilation pipeline when compared to other compilation pipelines for *JavaScript* analyses is that it is tightly connected to the text of the standard, in that the *ECMA-SL* implementation of the *JavaScript* standard follows the text of the standard line-by-line. Furthermore, the *ECMA-SL* compilation pipeline was designed so that it can be easily adapted to new versions of the standard. This is a major concern regarding existing implementations, which mostly target older versions of the standard and are difficult to extend.

In the context of this thesis, we contribute to the overarching *ECMA-SL* project in three ways: first, we defined the formal semantics of *ECMA-SL* and implemented its interpreter; second, we designed a new information flow monitor and inlining compiler for *ECMA-SL*; finally, we developed two distinct embeddings for running *ECMA-SL* in *JavaScript*. Below, we briefly describe each of these contributions.

ECMA-SL Semantics and Interpreter We define the formal semantics of *ECMA-SL* in small-step style and use it to guide the implementation of an *ECMA-SL* interpreter written in *OCaml*. As our goal is to perform security analysis on *ECMA-SL* code, we make our *ECMA-SL* interpreter parametric on the security monitor to be used. This means that one can change the security monitor to be used without modifying the implementation of the semantics.

ECMA-SL Information Flow Monitor We formally define a monitor for enforcing secure information flow in *ECMA-SL*. The proposed monitor follows the no-sensitive-upgrade discipline (9; 32), which mandates that no low-level resources be updated inside high-level contexts. This monitor was implemented in OCaml following two different approaches: a lock-step monitor and an inlining compiler. Both approaches were tested individually and compared with each other.

ECMA-SL JavaScript Embedders We design two embedders of *ECMA-SL* into *JavaScript*, a deep embedder and a shallow one. The deep embedder consists of an *ECMA-SL* interpreter written in *JavaScript*. The shallow embedder provides a more performant implementation, consisting of a compiler that translates *ECMA-SL* programs into *JavaScript* programs. While the deep embedder was tested against a small custom-made test suite, the shallow embedder was tested against a large fragment of *Test262* (33), the official *JavaScript* test suite.

Putting it all together By combining several elements of the constructed infrastructure, we are able to obtain a precise information flow inlining compiler for *JavaScript*. To this end, we first compile the given *JavaScript* program to *ECMA-SL* using the *JS2ECMA-SL* compiler; then, we inline the information flow monitor into the generated *ECMA-SL* program using our *ECMA-SL* inlining compiler; finally, we compile the obtained inlined *ECMA-SL* program back to *JavaScript* using our shallow embedder. This compilation pipeline is illustrated in the figure below.

Structure of the thesis This document is organized as follows. Chapter 2 presents an overview of the secure information flow literature, focusing on information flow monitors and information flow analyses for *JavaScript*. Chapter 3 introduces the *ECMA-SL* intermediate language, explaining its syntax and semantics. Chapter 4 presents our information flow analyses for *ECMA-SL*, consisting of a security monitor and an inlining compiler. Chapter 5 presents both of the embedders that we developed for running *ECMA-SL* in *JavaScript*. Chapter 6 presents the evaluation of the proposed infrastructure. Finally, Chapter 7 draws some conclusions about our work and points out some future research directions.

Chapter 2

Related Work

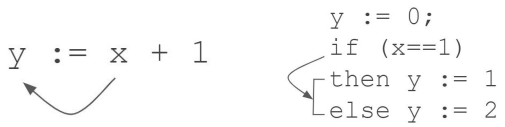
2.1 Secure Information Flow

Programs manipulate information to complete the task at hand. This manipulation creates dependencies between resources. By resources, we mean any entity that can store information and be manipulated by the program, such as program variables, object properties in JavaScript, pointers in C, and references in OCaml. Understanding these dependencies is fundamental to check whether or not an execution is secure.

Information flow security (34; 35) focuses on two main properties: confidentiality and integrity. The former is related to the disclosure of unauthorized data, that is, users without sufficient permissions should not be able to access private data. While the latter is related to the ability to change existing data, that is, untrusted users should not be able to modify critical/trusted data. More concretely, confidentiality mandates that public outputs cannot depend on private inputs, while integrity mandates that high integrity sinks cannot depend on low integrity sources. To avoid clutter, in the remainder of the document we focus on confidentiality. All results apply trivially to integrity.

In order to specify an information flow policy, the system's designer needs to assign security levels to data resources and specify the relation between these levels. Although normally expressed as a complex lattice (following (36)), for the sake of simplicity, we consider a simple two-point lattice: private (*high level*, *H*) and public (*low level*, *L*). This lattice captures a flow relation $\leq$, with $L \leq H$ and $H \not\leq L$, meaning that information may flow from L-labelled resources to H-labelled resources. The results presented in this document generalize trivially to arbitrary lattices.

Given an information flow policy described by a lattice and a security labelling, we distinguish two forms of flows: *explicit* and *implicit* (37).



(a) Explicit flow (b) Implicit flow

Figure 2.1: Types of flows

As illustrated in Figure 2.1 (a), *explicit flows* occur when there is an assignment. For instance, in the example above, there is an *explicit flow* between `y` and `x`, because the final value of `y` depends on the initial value of `x` via an assignment. On the other hand, *implicit flows* are created due to the control

structure of the program. For instance, in Figure 2.1 (b), the final value of y depends on the initial value of x via an *if* statement.

2.1.1 Non-Interference

Secure information flow can be defined as a mathematical property called *non-interference* (15). This property states that a program is safe if, during its execution, there is no propagation of private information into public resources; put formally, we say that a statement S is non-interferent if and only if, for all stores ρ_1 and ρ_2, the following holds:

$$\rho_1 \sim_L^\Gamma \rho_2 \ \wedge \ \rho_1, S \Downarrow \rho_1', \omega_1 \ \wedge \ \rho_2, S \Downarrow \rho_2', \omega_2 \Rightarrow \rho_1' \sim_L^\Gamma \rho_2' \ \wedge \ \omega_1 \sim_L^\Gamma \omega_2$$

where: Γ is a security labeling mapping program variables to security levels, ρ_1 and ρ_2 denote the two initial stores, ρ_1' and ρ_2' denote the final stores, and ω_1 and ω_2 denote the two output streams. The low equality $\rho_1 \sim_L^\Gamma \rho_2$ states that the initial stores coincide in their low projections, *mutatis mutandis* for output streams $\omega_1 \sim_L^\Gamma \omega_2$. Informally, non-interference mandates that the execution of a statement S in two stores ρ_1 and ρ_2, that coincide in their low projections, produce two stores ρ_1' and ρ_2', that also coincide in their low projections. Figure 2.2 shows an example of a non-interferent execution of a given statement S with respect to the security labeling $\Gamma = [x \mapsto H, y \mapsto L]$. Given a security labeling Γ, we define the set $NI(\Gamma)$ to be the set of programs that are non-interferent with respect to Γ.

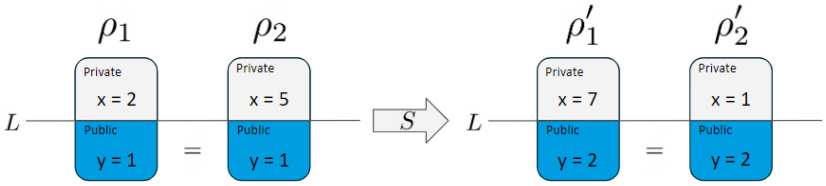


Figure 2.2: Non-interferent executions for $\Gamma = [x \mapsto H, y \mapsto L]$

2.1.2 Information Flow Bugs

One of our goals is to automatically identify information flow bugs. Here, we define a information flow bug as a pair of initial stores that whiteness a violation of non-interference; put formally, an information flow bug is a pair of initial stores (ρ_1, ρ_2), such that:

$$\rho_1 \sim_\Gamma^L \rho_2 \ \wedge \ \rho_1, S \Downarrow \rho_1', \omega_1 \ \wedge \ \rho_2, S \Downarrow \rho_2', \omega_2 \Rightarrow \rho_1' \not\sim_\Gamma^L \rho_2' \ \vee \ \omega_1 \not\sim_\Gamma^L \omega_2$$

Let us consider the programs presented in Figure 2.3. Program (a) contains an *explicit* leak and program (b) contains an implicit leak, and both violate non-interference. Hence, for each program, we should be able to find an information flow bug that witnesses the corresponding violation. In both cases, a possible information flow bug is captured by the stores $\rho_1 = [xH \mapsto 1]$ and $\rho_2 = [xH \mapsto 2]$.

(a) Explicit leak (b) Implicit leak

Figure 2.3: Types of leaks

2.2 Information Flow Monitoring

Information-flow security theory turns out to be useless in the real world if there are no enforcement mechanisms to apply it. These mechanisms can be divided in three broad classes: static, dynamic, and hybrid.

Static Enforcement Mechanisms. Static information flow analysis is mainly focused on guaranteeing the absence of information flow bugs, meaning that the analyzed program behaves securely for every possible input. Static analyses take place at static time, meaning that the program is analyzed before being executed. Examples of this type of analysis are:

- **Type-Systems** (e.g. (6; 7; 8)) require that the program resources be annotated with information flow types and check if the operations performed by the program are consistent with the supplied information flow types.
- **Abstract interpreters** (e.g. (38; 39; 22; 40)) require that the program resources be represented as an abstract domain connected with the concrete domain via a Galois connection (41; 42), and execute the program abstractly to guarantee that no insecure operations can take place concretely.
- **Self-Composition** (43) is a technique for reducing the verification of hyper-properties, such as non-interference, to the verification of simpler trace properties through the transformation of the given program into its *self-composition*. This technique is particularly interesting as it allows for the application of out-of-the-box program analysis tools, such as symbolic execution (44; 45) and separation-logic-based verification (46; 47), to the verification of secure information flow.
- **Relational logics** (e.g. (48; 49; 18)) reason about two executions of a program or of different programs on different stores. Relational logics can be used to encode a wide range of security properties, of which classical non-interference is just an example.

Dynamic Enforcement Mechanisms. Dynamic information flow analyzes operate at run-time, that is, the analyzer executes together with the program to be analyzed and has access to its run-time states. This type of analysis has special interest due to the dynamic nature of some widely used programming languages such as JavaScript, Python, and PHP. The following techniques stand out:

- **Run-time monitoring** (e.g. (9; 50; 51)), involves the creation of a component called *monitor* that runs in parallel with the *interpreter*. For each operation performed, the *interpreter* "asks" the monitor if it is safe to run it.
- **Inlining compilers** (e.g. (12; 52)) are an alternative to run-time monitors with equivalent results. This method extends the program to be analyzed with the additional monitoring operations that are meant to block insecure information flows; in other words, an inlining compiler simply inlines an

information flow monitor in the program being compiled. This solution avoids the need for changing the run-time implementation of the language.

Hybrid Enforcement Mechanisms. Hybrid methods combine both approaches. The following techniques stand out:

- **Hybrid monitors** (e.g. (53; 54; 14)) are extensions of dynamic run-time monitors that pursue a more precise analysis. This method statically examines the unengaged branches, determining which resources are updated while analyzing active branches using dynamic analysis approaches.
- **Hybrid type-systems** (e.g. (55; 56)) combine static and dynamic typing in order to avoid rejecting programs due to imprecise typing information. Program regions that cannot be precisely typed statically are dynamically checked using standard monitoring techniques.

2.2.1 Lock-step Monitors

A lock-step information flow monitor is a security mechanism that enforces non-interference, running in lock-step with the targeted language semantics to prevent illegal information flows. The idea is to *"monitor"* the execution of the input program by checking if each operation can be safely executed before its actual execution. To this end, the monitor keeps an internal representation of the security levels of the resources handled by the given program.

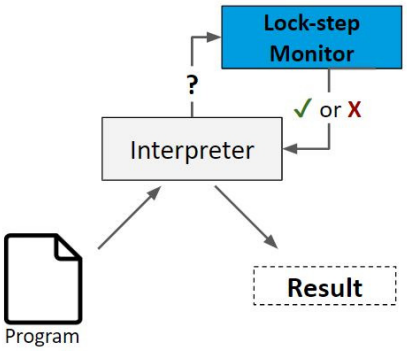


Figure 2.4: Lock-step monitor architecture

We refer to this monitor as a lock-step monitor given that for each operation, the interpreter *"asks"* the lock-step monitor whether or not the operation is secure. If it is secure, the interpreter is allowed to execute it; otherwise, the execution is aborted. This process ends with one of two alternatives: either the execution of the program is completed, meaning that the execution did not violate the information flow policy enforced by the monitor, or it is aborted due to a potential illegal flow.

There are various types of information flow monitors, ranging from purely-dynamic (9; 51; 50) to hybrid (53; 14; 54). Keep in mind that different types of monitors may use different security labelling schemes. Below, we describe the main types of information-flow monitors proposed in the literature:

- **No-sensitive-upgrade** (NSU) (9; 32) forbids sensitive upgrades, i.e., does not allow the update of public (low) resources under private (high) contexts.
- **Permissive Upgrade** (PU) (10; 57) allows sensitive upgrades to take place, but marks the resources upgraded in sensitive contexts, forbidding the program to branch depending on the content of these resources.

- **Multi-Facet** (MF) (11; 58) simulates multiple executions for different security levels by keeping, for each level, a potentially different facet of the same value. This strategy allows values to appear differently for observers at different levels.
- **Postmortem Analysis** (PA) (59) utilizes termination core dumps to analyze the execution of the program at hand.
- **Secure Multi-Execution** (SME) (60; 61) executes a program one time per security level, applying special restrictions to input/output operations for each concrete execution. In summary: (1) output operations only take place in the concrete executions with their respective security levels, being otherwise ignored; (2) high-level input operations are skipped by low-level executions, which get a default value of the appropriate type instead; and (3) low-level input operations are stalled in high-level executions, which have to wait for their respective values to be computed in the corresponding low-level executions.

Another point worth mentioning is that the *output* command can have distinct behaviors when executing over an illegal operation, such as displaying a default value or blocking the execution.

2.2.2 Monitor Inlining

As shown in Figure 2.5, an alternative to a lock-step monitor is to inline the information flow monitor in the program to be executed with the help of a dedicated compiler, typically called an *inlining compiler*. The general idea behind monitor inlining, as originally presented in (12), is to inline the monitoring logic into the program itself, effectively delegating to the language interpreter the enforcement of the information flow policy. This means that we do not have to modify the original implementation of our programming language to enforce secure information flow. Consider for instance JavaScript programs executing in the browser. If we were to implement the monitor directly on top of the browser, we would have to extend all the different JavaScript engines (e.g. Blink (62), SpiderMonkey (63), WebKit (64), and V8 (65)) with support for information flow tracking. Instead, if we inline the monitor in the programs to be executed, we can run those programs securely in all available browsers.

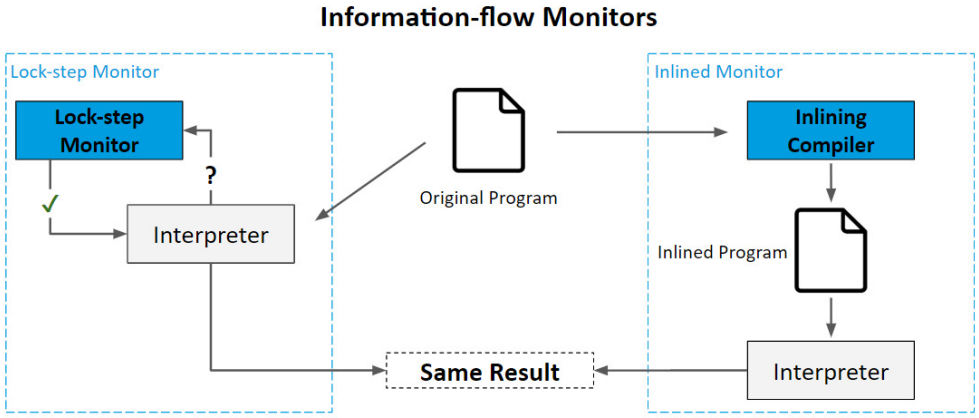


Figure 2.5: Comparison between Lock-step and Inlined Monitor Architectures

The basic idea behind an inlining compiler is to add a new shadow variable for each program variable of the original program. Shadow variables are used to keep track of the security levels of their corresponding original variables. For instance, given a variable x, we denote by $\hat{x}$ the shadow variable that is used to store the security level of x. Additionally, an inlining compiler must keep track of the level of

the current context in a dedicated variable pc. For an inlining compiler to work, one has to extend the language with security levels and level-related operators; for instance, the operators $\sqsubseteq$ and $\sqcup$ to compare security levels and compute the least-upper-bound between two security levels.

While the original work on information flow inlining compilers for JavaScript makes use of shadow variables, more recent work has been using an alternative technique which we will refer to as *wrapper objects* (26; 66). In a nutshell, instead of adding a new shadow variable for each variable of the original program, one simply wraps each runtime value inside an object that contains both the original value and its respective security label. Naturally, for this approach to work, the generated code must unwrap security labeled values before applying any primitive operator of the language and re-wrap them again with the appropriate label. For instance, the expression $x + y$ is compiled to something like:

$$wrap(getValue(x) + getValue(y), lub(getLevel(x), getLevel(y))$$

2.2.3 Comparing the Different Approaches

Inlining compilers and monitors have relative strengths and weaknesses. Inlining compilers are easier to implement, do not require changing the language run-time, and are orthogonal to different language implementations (vide JavaScript). Information flow monitors are more performant and easier to reason about.

Both strategies have serious drawbacks. They have both many false positives and they impose a serious performance degradation (67). For instance, using the Unix time command to measure one run of the benchmark, the inlining compiler presented in (26) has a 15.6x slowdown, while the JSFlow (27) is 1680x slower than the original. These limitations are so serious that they have not been applied in practice.

2.3 Information Flow Control in JavaScript

As stated in the Introduction, the dynamic nature of JavaScript renders it a hard target for static analyses. Hence, most of the information flow analyses for JavaScript proposed so far are based on dynamic approaches. In the Introduction, we briefly described the inlining compiler proposed by Fragoso Santos and Rezk (25), the JEST inlining compiler by Chudnov and Naumann (26), and the JSFlow engine proposed by Hedin et al. (27). These three analyses were based on the seminal work of Sabelfeld and Hedin (68), who were the first to propose a dynamic information flow monitor for JavaScript. Their monitor follows the no-sensitive-upgrade discipline of Austin and Flanagan (9). However, the adaptation of the NSU discipline to the setting of JavaScript required the introduction of language-specific security labels to deal with the dynamic creation and deletion of object properties. More concretely, the authors of (68) were the first to associate two security levels with each object property: one security level for the value of the property, called *value level*, and one security level for the existence of the property, called *existence level*. Given an object o and a property p, these two security levels allow one to differentiate the three following scenarios:

- *Low existence level and low-value level*: The attacker may observe the existence of the property and its value;
- *Low existence level and high-value level*: The attacker may observe the existence of the property but not its value;

- *High existence level and high-value level*: The attacker may neither observe the existence of the property nor its value.

The authors associate a further security level with every object, called the *structure security level*. In a nutshell, the structure security level of an object is an upper bound on the existing levels of its properties, meaning that properties can only be added to or removed from the object in contexts whose levels are lower than or equal to its structure security level. In other words, the domain of an object with a low structure level is only allowed to change in low-level contexts.

The labeling scheme introduced by Hedin and Sabelfeld to track information flow in JavaScript has been central to the subsequent information flow analyses and it is the one we use in our work. In the following, we will briefly review several recent research efforts on information flow analysis for JavaScript. We will not cover taint analyses for JavaScript (69; 70; 71; 72; 73; 74; 75; 76) as there is a wide variety of such analyses in the literature and their goals are substantially different from ours in that we want our analysis to verify the non-interference property, which requires controlling implicit flows. In contrast, taint analyses usually ignore implicit flows, focusing only on explicit data dependencies between sources and sinks.

Before proceeding to the description of the main existing tools for controlling information flows in JavaScript, we will quickly establish some classification criteria to guide our analysis of these tools. As stated before, there are two main approaches to implementing an information flow monitor: either one instruments an existing interpreter with the monitoring logic, or one inlines the monitoring logic directly into the program to be executed. When it comes to information flow analysis for JavaScript, some tools opt for the first approach (27; 77; 11), while others opt for the second (66; 78; 26; 79). Furthermore, different tools implement different types of monitors; while most of the existing information flow monitors for JavaScript follow the no-sensitive-upgrade (NSU) discipline (27; 79), there are also permissive-upgrade (PU) monitors (77; 80; 81; 66), multi-faceted values (MF) monitors (11), hybrid (H) monitors (78), and secure multi-execution (SME) monitors (60; 61; 82). Focusing only on information flow inlining compilers for JavaScript, they can be further divided into two groups: those that use wrapper objects (26; 66) and those that use shadow properties/variables (78; 79).

Finally, Table 2.1 classifies the existing tools for controlling information flow in JavaScript according to the criteria just described: instrumented interpreter vs. inlining compiler, type of information flow monitor (NSU, PU, MF, SME, H), and, if applicable, wrapper objects vs. shadow properties.

2.3.1 Information Flow Control Tools for JavaScript

IF-Transpiler The IF-Transpiler (78) is the first inlining compiler for JavaScript based on a hybrid monitor. Hybrid monitors must raise the security levels of the resources that might have been updated in the branches not taken by the execution. To this end, IF-Transpiler comes with a static analysis to determine an upper bound on the resources that can be modified in every program branch. However, this analysis is not presented in the paper and its soundness is not discussed, thus compromising the soundness of the entire approach.

To evaluate their compiler, the authors created two benchmarks: a functional benchmark consisting of twenty-five JavaScript programs with subtle information flow leaks, and a performance benchmark consisting of nine computationally intensive algorithms, including JavaScript implementations of SHA, MD5, and FT.

GIFC The GIFC tool (66) is the first inlining compiler for JavaScript that follows the permissive upgrade discipline (10; 57). This discipline differs from the no-sensitive-upgrade discipline in that it is more permissive. More concretely, it allows for the assignment of low-level resources under high-level contexts,

marking those resources with a dedicated taint label. Tainted resources cannot be output by the system or used to determine the control flow of the program.

The GIFC inlining compiler supports a considerable fragment of the JavaScript semantics, as well as a subset of the DOM API and some JavaScript built-in functions. Furthermore, GIFC comes with a general mechanism for associating information flow signatures with API functions, following the methodology first proposed in (83; 79). Hence, GIFC can be modularly extended with support for more DOM functions and built-in JavaScript functions.

In their evaluation of GIFC, the authors try to demonstrate that it can be used to find information flow bugs in client-side JavaScript applications and that its performance is comparable with that of its main competitors. To this end, the authors used the two benchmarks that come with the IF-Transpiler tool (78). They also extended the functional benchmark with five additional test cases specifically aimed at the implemented DOM features. The authors conclude that GIFC is more permissive than its main competitors and exhibits comparable performance.

WebPol Bichhawat et al. (80) instrumented the WebKit JavaScript engine with support for tracking information flow labels. Their instrumentation combines the permissive-upgrade discipline of Austin and Flanagan (10) with a classical post-dominator analysis used to determine the merging points of program branches. Later, the authors extended their instrumentation with support for a large fragment of the DOM API, including DOM events. More recently, the authors developed WebPol (77). WebPol is a policy framework that allows webpage developers to control the flows of information within their pages. WebPol extends the authors' previous work (80; 81) with a fine-grained policy specification component, which the authors use to enforce secure information flow in two real-world security-sensitive web applications.

ZaphodFacets ZaphodFacets (11; 84) is an instrumented JavaScript interpreter that uses the multi-faceted values approach to enforce secure information flow in JavaScript. ZaphodFacets is implemented on top of the Narcissus JavaScript engine (85), extending it with support for multi-faceted execution. In a nutshell, the multi-faceted approach mandates that one keeps multiple versions of the same value, each corresponding to a given security level. The multi-faceted approach is more permissive than both the NSU and PU approaches, since it does not cause the execution to halt due to implicit information flow leaks. Instead, it keeps various versions of the same value so that low-level observers cannot see the changes that might have occurred inside high-level contexts (they see the original version of the value and not the changed one). ZaphodFacets supports a large fragment of the JavaScript language, including exceptions. Furthermore, it also supports the DOM API by leveraging `dom.js`, a concrete implementation of the DOM, and connecting it to the underlying real DOM implementation through the use of listeners and special hooks. As most IFC mechanisms for JavaScript are based on interpreter instrumentation, ZaphodFacets exhibits prohibitive performance due to the additional interpretation layer.

FlowFox FlowFox (82) is an IFC mechanism for client-side JavaScript applications based on the secure multi-execution strategy (60; 61). It was implemented on top of the Mozilla Firefox browser and evaluated on a benchmark comprised of various Alexa top-500 websites. Recall that the core idea behind the SME strategy is to execute a program one time per security level, applying special restrictions to input/output operations for each concrete execution. In the context of the browser, these input/output operations correspond to calls to the browser API, which have to be instrumented to enforce the SME constraints. FlowFox was the first IFC mechanism capable of handling real-world client-side JavaScript programs with an acceptable performance degradation (around 20%). However, this approach has important limitations: it does not scale to security lattices composed of multiple security levels, and it may

Monitor Type	Interpreter Instrumentation	Inlining Compilers	
		Shadow Properties/Variables	Wrapper Objects
Hybrid	IF-Transpiler (78)		
No-Sensitive-Upgrade	JSFlow (27)	Fragoso Santos & Rezk (79)	Jest (26)
Permissive Upgrade	WebPol (77; 80; 81)		GIFC (66)
Multi-Facet	ZaphodFacets (11)		
SME	FlowFox (82)		

Table 2.1: Information Flow Control for JavaScript: Existing Tools

generate application crashes when there are information flow leaks.

Summary In this section, we have seen two approaches to control information flows in client-side JavaScript programs: either one instruments the browser or one inlines the control mechanism into the given programs via an inlining compiler. The first approach is tied to a specific version of a browser and is, therefore, more difficult to maintain. In contrast, the second approach is general and easier to maintain but often less precise. Engineering precise inlining compilers for JavaScript is a challenging task due to the complexity of the JavaScript semantics, which is full of implicit behaviors that can be leveraged to encode sophisticated information leaks. In contrast to all the existing information flow inlining compilers for JavaScript, our inlining compiler is the only one that is based on a precise operational model of the JavaScript semantics in the form of JS2ECMA-SL compiler (86). By using this compiler, we are guaranteed to capture all the implicit flows present in the JavaScript semantics. However, this comes at the cost of performance (ECMA-SL programs are three orders of magnitude larger than their JavaScript counterparts). Hence, the goal of this project is not to provide an alternative dynamic mechanism for securing information flows in real-world JavaScript applications. Instead, our goal is for our inlining compiler to be used as a sanity check for other tools specifically designed to be performant. In particular, the results of these tools can be compared against the results of our inlining compiler to uncover existing implementation bugs.

Chapter 3

ECMA-SL

ECMA-SL is an intermediate language for specifying the *JavaScript* standard and reasoning about *JavaScript* programs. The *ECMA-SL* project comes with a thoroughly tested compiler from *JavaScript* to *ECMA-SL*. Using this compiler, one can design new program analyses for *JavaScript* by targeting *ECMA-SL* instead of the entire *JavaScript* language. In the remainder of this chapter, we will explain both the syntax and the semantics of *ECMA-SL*, which we will use throughout this document.

3.1 Overview

ECMA-SL is a dedicated intermediate language for *JavaScript* analysis and specification. Designed as part of a larger research project, *ECMA-SL* has a core version, which was developed in the context of this thesis, and an extended version, which was used to implement a new JavaScript interpreter that follows the text of the JavaScript standard line-by-line. Core *ECMA-SL* is a strict fragment of Extended ECMA-SL, specifically designed to be the target of new static analyses for *JavaScript*. The ECMA-SL compilation pipeline works by first compiling JavaScript to Extended ECMA-SL and only then Extended ECMA-SL to Core ECMA-SL. In this chapter, we focus on the syntax and semantics of Core ECMA-SL. More details about the compilation pipeline can be found in (86).

In summary, the *ECMA-SL* project comes with:

- A compiler from *JavaScript* to Extended *ECMA-SL* written in *JavaScript*;
- A compiler from Extended *ECMA-SL* to Core *ECMA-SL*;
- A Core *ECMA-SL* interpreter written in *OCaml* (developed in the context of this thesis);
- A Shallow Embbeder of *ECMA-SL* into *JavaScript* (developed in the context of this thesis).

3.2 Core ECMA-SL

ECMA-SL is a simple imperative language with top-level functions and commands for operating on extensible objects. Importantly, it supports the core dynamic features of JavaScript: extensible objects, dynamic field access, and deletion, dynamic function calls, and runtime code evaluation. The Core *ECMA-SL* language is composed of the syntactic categories described in the table below, which comprise: statements $st \in \mathcal{S}t$, expressions $e \in \mathcal{E}$, variables $x \in \mathcal{X}$, and values $v \in \mathcal{V}$. Values include integers $i \in \mathcal{I}$, floats $f \in \mathcal{F}$, booleans $b \in \mathcal{B}$, strings $s \in \mathcal{S}$, types $\tau \in \mathcal{TY}$, locations $l \in \mathcal{L}$, and symbols $sy \in \mathcal{SY}$. Expressions include values, program variables, and a variety of unary and binary operators.

Finally, statements include both the usual imperative statements used to manage the variable store and program control-flow (variable assignment, skip, while, sequence, conditional statement, function call, and return), as well as various non-control-flow statements that provide the machinery for interacting with ECMA-SL objects (object creation, dynamic field access, dynamic field assignment, field deletion, and field collection).

Following well-established approaches (87; 14), the ECMA-SL semantics communicates with the information flow monitor that will be introduced in the next chapter via semantic labels $o \in \mathcal{O}$. In a nutshell, a semantic label is a run-time value that carries the information required by the monitor to control the information flows of the executing program. These labels will be explained in more detail later in this chapter.

Syntax of the ECMA-SL Language

Integers: $i \in \mathcal{I}$ Floats: $f \in \mathcal{F}$ Booleans: $b \in \mathcal{B}$

Vars: $x \in \mathcal{X}$ Strings: $s \in \mathcal{S}$ Types: $\tau \in \mathcal{TY}$

Locs: $l \in \mathcal{L}$ Symbol: $sy \in \mathcal{SY}$

Values: $v \in \mathcal{V}$:= $i \mid f \mid s \mid b \mid l \mid [v_1, \ ..., \ v_n] \mid \tau \mid (v_1, \ ..., \ v_n) \mid$ void $\mid$ null $\mid sy$

Expressions: $e \in \mathcal{E}$:= $v \mid x \mid \ominus e \mid e_1 \oplus e_2 \mid \otimes(e_1, \ ..., \ e_n)$

Statements: $st \in \mathcal{S}t$:= $st_1; st_2 \mid$ skip $\mid$ merge $\mid$ fail $(e) \mid$ return $(e) \mid$
if (e) then $\{st_1\}$ else $\{st_2\} \mid$ while (e) do $\{st\} \mid$
$x := e \mid x := e(e_1, \ ..., \ e_n) \mid x := \{\} \mid x := e_1$ in $e_2 \mid$
$e_1[e_2] := e_3 \mid$ delete $e_1[e_2] \mid x := e_1[e_2] \mid$
$x := e@(e_1, \ ..., \ e_n) \mid x :=$ fields e

Monitor Labels: $o \in \mathcal{O}$:= BranchLab $(e, b) \mid$ AssignLab $(x, e) \mid$ MergeLab $() \mid$
AssignCallLab $([y_k \mid_{k=0}^{n}], [e_k \mid_{k=0}^{n}], x, f) \mid$ ReturnLab $(e) \mid$ AssignNewObjLab $(x, l) \mid$
UpgVarLab $(x, \sigma) \mid$ UpgObjLab $(l, e_o, \sigma) \mid$ UpgStructLab $(l, e_o, \sigma) \mid$
UpgPropValLab $(l, f, e_o, e_f, \sigma) \mid$ UpgPropExistsLab (l, f, e_o, e_f, σ)

Semantic Domains In order to define the semantics of *ECMA-SL*, we first have to introduce stores ρ, heaps h, and call stacks cs. A store ρ is a partial function mapping variables $x \in \mathcal{X}$ to values $v \in \mathcal{V}$. A heap is a function that maps pairs of locations $l \in \mathcal{L}$ and field names $f \in \mathcal{FN}$ to values $v \in \mathcal{V}$. Statements are evaluated with respect to a call stack cs. Essentially, the call stack cs keeps track of the execution context of the calling function. More concretely, when evaluating a function call, the call stack is extended with a record that saves the calling context; analogously, when evaluating a return statement, the semantics reinstates the previous execution context (that was saved in the call stack). Formally, a call stack is a list of triples of the form (x, ρ, st), where x is the variable to which the result of the executing function is to be assigned, ρ is the store of the calling function, and st is the continuation statement of the calling function; essentially, when the executing function returns, its value is assigned to x in the store ρ and the semantics proceeds with the execution of st.

Expression Evaluation Below, we define a standard big-step semantics for *ECMA-SL* expressions, writing $[\![e]\!]_\rho = v$ to denote that the evaluation of the expression e in the store ρ yields the value v.

VALUE
$$\mathsf{h}[\![v]\!]_\rho = v$$

VARIABLE
$$[\![x]\!]_\rho = \rho(x)$$

BYNARY OPERATION
$$\frac{[\![e_1]\!]_\rho = v_1 \qquad [\![e_2]\!]_\rho = v_2 \qquad \bar{\oplus}(v_1, v_2) = v}{[\![e_1 \oplus e_2]\!]_\rho = v}$$

UNARY OPERATION
$$\frac{[\![e]\!]_\rho = v_1 \qquad \bar{\ominus} v_1 = v}{[\![\ominus e]\!]_\rho = v}$$

N-ARY OPERATION
$$\frac{([\![e_k]\!]_\rho = v_k)\ |_{k=0}^{n} \qquad \bar{\otimes}(v_1, ..., v_k) = v}{[\![\otimes(e_1, ..., e_n)]\!]_\rho = v}$$

We use $\ominus$ to range over unary operators, $\oplus$ binary operators, and $\otimes$ n-ary operators. Furthermore, we refer to the semantic counterpart of a given syntactic operator by adding an overline to its symbol; for instance, we write $\bar{\oplus}$ to refer to the semantic counter-part of the binary operator $\oplus$.

Evaluation of Basic Statements Using the semantics of expressions, we define a small-step semantics for *ECMA-SL* statements in Figures 3.1 and 3.2. The semantic judgment $\{h, \rho, cs, st\} \xrightarrow{o} \{h', \rho', cs', st'\}$ means that the evaluation of the statement st on heap h, store ρ, and call stack cs, results in the heap h', store ρ', call stack cs', and continuation st'. Semantic transitions are annotated with a label o to be consumed by the information flow monitor, which we will present later. Essentially, the label o carries all the information required by the monitor for its state transition. For instance, when evaluating an Assignment, the label o will be AssignLab (x, e). Notice that the label $\bullet$ is used when no information is required by the monitor. The semantics of *ECMA-SL* statements is standard, following that of typical object calculi for reasoning about *JavaScript* (29; 31). Here, we only explain the semantic rules for variable assignments, conditional statements, function calls, field assignments, and field deletions. The remaining rules are analogous.

Variable Assignment This rule evaluates the expression e, obtaining the value v, and updates the variable x to v in the output store ρ. The generated label simply records the variable name and the assigned expression.

Conditional Statement These rules (true and false) first evaluate the test expression e in the store ρ and create the respective label o. If the test result is true, the output continuation is prefixed with the first statement st_1. Otherwise, it is prefixed with the second statement st_2. In both cases, we add a merge statement to the continuation for signaling the end of the branch to the monitor.

Function Call This rule firsts evaluates the expression e, obtaining the identifier of the function to be called, f. It then searches for that function's body st' and formal parameters $y_k\ |_{k=1}^{n}$. The parameters are used to create a new store ρ' for executing the function's body. The store ρ' maps each parameter y_k to the corresponding argument v_k, previously determined by the evaluation of each expression e_k. The current execution store ρ and the assigned variable x are saved in the call stack cs, generating an extended call stack cs'. After generating the appropriate label o, the semantics returns a state containing the new store ρ', the new call stack cs', and the next statement to be evaluated, which is the function's body st'.

Return This rule first evaluates the expression e, obtaining the value v. It then pops the most recent entry from the call stack cs, which is composed of a variable x, a store ρ', and a statement st. The

COND. STATEMENT - TRUE

$$\frac{[\![e]\!]_\rho = \text{true} \qquad o = \text{BranchLab}\,(e, \text{true}\,)}{\{h, \rho, cs, \text{if}\,(e)\ \text{then}\ \{st_1\}\ \text{else}\ \{st_2\}\} \xrightarrow{o} \{h, \rho, cs, st_1; \text{merge}\,\}}$$

COND. STATEMENT - FALSE

$$\frac{[\![e]\!]_\rho = \text{false} \qquad o = \text{BranchLab}\,(e, \text{false}\,)}{\{h, \rho, cs, \text{if}\,(e)\ \text{then}\ \{st_1\}\ \text{else}\ \{st_2\}\} \xrightarrow{o} \{h, \rho, cs, st_2; \text{merge}\,\}}$$

ASSIGN

$$\frac{[\![e]\!]_\rho = v \quad o = \text{AssignLab}\,(x, e)}{\{h, \rho, cs, x := e\} \xrightarrow{o} \{h, \rho[x \mapsto v], cs, \text{skip}\,\}}$$

EXCEPTION

$$\frac{[\![e]\!]_\rho = v}{\{h, \rho, cs, \text{fail}\,(e)\} \xrightarrow{\bullet} \lightning(v)}$$

MERGE

$$\frac{o = \text{MergeLab}\,()}{\{h, \rho, cs, \text{merge}\,\} \xrightarrow{o} \{h, \rho, cs, \text{skip}\,\}}$$

LOOP

$$\{h, \rho, cs, \text{while}\ (e)\ \text{do}\ \{st\}\} \xrightarrow{\bullet} \{h, \rho, cs, \text{if}\,(e)\ \text{then}\ \{st; \text{while}\ (e)\ \text{do}\ \{st\}\}\ \text{else}\ \{\text{skip}\,\}\}$$

ASSIGN CALL

$$\frac{\begin{array}{c}[\![e]\!]_\rho = f \qquad \text{prog}\,(f) = f(y_k\ |_{k=0}^{n})\{st'\} \qquad ([\![e_k]\!]_\rho = v_k)\ |_{k=0}^{n} \\ \rho' = [(y_k \mapsto v_k)\ |_{k=0}^{n}] \qquad cs' = \{x, \rho, st\} :: cs \\ o = \text{AssignCallLab}\,([y_k\ |_{k=0}^{n}], [e_k\ |_{k=0}^{n}], x, f)\end{array}}{\{h, \rho, cs, x\ :=\ e(e_1,\ ...,\ e_n); st\} \xrightarrow{o} \{h, \rho', cs', st''\}}$$

RETURN

$$\frac{[\![e]\!]_\rho = v \qquad cs = \{x, \rho', st\} :: cs' \qquad o = \text{ReturnLab}\,(e)}{\{h, \rho, cs, \text{return}\,(e)\} \xrightarrow{o} \{h, \rho'[x \mapsto v], cs', st\}}$$

SEQUENTIAL COMPOSITION - 1

$$\frac{st_1 \notin \mathcal{C}all \qquad \{h, \rho, cs, st_1\} \xrightarrow{o} \{h', \rho', cs, st_1'\}}{\{h, \rho, cs, st_1; st_2\} \xrightarrow{o} \{h', \rho', cs, st_1'; st_2\}}$$

SEQUENTIAL COMPOSITION - 2

$$\{h, \rho, cs, \text{skip}\,; st\} \xrightarrow{\bullet} \{h, \rho, cs, st\}$$

ASSIGN INTERCEPTED CALL

$$\frac{[\![e]\!]_\rho = f \qquad ([\![e_k]\!]_\rho = v_k)\ |_{k=0}^{n} \qquad o = \text{getLab}\,(f, x, e, [e_k\ |_{k=0}^{n}], [v_k\ |_{k=0}^{n}])}{\{h, \rho, cs, x\ :=\ e@(e_1,\ ...,\ e_n)\} \xrightarrow{o} \{h, \rho, cs, \text{skip}\,\}}$$

Figure 3.1: Semantic of Basic Statemets: $\{h, \rho, cs, st\} \xrightarrow{o} \{h', \rho', cs', st'\}$

OBJECT CREATION

$$\frac{l \notin locs(h) \qquad o = \mathsf{AssignNewObjLab}\,(x, l)}{\{h, \rho, cs, x \ := \ \{\}\} \xrightarrow{o} \{h, \rho[x \mapsto l], cs, \mathsf{skip}\,\}}$$

FIELD ASSIGN

$$\frac{h = h' \uplus (l, f) \mapsto - \qquad \begin{array}{c}[\![e_1]\!]_\rho = l \qquad [\![e_2]\!]_\rho = f \qquad [\![e_3]\!]_\rho = v \\ h'' = h' \uplus (l, f) \mapsto v \qquad o = \mathsf{FieldAssignLab}\,(l, f, e_1, e_2, e_3)\end{array}}{\{h, \rho, cs, e_1[e_2] \ := \ e_3\} \xrightarrow{o} \{h'', \rho, cs, \mathsf{skip}\,\}}$$

FIELD LOOKUP

$$\frac{[\![e_1]\!]_\rho = l \qquad [\![e_2]\!]_\rho = f \qquad h = (l, f) \mapsto v \uplus - \qquad o = \mathsf{FieldLookupLab}\,(x, l, f, e_1, e_2)}{\{h, \rho, cs, x \ := \ e_1[e_2]\} \xrightarrow{o} \{h, \rho[x \mapsto v], cs, \mathsf{skip}\,\}}$$

FIELD DELETE

$$\frac{[\![e_1]\!]_\rho = l \qquad [\![e_2]\!]_\rho = f \qquad h = h' \uplus (l, f) \mapsto - \qquad o = \mathsf{FieldDeleteLab}\,(l, f, e_1, e_2)}{\{h, \rho, cs, \mathsf{delete}\ e_1[e_2]\} \xrightarrow{o} \{h', \rho, cs, \mathsf{skip}\,\}}$$

OBJECT HAS FIELD CHECK - TRUE

$$\frac{[\![e_1]\!]_\rho = l \qquad [\![e_2]\!]_\rho = f \qquad (l, f) \in dom(h) \qquad o = \mathsf{AssignInObjCheckLab}\,(x, f, l, e_2, e_1)}{\{h, \rho, cs, x \ := \ e_1 \ \mathsf{in} \ e_2\} \xrightarrow{o} \{h, \rho[x \mapsto \mathsf{true}\,], cs, \mathsf{skip}\,\}}$$

OBJECT HAS FIELD CHECK - FALSE

$$\frac{[\![e_1]\!]_\rho = l \qquad [\![e_2]\!]_\rho = f \qquad (l, f) \notin dom(h) \qquad o = \mathsf{AssignInObjCheckLab}\,(x, f, l, e_2, e_1)}{\{h, \rho, cs, x \ := \ e_1 \ \mathsf{in} \ e_2\} \xrightarrow{o} \{h, \rho[x \mapsto \mathsf{false}\,], cs, \mathsf{skip}\,\}}$$

OBJECT'S FIELDS

$$\frac{[\![e]\!]_\rho = l \qquad v = \mathtt{field}(h, l) \qquad o = \mathsf{AssignLab}\,(x, e)}{\{h, \rho, cs, x \ := \mathsf{fields}\,e\} \xrightarrow{o} \{h, \rho[x \mapsto v], cs, \mathsf{skip}\,\}}$$

Figure 3.2: Semantic of Heap Statements: $\{h, \rho, cs, st\} \xrightarrow{o} \{h', \rho', cs', st'\}$

variable x is then updated to v in the store ρ'. After generating the appropriate label o, the semantics returns a state containing the retrieved store ρ', the remaining call stack cs', and the next statement to be evaluated st.

Field Assignment This rule first evaluates the expressions e_1, e_2, and e_3, obtaining the location of the object l, the name of the field f, and the value v to be assigned, respectively. Then, the value v is assigned to the pair (l,f) in the heap h and the respective label o is created.

Field Deletion This rule first evaluates the expressions e_1 and e_2 obtaining the location of the object l and the name of the field to be deleted f. If the pair (l,f) exists in the heap h, the object's field is deleted. Finally, the respective label o is created.

3.2.1 Example

To better understand how the semantics of *ECMA-SL* works, consider Figures 3.3, 3.4, and 3.5, which illustrate the behavior of *ECMA-SL* stores ρ, heaps h, and call stacks cs. Each box represents an execution context at a given execution point and the transitions between boxes correspond to the small-step transitions of the semantics. Note that we annotate each transition with the label o to be consumed by the monitor. On the right-hand side of each code snippet, one can find various colored lines, each corresponding to a different statement s_i. These statements are used inside the execution contexts in the figures.

Control Flow and Continuations The point of the example given in Figure 3.3 is to show how conditional statements are interpreted. In this case, both conditional guards evaluate to $true$, extending the current continuation with the content of their then branch followed by a merge statement. For instance, the evaluation of the first conditional statement yields the continuation $s_2; merge; s_5$, where, the statement s_2 denotes the then branch of the conditional, the $merge$ statement signals the end of the branch, and the statement s_5 denotes the subsequent statements.

Heap Commands The point of the example given in Figure 3.4 is to illustrate the behaviour of the heap manipulation commands. In particular, the example creates a new object $o2$, reads the field p of the object $o1$, assigns it to $o2.q$, and finally deletes the field p of $o1$. Note that each object corresponds to a key-value map stored at its corresponding location.

Function Calls and Call Stack The point of the example given in Figure 3.5 iis to illustrate the behaviour of the call stack manipulation commands. In particular, the example calls the function f with the argument 3. Hence, the current execution state is stored in the call stack with the store ρ, a reference to the assigned variable z, and the continuation s_2. Then, a new store is generated where the parameter of the called function x is mapped to 3, and the continuation statement is set to the function's body s_3. When the return statement is reached, the previous execution state is retrieved from the call stack and the returned value 5 is assigned to z.

3.3 Implementation

The main ECMA-SL engine is implemented in OCaml (88). This engine comes with: a parser for Extended ECMA-SL, a parser for Core ECMA-SL, a compiler from Extended ECMA-SL to Core ECMA-SL,

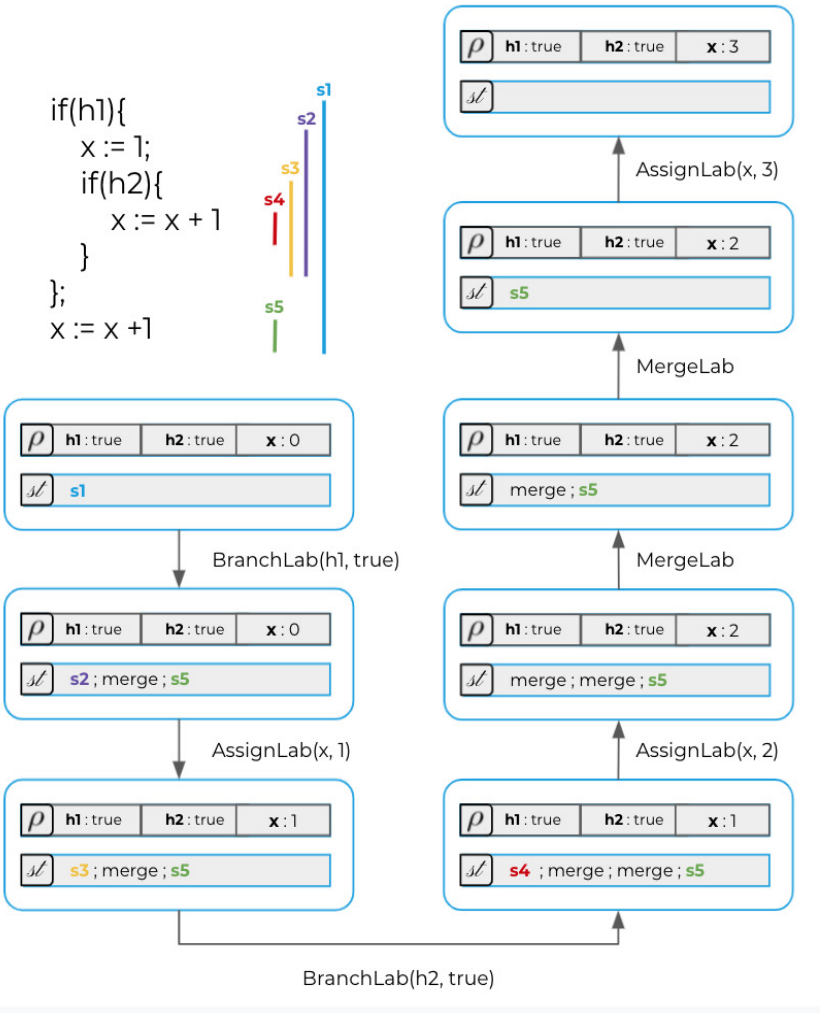


Figure 3.3: Control Flow and Continuations Example

and an interpreter for Core ECMA-SL. As part of this thesis, we have implemented the parser for Core ECMA-SL and its interpreter.

The parser of Core ECMA-SL was implemented using the menhir parser generator (89). It comprises 131 rules and 354 lines of code. Importantly, designing the parser involved solving a non-trivial number of shift-reduce and reduce-reduce conflicts (90) by establishing the appropriate precedences between the various operators of the language. For instance, the conflict produced by the binary minus operator (`1 - 2`) and the unary negative operator (`-5`).

The Core ECMA-SL interpreter consists of 423 lines of OCaml code. It lies on top of an implementation of the ECMA-SL semantic domains: store, heap, and call stack. Each semantic domain is implemented as an OCaml module and the semantics is implemented as an OCaml functor (91). We have chosen to implement the semantics as a functor to make it parametric on the security monitor to be used. This means that one can change the security monitor to be used without modifying the implementation of the semantics. To illustrate the modularity of our approach, we have implemented several monitors, connecting all of them to the same implementation of the semantics.

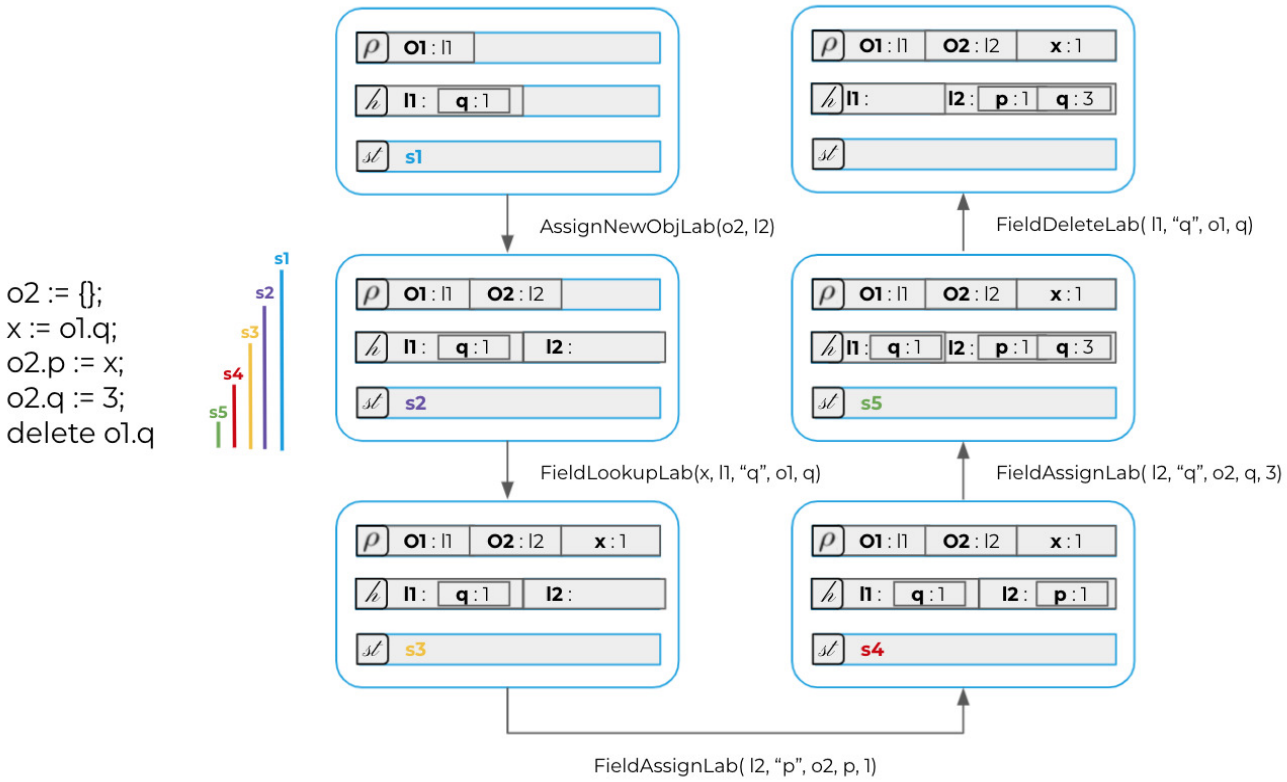


Figure 3.4: Heap Example

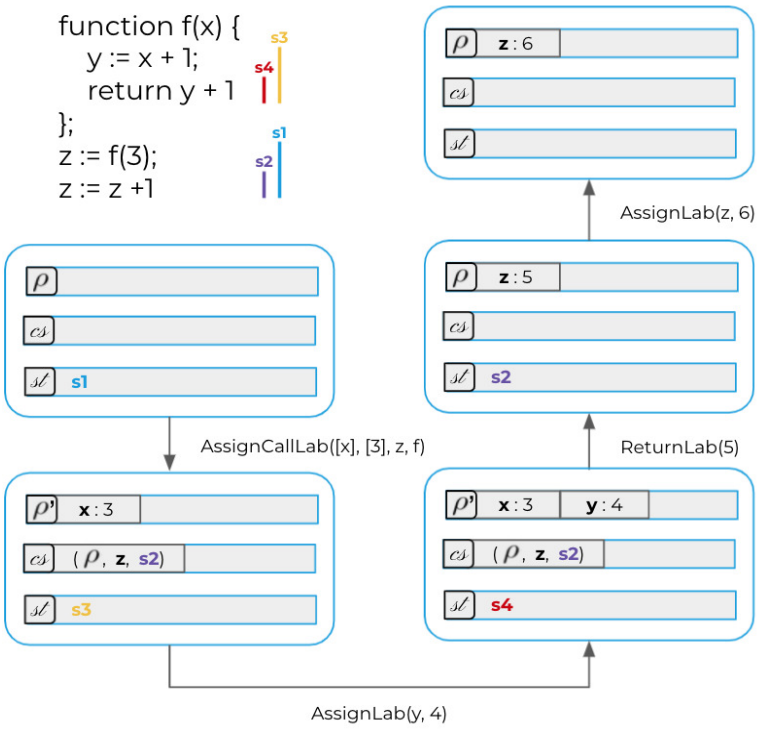


Figure 3.5: Function Calls and Call Stack Example

Chapter 4

Information Flow Security for ECMA-SL

4.1 ECMA-SL Monitor Infrastructure

In order to present the security monitor independently of the language semantics, we pair up semantic transitions with monitor transitions. We refer to the combined transition as monitored semantics. The idea is simple: the language semantics performs a single step, generating a semantic label with the relevant information, and this label is given to the security monitor for it to produce the correspondent monitoring step. The monitored semantics transition is defined in the table below:

Monitored Semantics

Monitor Configuration:	Φ
Monitored Semantics Configuration:	(Ω, Φ)
Semantic Transition:	$\Omega \xrightarrow{o} \Omega'$
Monitor Transition:	$\Phi \xrightarrow{o} \Phi'$
Monitored Semantic Transition	$\dfrac{\Omega \xrightarrow{o} \Omega' \quad \Phi \xrightarrow{o} \Phi'}{\{\Omega, \Phi\} \xrightarrow{o} \{\Omega', \Phi'\}}$

We use Φ to denote the monitor configuration and Ω to denote the language semantic configuration, forming the pair (Ω, Φ), which describes the monitored semantics configuration. Analogously to semantic transitions, monitor transitions are also annotated with a semantic label $o \in O$. Essentially, the interpreter gives a step generating the label o, and that label is then consumed by the security monitor, which performs a parallel step to that of the language semantics. Therefore, we can define the monitored semantic transition as $\{\Omega, \Phi\} \xrightarrow{o} \{\Omega', \Phi'\}$, where Ω' and Φ' are the result of applying the semantics and the monitor to Ω and Φ, respectively. In order to illustrate how this mechanism can be used to couple the semantics with an arbitrary monitor, we present two simple monitors. The permissive monitor, which accepts every execution, has a single configuration $\heartsuit$ and a single transition $\heartsuit \rightarrow \heartsuit$ that is applicable regardless of the consumed label. The obstructive monitor does not have any transition, meaning that every execution is blocked when trying to perform its first step.

Permissive Monitor

$\Phi = \heartsuit$	$\{\heartsuit \xrightarrow{o} \heartsuit \mid o \in \mathcal{O}\}$

Obstructive Monitor

$\Phi = \spadesuit$	$\emptyset$

4.2 ECMA-SL Security Domains

To define the semantics of our *ECMA-SL* security monitor, we first introduce security stores $s\rho$, security heaps sh, and security call stacks scs. A *security store* $s\rho$ is a partial function mapping variables $x \in \mathcal{X}$ to security levels $\sigma \in \mathcal{L}ev$; for instance, $s\rho(x) = \sigma_x$ means that the value stored in x can only be observed at level σ_x. A *security heap* sh is a function that maps pairs of locations $l \in \mathcal{L}$ and field names $f \in \mathcal{FN}$ to pairs of security levels; for instance, $sh(l, f) = (\sigma_1, \sigma_2)$ means that the existence of field f in object l can be seen at level σ_1, while its value can only be seen at level σ_2. In the following, we will refer to σ_1 as the *existence level* of f in l and σ_2 as its *value level*; we will further assume that the value level is always greater than or equal to the existence level ($\sigma_1 \sqsubseteq \sigma_2$). Analogously to the *ECMA-SL* semantics' call stack, the security call stack scs keeps track of the security context of the calling function. Formally, a security call stack is a list of triples of the form $\{x, s\rho, pc\}$, where x is the variable to which the result of the executing function is to be assigned, $s\rho$ is the security store of the calling function, and pc is the stack of security levels that was active when the current function was called. Essentially, when the executing function returns, its resulting security level is assigned to x in the security store $s\rho$, and the stack of security levels of the calling context, pc, is restored.

Security Domains

Security Store:	$s\rho \in \mathcal{S}Store \;::\; \mathcal{X} \rightharpoonup \mathcal{L}ev$
Security Heap:	$sh \in \mathcal{SH}eap \;::\; \mathcal{L} \times \mathcal{FN} \rightharpoonup (\mathcal{L}ev \times \mathcal{L}ev)$
Security Metadata:	$sm \in \mathcal{SM}etadata \;::\; \mathcal{L} \rightharpoonup (\mathcal{L}ev \times \mathcal{L}ev)$

In order to formally define the observational power of an attacker at a given security level, we resort to the notion of low-projection. The low-projection of a state at a given security level σ corresponds to the part of that state that an observer at level σ can see. Here, as *ECMA-SL* configurations are composed of a heap, a store, and a call stack, we define low-projections for heaps and stores. We write: $\rho \restriction^{\sigma}_{s\rho}$ for the low-projection of ρ at level σ with respect to $s\rho$ and $h \restriction^{\sigma}_{sh}$ for the low-projection of h at level σ with respect to sh. The formal definitions are given in Table 4.1.

Low-projection for Stores The low-projection of a store ρ with respect to a security store $s\rho$ at a given level σ is computed point-wise. For each x in the domain of the store, we check if its security level given by $s\rho$ is smaller than or equal to σ ($s\rho(x) \sqsubseteq \sigma$). If it is, we keep it in the low-projection; otherwise, it is simply erased.

Low-projection for Heaps The low-projection of a heap h with respect to a security heap sh at a given level σ is also computed point-wise. For each pair (l, f) in the domain of the heap, we check if both the existence level σ_1 and the value level σ_2 of f in l are smaller than or equal to σ ($\sigma_1, \sigma_2 \sqsubseteq \sigma$). If both levels are visible, the entire cell is kept in the low-projection as a σ-observer sees both the existence of the field and its value. If only the existence level is visible, the pair (l, f) is kept in the low-projection, but its value is replaced with the special symbol $?$ denoting an unknown value. Finally, if both levels are invisible, the cell is entirely excluded from the low-projection.

Example To better understand the concept of low-projection, let us consider the labeled heap given in Figure 4.1. The labeled heap contains five ECMA-SL objects. Each object is depicted as a circle containing its corresponding fields. Recall that each field is associated with two levels: the existence level and the value level. The existence level is represented on the right-hand side of each field, while the value level annotates the arrow that connects the field to its corresponding value; for instance, the

Low-Projection for Stores: $\rho \restriction^{\sigma}_{s\rho}$

EMPTY STORE
$$\varnothing \restriction^{\sigma}_{s\rho} \triangleq \varnothing$$

VISIBLE VALUE
$$\frac{s\rho = x \to \sigma' \uplus s\rho' \quad \sigma' \sqsubseteq \sigma}{(\rho \uplus x \to v) \restriction^{\sigma}_{s\rho} \triangleq x \to v \uplus \rho \restriction^{\sigma}_{s\rho'}}$$

INVISIBLE VALUE
$$\frac{s\rho = x \to \sigma' \uplus s\rho' \quad \sigma' \not\sqsubseteq \sigma}{(\rho \uplus x \to v) \restriction^{\sigma}_{s\rho} \triangleq \rho \restriction^{\sigma}_{s\rho'}}$$

Low-Projection for Heaps: $h \restriction^{\sigma}_{sh}$

EMPTY HEAP
$$\varnothing \restriction^{\sigma}_{sh} \triangleq \varnothing$$

VISIBLE CELL WITH VISIBLE VALUE
$$\frac{sh = sh' \uplus (l,p) \to (\sigma_1, \sigma_2) \quad \sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma}{(h \uplus (l,p) \to v) \restriction^{\sigma}_{sh} \triangleq (l,p) \to v \uplus h \restriction^{\sigma}_{sh'}}$$

INVISIBLE CELL WITH INVISIBLE VALUE
$$\frac{sh = sh' \uplus (l,p) \to (\sigma_1, \sigma_2) \quad \sigma_1 \not\sqsubseteq \sigma}{(h \uplus (l,p) \to v) \restriction^{\sigma}_{sh} \triangleq h \restriction^{\sigma}_{sh'}}$$

VISIBLE CELL WITH INVISIBLE VALUE
$$\frac{sh = sh' \uplus (l,p) \to (\sigma_1, \sigma_2) \qquad \sigma_1 \sqsubseteq \sigma \qquad \sigma_2 \not\sqsubseteq \sigma}{(h \uplus (l,p) \to v) \restriction^{\sigma}_{sh} \triangleq (l,p) \to ? \uplus h \restriction^{\sigma}_{sh'}}$$

Table 4.1: Low Projection for Store and Heap Domains

field **meta** of object **O1** has a low existence level and high value level. Figure 4.1 presents a labelled heap together with its low-projection at level L. The low-projection captures the parts of the heap that are visible to an observer at level L. For instance:

- the field **info** of object **O_1** has both low value level and low existence level; hence, both the field and its value are kept in then low-projection of the heap.
- the field **aux** of object **O_1** has both high value level and high existence level; hence, both the field and its value are excluded from the low-projection of the heap. Notice that, due to the low-level nature of **O_5**, it is kept in the low-projection of the heap.
- the field **meta** of object **O_1** has both high value level and high existence level; hence, both the field and its value are excluded from the low-projection of the heap.
- the field **address** of object **O_4** has a high value level but a low existence level; hence, the field is kept in the object, while its value is excluded from the low-projection of the heap.

4.3 ECMA-SL Security Monitor

This section presents a security monitor for enforcing secure information flow in ECMA-SL. The proposed monitor follows the no-sensitive-upgrade discipline (9), which mandates that no low-level resources be updated inside high-level contexts. This section first describes the application of the no-sensitive-upgrade discipline to ECMA-SL, enumerating the implicit flows that must be blocked by our monitor. Then, it presents the formalization of the monitor, carefully explaining its most important rules. Finally, it introduces an inlining compiler that implements the proposed monitor by instrumenting the given program with both label-tracking instructions and monitoring constraints.

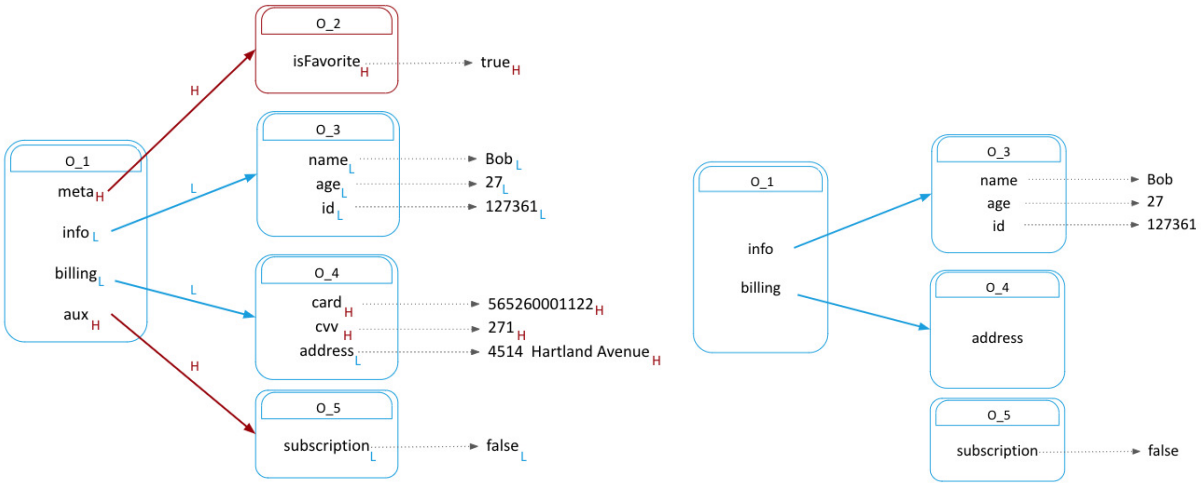


Figure 4.1: Low-projection example as a low-level observer

4.3.1 The No-sensitive-Upgrade Discipline for ECMA-SL

The no-sensitive-upgrade discipline establishes that low-level resources cannot be modified inside high-level contexts. By preventing low-level updates inside high-level contexts, the NSU discipline disallows control flow leaks from high-level resources to low-level resources. Intuitively, one could consider a naive strategy that would simply raise the security level of low-level resources updated inside high-level contexts to the level of the context itself. This strategy, however, does not work since it would partially leak the contents of the resources on which the control flow depends.

Table 4.2 illustrates the seven types of no-sensitive-upgrades which can be found in ECMA-SL programs. The given examples make use of a high-level input variable **h0** and two low-level input variables **l0** and **l1**. For each case, one can see that the initial value of **h0** flows into **l0**, justifying the need for disallowing that type of information flow. In the following, we give a short explanation of each type of no-sensitive-upgrade.

Low-Level Variable Assignment in a High-Level Context (Type I) The monitor blocks low-level assignments under high-level guards. Hence, in the example, the monitor throws an information flow exception in the assignment of **l1** inside the $then$ branch of the first *if* statement.

Low-Level Field Assignment in a High-Level Context (Type II) The monitor blocks low-level field assignments under high-level guards. Hence, in the example, the monitor throws an information flow exception when executing the assignment to the field **f0** belonging to the object **o** inside the *then* branch of the first *if* statement.

Low-Level Field Deletion in a High-Level Context (Type III) The monitor blocks low-level field deletions under high-level guards. Hence, in the example, the monitor throws an information flow exception when deleting the field **f0** belonging to the object **o** inside the *then* branch of the first *if* statement.

Low-Level Field Assignment via High-Level Object (Type IV) The monitor blocks assignments to low-level fields when the expression denoting the object depends on high-level data. Hence, in the

Type I

```
l1 := true;
l0 := true;
if (h0) then {
  l1 := false
};
if (l1) then {
  l0 := false
}
```

Type II

```
l1 := true;
o := {}^L;
o.f0 := true;
if (h0) then {
  o.f0 := false
};
if (o.f0) then {
  l0 := false
}
```

Type III

```
l1 := true;
o := {}^L;
o.f0 := true;
if (h0) then {
  delete o.f0
};
x := f0 in o;
if (x) then {
  l0 := false
}
```

Type IV

```
l1 := true;
ol := {}^L;
oh := {}^H;
ol.f0 := false;
if (h0) then {
  oh := ol
};
oh.f0 := true;
if (!ol.f0) then {
  l0 := false
}
```

Type V

```
l0 := true;
o := {}^L;
o.f0 := false;
fh := "f0";
if (h0) then {
  fh := "f1"
};
o[fh] := true;
if (!o.f1) then {
  l0 := false
}
```

Type VI

```
l0 := true;
ol := {}^L;
oh := {}^H;
ol.f0 := true;
oh.f0 := true;
if (h0) then {oh := ol};
delete oh.f0;
l2 := f0 in ol;
if (l2) then {l0 := false}
```

Type VII

```
l0 := true;
o := {}^H;
fh := "f1";
o[fh] := true;
if (h0) then {fh := "f0"};
delete o[proph];
l2 := f0 in o;
if (l2) then {l0 := false}
```

Type VIII

```
function f1(){return true};
function f2(){return false};
if (h0) then {
  fh := "f1"
};
else {
  fh := "f2"
};
l0 := fh()
```

Table 4.2: Naive Approach vs No-sensitive-upgrade

example, when **oh** is updated to the same object as **ol**, the assignment of **true** to **oh.f0** throws an information flow exception because it tries to update a low-level field through a high-level object.

Low-Level Field Assignment via a High-Level Field Name (Type V) The monitor blocks assignments to low-level fields when the field name was computed using high-level data. Hence, in the example, the assignment of **'f1"** to **fh** is allowed to go through, even though it is performed inside a high-level guard, due to the high-level nature of the variable **fh**. However, the following field assignment, **o[fh] = true**, throws an information flow exception, because it tries to update the value of a low-level field through a high-level field name.

Low-Level Field Deletion via a High-Level Object (Type VI) The monitor blocks the deletion of low-level fields when the location pointing to the object that binds the field was computed using high-level data. Hence, in the example, the high-level variable **oh** can hold both low-level locations and high-level locations. Therefore, the assignment **oh = ol** is legal. However, when **oh** is set to point to the same location as **ol**, the deletion of **oh[f0]** throws an information flow exception since it constitutes a low-level field deletion via a high-level location.

Low-Level Field Deletion via a High-Level Field Name (Type VII) The monitor blocks the deletion of a low-level field when the corresponding field name was computed using high-level data. The assignment of **f0** to **fh** is allowed under the high-level guard. However, when **fh** is set to **f0**, the deletion of **o[fh]** throws an information flow exception since it constitutes a low-level field deletion via a high-level field name.

High-Level Function Call with Low-Level Return (Type VIII) The monitor blocks low-level assignments that result from high-level function calls. Hence, in the example, the called function **fh** is determined under a high-level guard, inside the *if* statement. However, the function return is assigned to the low-level variable **l0**, throwing an information flow exception since it constitutes a low-level assignment via a high-level function call.

4.3.2 Monitor Definition

We define an information flow monitor for *ECMA-SL* statements in small-step style. The monitor transition $\{sm, sh, s\rho, pc\} \xrightarrow{o} \{sm', sh', s\rho', pc'\}$ means that the monitor step triggered by o in the security heap sh, security store $s\rho$, security call stack scs, and program counter pc generates the security heap sh', security store $s\rho'$, security call stack scs', and program counter pc'. Note that, while the monitor transition requires the label o as an input, the semantics transition generates the label o as an output.

Before proceeding to the description of the monitor, we introduce expression levels. The security level of an expression is given by the least-upper-bound between the levels of all variables that occur in it according to the following equation:

$$\mathsf{lev}(s\rho, e) = \bigsqcup \{s\rho(x) \mid x \in Vars(e)\} \tag{4.1}$$

Where $Vars(e)$ denoted the set of program variables occurring in e.

Bellow, we explain four rules of the *ECMA-SL* monitor, which illustrate how our monitor propagates security labels and enforces the no-sensitive-upgrade discipline. The complete set of rules is given in Figures 4.2 and 4.3.

Conditional Statement This rule starts by obtaining the level of the expression e, σ_e. The monitor then extends the program counter pc with the least-upper-bound between σ_e and its current context level ($lvl(pc)$). We take the least-upper-bound between σ_e and $lvl(pc)$ instead of simply σ_e to guarantee that the levels in the program counter are monotonically decreasing: higher security levels on top and lower security levels below (the level of the current execution context always corresponds to the level at the top of the pc stack). Had we not done this, we would have to traverse the entire pc stack every time we would need to obtain the level of the current context.

$$\textsc{BranchLab}\quad \frac{o = \mathsf{BranchLab}\,(e) \qquad \sigma_e = \mathsf{lev}(s\rho, e) \qquad pc' = (\sigma_e \sqcup lvl(pc)) :: pc}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh, s\rho, scs, pc'\}}$$

Function Call This rule first computes the least-upper-bound between the level of the expression that denotes the function to be called and the level of the current program counter, obtaining the level σ'_{pc}. Note that this level corresponds to the security level of the context in which the body of the function is to be executed; hence, we set the initial pc stack of the function to $[\sigma'_{pc}]$. Then, a new store is created where each argument, y_k, is mapped to the least-upper-bound between the level of its corresponding expression, $\mathsf{lev}(s\rho, e_k)$, and σ'_{pc}. Finally, the rule saves the old security configuration by pushing it onto the security call stack scs, creating a new security call stack scs'.

This rule additionally checks if the new context level σ'_{pc} is less than or equal to the level of the variable to which the return of the call will be assigned, preventing leaks of Type VIII.

$$\textsc{AssignCallLab}\quad \frac{\begin{array}{c} o = \mathsf{AssignCallLab}\,([y_k\ |_{k=0}^{n}], [e_k\ |_{k=0}^{n}], x, e_f) \qquad \sigma'_{pc} = \mathsf{lev}(s\rho, e_f) \sqcup lvl(pc) \\ \sigma'_{pc} \sqsubseteq \mathsf{lev}(s\rho, x) \qquad scs' = \{pc, s\rho, x\} :: scs \qquad s\rho' = [(y_k \mapsto \mathsf{lev}(s\rho, e_k) \sqcup \sigma'_{pc})\ |_{k=0}^{n}] \end{array}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh, s\rho', scs', [\sigma'_{pc}]\}}$$

Field Assignment (Field Update) This rule starts by obtaining the security levels of the expressions denoting the object, field, and value involved in the field assignment, respectively, σ_o, σ_f, and σ_v. Then, it obtains the context level, σ_{ctx}, by computing the least-upper-bound between σ_o, σ_f, and $lvl(pc)$. The NSU discipline mandates that the context level be less than or equal to the level of the resource being updated. Hence, we check that σ_{ctx} is smaller than or equal to the value level of the field being updated, preventing leaks of types II, IV, and V. If it is, the value level of the object's field is updated to $\sigma_v \sqcup \sigma_{ctx}$. If not, an information flow error is raised.

$$\textsc{FieldAssignLab - Field Update}\quad \frac{\begin{array}{c} o = \mathsf{FieldAssignLab}\,(l, f, e_1, e_2, e_3) \qquad \sigma_o = \mathsf{lev}(s\rho, e_1) \qquad \sigma_f = \mathsf{lev}(s\rho, e_2) \qquad \sigma_v = \mathsf{lev}(s\rho, e_3) \\ \sigma_{ctx} = \sigma_o \sqcup \sigma_f \sqcup lvl(pc) \qquad sh = sh' \uplus (l, f) \mapsto (\sigma_{exists}, \sigma_{val}) \qquad \sigma_{ctx} \sqsubseteq \sigma_{val} \\ sh'' = sh' \uplus (l, f) \mapsto (\_, \sigma_v \sqcup \sigma_{ctx}) \end{array}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh'', s\rho, scs, pc\}}$$

Field Delete This rule starts by obtaining the security levels of the expressions denoting the object and field involved in the field deletion, respectively σ_o and σ_f. Then, it obtains the context level, σ_{ctx}, by computing the least-upper-bound between $lvl(pc)$, σ_o, and σ_f. Following the NSU discipline, we check that σ_{ctx} is smaller than or equal to the existence level of the field being updated, preventing leaks of

EMPTYLAB

$$\frac{o = \text{EmptyLab}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh, s\rho, scs, pc\}}$$

ASSIGNLAB - VARIABLE UPDATE

$$\frac{o = \text{AssignLab}\,(x, e) \quad \sigma_e = \mathsf{lev}(s\rho, e) \quad \sigma_{pc} = lvl(pc) \quad \sigma_{pc} \sqsubseteq \mathsf{lev}(s\rho, x)}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh, s\rho[x \mapsto \sigma_e \sqcup \sigma_{pc}], scs, pc\}}$$

ASSIGNLAB - VARIABLE DEFINITION

$$\frac{o = \text{AssignLab}\,(x, e) \quad \sigma_e = \mathsf{lev}(s\rho, e) \quad x \notin dom(s\rho)}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh, s\rho[x \mapsto \sigma_e \sqcup lvl(pc)], scs, pc\}}$$

ASSIGNCALLLAB

$$\frac{o = \text{AssignCallLab}\,([y_k\ |_{k=0}^{n}], [e_k\ |_{k=0}^{n}], x, e_f) \quad \sigma'_{pc} = \mathsf{lev}(s\rho, e_f) \sqcup lvl(pc) \quad \sigma'_{pc} \sqsubseteq \mathsf{lev}(s\rho, x) \quad scs' = \{pc, s\rho, x\} :: scs \quad s\rho' = [(y_k \mapsto \mathsf{lev}(s\rho, e_k) \sqcup \sigma'_{pc})\ |_{k=0}^{n}]}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh, s\rho', scs', [\sigma'_{pc}]\}}$$

RETURNLAB

$$\frac{o = \text{ReturnLab}\,(e) \quad \sigma_e = \mathsf{lev}(s\rho, e) \quad \sigma_f = \sigma_e \sqcup lvl(pc) \quad scs = \{pc', s\rho', x\} :: scs' \quad \{sh, s\rho', scs', pc', \text{AssignLab}\,(x, \sigma_f)\} \xrightarrow{o} \{sh, s\rho'', scs', pc'\}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh, s\rho'', scs', pc'\}}$$

BRANCHLAB

$$\frac{o = \text{BranchLab}\,(e) \quad \sigma_e = \mathsf{lev}(s\rho, e) \quad pc' = (\sigma_e \sqcup lvl(pc)) :: pc}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh, s\rho, scs, pc'\}}$$

MERGELAB

$$\frac{o = \text{MergeLab} \quad pc' = \mathsf{pop}(pc)}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh, s\rho, scs, pc'\}}$$

Figure 4.2: Semantic of Monitor Basic Commands: $\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm', sh', s\rho', scs', pc'\}$

types III, VI, and VII. If the constraint is satisfied, the object's field is deleted; otherwise, an information flow error is raised.

FIELDDELETELAB

$$\frac{o = \text{FieldDeleteLab}\,(l, f, e_1, e_2) \quad \sigma_o = \mathsf{lev}(s\rho, e_1) \quad \sigma_f = \mathsf{lev}(s\rho, e_2) \quad \sigma_{ctx} = \sigma_o \sqcup \sigma_f \sqcup lvl(pc) \quad sh = sh' \uplus (l, f) \mapsto (\sigma_{exists}, \_) \quad \sigma_{ctx} \sqsubseteq \sigma_{exists}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh', s\rho, scs, pc\}}$$

4.3.3 Example

To better understand the inner workings of our information flow monitor, we will now illustrate its application to two simple examples. Figures 4.4 and 4.5 show the monitored execution of the Type I and Type II programs given in Table 4.2. In both cases, we represent the transitions of the semantics on the left and the transitions of the monitor on the right. Similarly to semantic execution contexts, each security context is represented as a box containing its corresponding security store $s\rho$, heap sh, call stack scs, and pc stack. Transitions between execution contexts are labeled with the respective security labels. Furthermore, we represent each semantic context together with its corresponding security context, with the semantic context on the left and security context on the right.

Type I Example The example starts with the pc stack $[L]$ indicating that the program is executing in a low context. After evaluating the guard of the conditional statement, the pc stack is extended to $[H, L]$. Then, within the scope of the conditional statement, the execution of the assignment `l1 := false` triggers an information flow exception as it constitutes a no-sensitive-upgrade. Note that the level of `l1` is L and the level of the execution context is H.

NEWOBJLAB

$$\frac{o = \text{AssignNewObjLab} \qquad \sigma_{pc} = lvl(pc) \qquad sm' = sm \uplus l \mapsto (\sigma_{pc}, \sigma_{pc})}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm', sh, s\rho[l \mapsto \sigma_{pc}], scs, pc\}}$$

FIELDLOOKUPLAB

$$\frac{\begin{array}{c} o = \text{FieldLookupLab}\,(x, l, f, e_1, e_2) \\ \sigma_o = \mathsf{lev}(s\rho, e_1) \qquad \sigma_f = \mathsf{lev}(s\rho, e_2) \qquad \sigma_{ctx} = \sigma_o \sqcup \sigma_f \sqcup lvl(pc) \\ \sigma_{ctx} \sqsubseteq \mathsf{lev}(s\rho, x) \qquad sh = sh' \uplus (l, f) \mapsto (\_, \sigma_{val}) \qquad \sigma_1 = \sigma_{ctx} \sqcup \sigma_{val} \end{array}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh, s\rho[x \mapsto \sigma_1], scs, pc\}}$$

FIELDDELETELAB

$$\frac{\begin{array}{c} o = \text{FieldDeleteLab}\,(l, f, e_1, e_2) \qquad \sigma_o = \mathsf{lev}(s\rho, e_1) \qquad \sigma_f = \mathsf{lev}(s\rho, e_2) \\ \sigma_{ctx} = \sigma_o \sqcup \sigma_f \sqcup lvl(pc) \qquad sh = sh' \uplus (l, f) \mapsto (\sigma_{exists}, \_) \qquad \sigma_{ctx} \sqsubseteq \sigma_{exists} \end{array}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh', s\rho, scs, pc\}}$$

FIELDASSIGNLAB - FIELD UPDATE

$$\frac{\begin{array}{c} o = \text{FieldAssignLab}\,(l, f, e_1, e_2, e_3) \\ \sigma_o = \mathsf{lev}(s\rho, e_1) \qquad \sigma_f = \mathsf{lev}(s\rho, e_2) \qquad \sigma_e = \mathsf{lev}(s\rho, e_3) \qquad \sigma_{ctx} = \sigma_o \sqcup \sigma_f \sqcup lvl(pc) \\ sh = sh' \uplus (l, f) \mapsto (\sigma_{exists}, \sigma_{val}) \qquad \sigma_{ctx} \sqsubseteq \sigma_{val} \qquad sh'' = sh' \uplus (l, f) \mapsto (\_, \sigma_e \sqcup \sigma_{ctx}) \end{array}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh'', s\rho, scs, pc\}}$$

FIELDASSIGNLAB - FIELD DEFINITION

$$\frac{\begin{array}{c} o = \text{FieldAssignLab}\,(l, f, e_1, e_2, e_3) \qquad \sigma_o = \mathsf{lev}(s\rho, e_1) \\ \sigma_f = \mathsf{lev}(s\rho, e_2) \qquad \sigma_e = \mathsf{lev}(s\rho, e_3) \qquad \sigma_{ctx} = \sigma_o \sqcup \sigma_f \sqcup lvl(pc) \\ (l, f) \notin dom(sh) \qquad sm = sm' \uplus loc \mapsto (\sigma_{struct}, \sigma_{obj}) \qquad \sigma_{ctx} \sqsubseteq \sigma_{struct} \qquad sh'' = sh' \uplus (l, f) \mapsto (\sigma_{ctx}, \sigma_e \sqcup \sigma_{ctx}) \end{array}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh'', s\rho, scs, pc\}}$$

ASSIGNINOBJCHECKLAB

$$\frac{\begin{array}{c} o = \text{AssignInObjCheckLab}\,(x, f, l, e_1, e_2) \qquad \sigma_l = \mathsf{lev}(s\rho, e_2) \qquad \sigma_f = \mathsf{lev}(s\rho, e_1) \\ \sigma_{ctx} = lvl(pc) \sqcup \sigma_l \sqcup \sigma_f \qquad sh = sh' \uplus (l, f) \mapsto (\sigma_{exists}, \sigma_{val}) \qquad \sigma_{ctx} \sqsubseteq \mathsf{lev}(s\rho, x) \end{array}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh, s\rho[x \mapsto \sigma_{exists} \sqcup \sigma_{ctx}], scs, pc\}}$$

UPGVARLAB

$$\frac{o = \text{UpgVarLab}\,(x, \sigma) \qquad \sigma_{pc} = lvl(pc) \qquad \sigma_{pc} \sqsubseteq \mathsf{lev}(s\rho, x) \qquad \sigma_{x'} = \sigma \sqcup \sigma_{pc}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh, s\rho[x \mapsto \sigma_{x'}], scs, pc\}}$$

UPGOBJLAB

$$\frac{\begin{array}{c} o = \text{UpgObjLab}\,(l, e_o, \sigma) \qquad \sigma_o = \mathsf{lev}(s\rho, e_o) \qquad \sigma_{ctx} = \sigma_o \sqcup lvl(pc) \qquad sm = sm' \uplus l \mapsto (\sigma_{struct}, \sigma_{obj}) \\ \sigma_{ctx} \sqsubseteq \sigma_{obj} \qquad \sigma_{obj'} = \sigma \sqcup \sigma_{ctx} \qquad sm' = sm \uplus l \mapsto (\sigma_{struct}, \sigma_{obj'}) \end{array}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm', sh, s\rho, scs, pc\}}$$

UPGSTRUCTLAB

$$\frac{\begin{array}{c} o = \text{UpgStructLab}\,(l, e_o, \sigma) \qquad \sigma_o = \mathsf{lev}(s\rho, e_o) \qquad \sigma_{ctx} = \sigma_o \sqcup lvl(pc) \qquad sm = sm' \uplus l \mapsto (\sigma_{struct}, \sigma_{obj}) \\ \sigma_{ctx} \sqsubseteq \sigma_{struct} \qquad \sigma_{struct'} = \sigma \sqcup \sigma_{ctx} \qquad sm' = sm \uplus l \mapsto (\sigma_{struct'}, \sigma_{obj}) \end{array}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm', sh, s\rho, scs, pc\}}$$

UPGPROPVALLAB

$$\frac{\begin{array}{c} o = \text{UpgPropValLab}\,(l, f, e_o, e_f, \sigma) \qquad \sigma_o = \mathsf{lev}(s\rho, e_o) \qquad \sigma_f = \mathsf{lev}(s\rho, e_f) \qquad \sigma_{ctx} = \sigma_o \sqcup \sigma_f \sqcup lvl(pc) \\ sh = sh' \uplus (l, f) \mapsto (\sigma_{exists}, \sigma_{val}) \qquad \sigma_{ctx} \sqsubseteq \sigma_{val} \qquad \sigma_{val'} = \sigma \sqcup \sigma_{ctx} \qquad sh' = sh \uplus (l, f) \mapsto (\sigma_{exists}, \sigma_{val'}) \end{array}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh', s\rho, scs, pc\}}$$

UPGPROPEXISTSLAB

$$\frac{\begin{array}{c} o = \text{UpgPropExistsLab}\,(l, f, e_o, e_f, \sigma) \qquad \sigma_o = \mathsf{lev}(s\rho, e_o) \qquad \sigma_f = \mathsf{lev}(s\rho, e_f) \qquad \sigma_{ctx} = \sigma_o \sqcup \sigma_f \sqcup lvl(pc) \\ sh = sh' \uplus (l, f) \mapsto (\sigma_{exists}, \sigma_{val}) \qquad \sigma_{ctx} \sqsubseteq \sigma_{exists} \qquad \sigma_{exists'} = \sigma \sqcup lev_{ctx} \qquad sh' = sh \uplus (l, f) \mapsto (\sigma_{exists'}, \sigma_{val}) \end{array}}{\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm, sh', s\rho, scs, pc\}}$$

Figure 4.3: Semantic of Monitor Commands: $\{sm, sh, s\rho, scs, pc\} \xrightarrow{o} \{sm', sh', s\rho', scs', pc'\}$

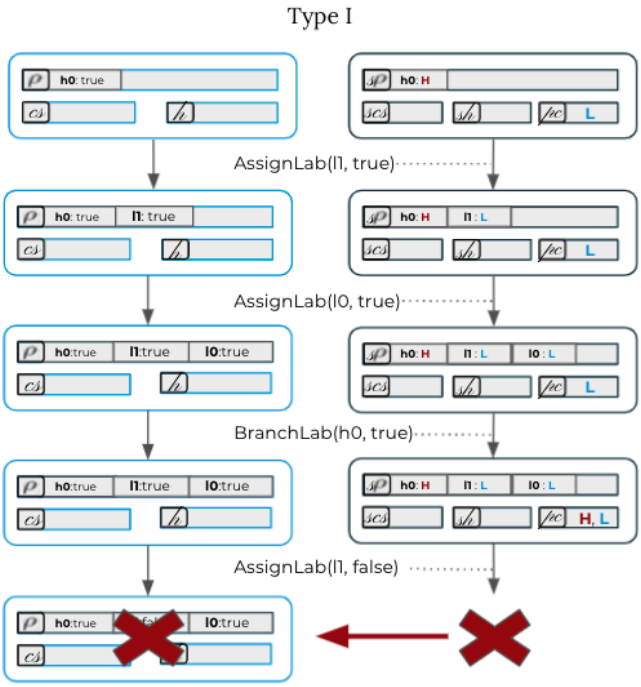


Figure 4.4: Monitored Execution Example (Type I)

Type II Example The example starts with the pc stack $[L]$ indicating that the program is executing in a low context. After the definition of a new object o, the guard of the conditional statement is evaluated; hence, the pc stack is extended to $[H, L]$. Then, within the scope of the conditional statement, the execution of the field assignment `o.f0` $:=$ `false` triggers an information flow exception as it constitutes a no-sensitive-upgrade. Note that the value level of `o.f0` is L and the level of the execution context is H.

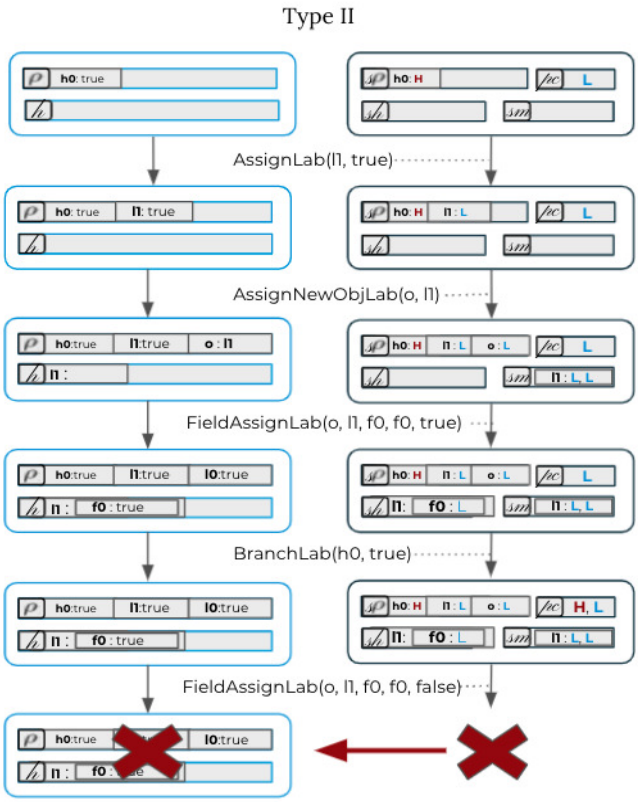


Figure 4.5: Monitored Execution Example (Type II)

4.3.4 Soundness

In order to formally define what it means for a monitor to be non-interferent, we have to introduce a low-equality relation between labeled ECMA-SL configurations. Informally, we say that the semantic configuration Ω_1 labelled by Φ_1 is low-equal to the semantic configuration Ω_2 labelled by Φ_2 at security level σ, written $\{\Omega_1, \Phi_1\} \sim_\sigma \{\Omega_2, \Phi_2\}$, if their low-projections coincide; formally:

$$\{\Omega_1, \Phi_1\} \sim_\sigma \{\Omega_2, \Phi_2\} \Leftrightarrow \Omega_1 \restriction^\sigma_{\Phi_1} = \Omega_2 \restriction^\sigma_{\Phi_2} \tag{4.2}$$

We define low-projection for configurations component-wise, by unpacking the tuple of the given configuration and applying to each of its elements its specific low-projection; formally:

$$\{h, \rho, cs, st\} \restriction^\sigma_{\{-,sh,s\rho,scs,pc\}} \stackrel{def}{=} \{h \restriction^\sigma_{sh}, \rho \restriction^\sigma_{s\rho}, cs \restriction^\sigma_{scs}, st \restriction^\sigma_{pc}\} \tag{4.3}$$

While the definitions of low-projection for heaps and stores were given in Section 4.2, the definitions of low-projection for call stacks and continuations have not been introduced yet. For clarity, we will first state the main non-interference theorem to be proven and only then give the missing definitions, as they are not essential for understanding the intuition behind the result.

Informally, we say that a security monitor is non-interferent if successfully-terminating monitored executions always preserve the low-equality relation. In other words, whenever an attacker cannot distinguish two labelled initial configurations, then the attacker is also unable to distinguish their corresponding final configurations. Hence, an attacker cannot use the monitored execution of a program as a means to obtain information about the confidential contents of a configuration. Theorem 1 states that

Low-Projection for Continuations: $st \restriction^{\sigma}_{pc}$

VISIBLE TOP STATEMENT
$$\frac{\sigma' \sqsubseteq \sigma}{st \restriction^{\sigma}_{\sigma'::pc'} \triangleq st}$$

INVISIBLE TOP STATEMENT
$$\frac{\sigma' \not\sqsubseteq \sigma \qquad st = st_1; merge; st_2 \quad no\ merges\ in\ st_1}{st \restriction^{\sigma}_{\sigma'::pc'} \triangleq st_1 \restriction^{\sigma}_{pc'}}$$

SINGLE INVISIBLE STATEMENT
$$\frac{\sigma' \not\sqsubseteq \sigma}{st \restriction^{\sigma}_{[\sigma']} \triangleq skip}$$

Low-Projection for Call Stacks: $cs \restriction^{\sigma}_{scs}$

EMPTY CALL STACK
$$\varnothing \restriction^{\sigma}_{scs} \triangleq \varnothing$$

VISIBLE CALL STACK REGISTER
$$\frac{cs = (x, \rho, st) :: cs' \qquad scs = (x, s\rho, pc) :: scs' \quad pc \sqsubseteq \sigma \qquad cs'' = cs' \restriction^{\sigma}_{scs'}}{cs \restriction^{\sigma}_{scs} \triangleq (x, \rho \restriction^{\sigma}_{s\rho}, st) :: cs''}$$

INVISIBLE CALL STACK REGISTER
$$\frac{cs = (x, \rho, st) :: cs' \qquad scs = (x, s\rho, pc) :: scs' \quad pc \not\sqsubseteq \sigma \qquad cs'' = cs' \restriction^{\sigma}_{scs'}}{cs \restriction^{\sigma}_{scs} \triangleq cs''}$$

Table 4.3: Low Projection for Continuation and Call Stack Domains

two successfully-terminating monitored executions that start from two low-equal configurations always produce two low-equal configurations.

Theorem 1 *(Non-interferent Monitor). For any semantic configurations* Ω_1 *and* Ω_2 *and monitor configurations* Φ_1 *and* Φ_2 *such that:* $\{\Omega_1, \Phi_1\} \xrightarrow{o_1} {}^*\{\Omega_1', \Phi_1'\}$, $\{\Omega_2, \Phi_2\} \xrightarrow{o_2} {}^*\{\Omega_2', \Phi_2'\}$, *and* $\{\Omega_1, \Phi_1\} \sim_\sigma \{\Omega_2, \Phi_2\}$, *it holds that:* $\{\Omega_1', \Phi_1'\} \sim_\sigma \{\Omega_2', \Phi_2'\}$.

We will now proceed to the formal definitions of low-projection for continuations and call stacks.

Low-Projection for Continuations In order to prove non-interference, one must be able to relate continuations that result from the execution of the same initial program. Note that two executions of a program that branches on a high value may generate two different continuations depending on that high value. However, after the execution of the high branch, one must be able to re-establish the equality between the continuations being executed. To make this argument more concrete, let us consider the example given in the figure below.

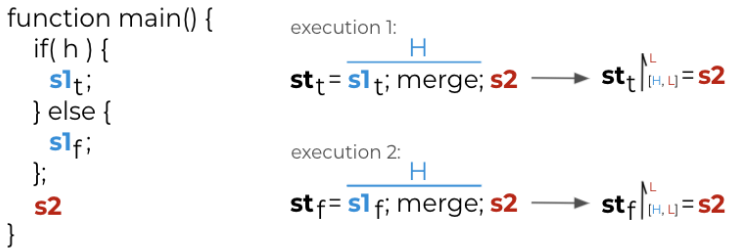


Figure 4.6: Low Projection of the Continuation Statement

The example shows a program that branches on the value of the high-level variable h. The small-step execution of the if statement may generate two different continuations, st_t and st_f, depending on the value of h. Even though these continuations are different from each other, we know that after the merging point, they will again coincide. This intuition is captured by the definition of low-projection for

continuations, which describes the part of a continuation that is visible at a given security level, using the information provided by the pc stack. For instance, $s1_t; merge; s2 \restriction^{L}_{[H,L]} = s2$ because the pc stack tells us that the branch that is currently executing has level H, while the observer is at level L. Hence, the observable part of the continuation only starts after the first merging point, where the context level is again set to L. The formal definition of low-projection, given in Table 4.3, generalizes this mechanism for pc stacks of arbitrary size.

Low-Projection for Call Stacks Analogously to continuations, we must also be able to relate call stacks that result from the execution of the same initial program. Let us consider the example given in Figure 4.7. This example illustrates two executions that vary in the value of the high-level variable h. In particular, in both the executions, the function $faux$ is called with the parameters 3 and h, producing the first call stack register $(x, s\rho_{main}, [L])$. In the body of $faux$, the executions branch in the value of h, calling the function g with distinct parameters. Hence, the second call stack register differs between the two executions. However, the continuations saved in the second call stack register coincide in their low parts. This mechanism is generalized in the definition of low-projection for call stacks given in Table 4.3.

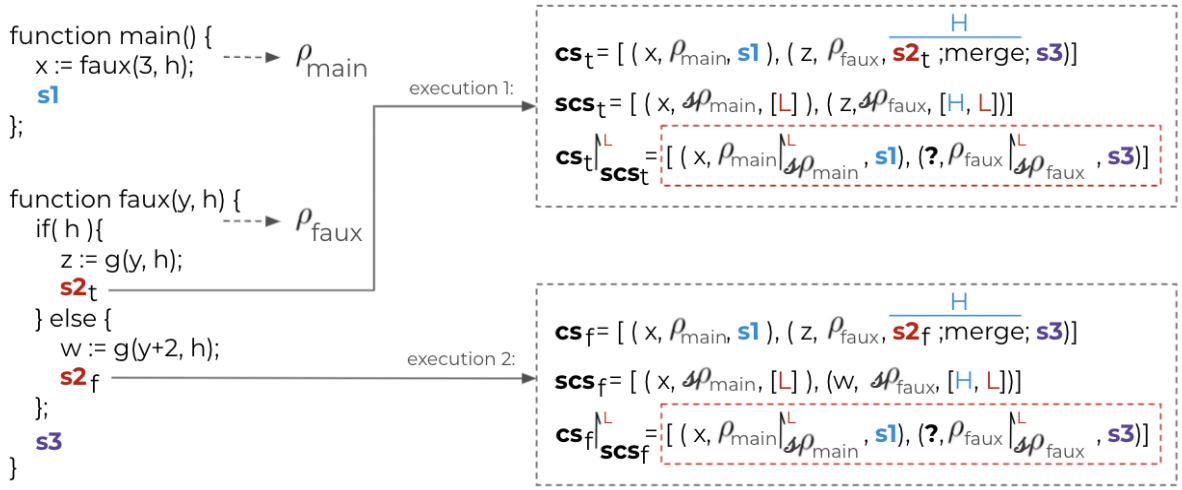


Figure 4.7: Low Projection of the Call Stack

Proofs We believe that the non-interference theorem presented here can be proved by case analysis using the definitions introduced in this section and techniques similar to those presented in (92). However, given the time constraints of this project, we did not have the required time for completing this goal.

4.4 ECMA-SL Monitor Inlining

The monitor inlining approach achieves the same results as the lock-step monitor approach by compiling the given program into an equivalent program that also 'monitors" itself. To this end, we extend the language with support for security levels and their associated operations (e.g. level comparison and least-upper-bound between levels). In the examples, we will use the usual two-point lattice $high - low$.

We have implemented our inlining compiler in OCaml following the information flow monitor described in the previous section. Analogously to well-established approaches (92; 52; 26; 25; 12), our compiler works by pairing up each variable with a *shadow variable* that holds its corresponding security level, and each object field with two *shadow fields* respectively holding its value and existence levels. More concretely, for each variable `x`, the compiler adds a new shadow variable `x_lev` that holds the security level of `x`. Analogously, for each field `f`, the compiler adds two new shadow fields `f_e_lev` and `f_v_lev` respectively holding the the existence and the value levels of `f`. In the following, we refer to `f_e_lev` as the shadow existence field of `f` and to `f_v_lev` as the shadow value field of `f`. Moreover, the compiler adds to every object `o` a shadow structure field, `struct_lev`, holding the structure security level of `o`. Figure 4.8 represents a labeled heap on the left and its instrumented counterpart on the right.

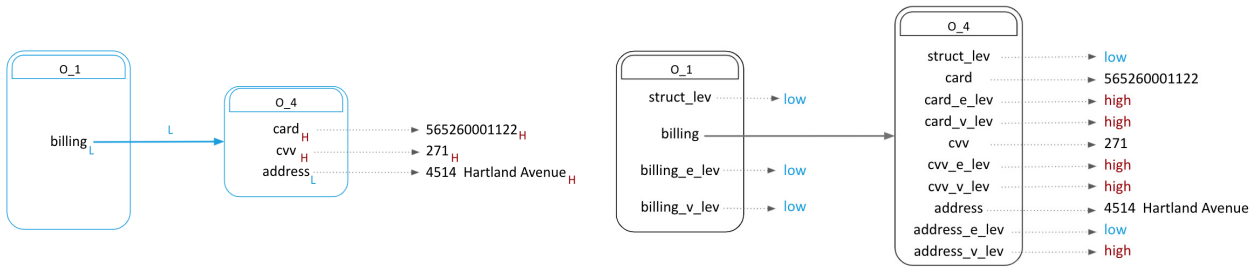


Figure 4.8: Inlining Transformation on Projections

In contrast to variables, whose names are known at compile-time, field names can be dynamically computed. Therefore, we make use of two runtime functions, `fieldExists` and `fieldValue`, to dynamically compute the existence shadow field and the value shadow field of a given field `f`. For instance, `fieldValue(f)` will evaluate to the value shadow field associated with `f`.

Functions are compiled one at the time. Besides the parameters of the original function, the compiled function receives as input a parameter `pc`, holding the level of the context in which the function is called. Additionally, for each parameter `x`, the compiled function receives an extra parameter `x_lev`, holding the security level of `x`. For instance, the function $\texttt{f}(\texttt{x},\texttt{y})\{...\}$ is compiled to $\texttt{f}(\texttt{x},\texttt{x\_lev},\texttt{y},\texttt{y\_lev},\texttt{pc})\{...\}$. After executing a function, we must have access both to its return value and to the security level associated with that value. Hence, every return statement is compiled to an equivalent one which returns a pair composed of the originally returned value and its corresponding security level. For instance, return `x` is compiled to return `(x, lev_x)`. The compilation of function literals must fit together with the compilation of call statements. To this end, call statements are compiled so that they include the additional parameters and process the returned pair to separate the originally returned value from its level. For instance, $\texttt{x} := \texttt{f}(\texttt{z}, 3)$ is compiled to:

$$(\texttt{x},\texttt{x\_lev}) := \texttt{f}(\texttt{z},\texttt{z\_lev}, 3, \texttt{low}, \texttt{pc})$$

During the execution of a function, the level of each control-flow context is kept in a dedicated variable. For simplicity, we create a fresh `pc` variable per control-flow context. The top-most `pc` variable corresponds to the parameter `pc`. The compilation of each control-flow statement introduces a new `pc` variable associated with the corresponding control-flow context. For instance, the if statement

if (h) then {x := 2} is compiled to:

```
pc' := lub(pc, h_lev);
if (h) then {
 leq_1 := leq(pc', x_lev);
 if (leq_1) then {
  lev_e := low
  x_lev := lub(pc', lev_e);
  x := e
 }
else {fail ("Illegal Variable Assign")}
}
```

Compiler Formalization We formalize our information flow compiler as a function mapping pairs consisting of an ECMA-SL statement st and a variable x_{pc} to a new statement st'. Formally, we write $C_{stmt}(st, x_{pc}) = st'$ to mean that the compilation of st results in st', assuming that the current context level is stored in the variable x_{pc}. In defining the compiler for statements, we make use of an auxiliary compiler $C_{expr}(e)$ which returns the statement st'_e that computes the level of e and assigns it to a fresh variable x_e. The statement st'_e uses the function `lub` to compute the least-upper-bound between all the shadow variables corresponding to program variables that occur in e; for instance:

$$C_{expr}(\mathtt{x} + \mathtt{y}) = \mathtt{lev\_1} := \mathtt{lub}(\mathtt{x\_lev}, \mathtt{y\_lev})$$

where `lev_1` corresponds to the generated program variable used to hold the level of the expression.

Our compiler is defined recursively in a syntax-directed fashion, following existing information flow compilers (92; 52; 26; 25; 12). Below we explain the *Variable Assignment*, *Conditional Statement*, *Field Delete*, and *Variable Upgrade* compilation rules. The remaining rules are similar.

Variable Assignment The compiled code first checks whether or not the NSU constraint associated with the assignment holds. To this end, it compares the level of the current context x_{pc} against the level of the variable to be assigned $\hat{x}$ using the function `leq`. If this constraint does not hold, the compiled code throws an information flow exception. If it does hold, the compiled code updates both the value and the level of x. To this end, it uses the function `lub` to compute the least-upper-bound between the level of the context x_{pc} and the level of the expression κ. The obtained level is then assigned to the shadow variable of x, $\hat{x}$.

$$\frac{C_{expr}(e) = \kappa := \mathtt{lub}(\hat{y}_1, ..., \hat{y}_n);}{\begin{array}{ll} C_{stmt}(x := e,\ x_{pc}) = & w_1 := \mathtt{leq}(x_{pc},\ \hat{x}); \\ & \text{if } (w_1) \text{ then } \{ \\ & \quad \kappa := \mathtt{lub}(\hat{y}_1, ..., \hat{y}_n) \\ & \quad \hat{x} := \mathtt{lub}(x_{pc},\ \kappa); \\ & \quad x :=\ e \\ & \} \\ & \text{else } \{\text{fail } (\texttt{"Illegal Variable Assign"})\} \end{array}}$$

Conditional Statement The compiled code first computes the level of the conditional guard e (denoted as κ). Then, it assigns the least-upper-bound between κ and the current x_{pc} level to a fresh variable x'_{pc}. Note that, x'_{pc} will hold the level of the control-flow context created by the *if* statement, either the

then-context or the else-context. The compiled branches of the conditional statement are obtained by applying the compiler recursively, using the new context x'_{pc}.

$$\frac{C_{expr}(e) = \kappa := \texttt{lub}(\hat{y}_1, ..., \hat{y}_n) \qquad st'_1 = C_{stmt}(st_1,\ x'_{pc}) \qquad st'_2 = C_{stmt}(st_2,\ x'_{pc})}{\begin{array}{rl} C_{stmt}(\text{if}\,(e)\ \text{then}\ \{st_1\}\ \text{else}\ \{st_2\},\ x_{pc}) = & \kappa := \texttt{lub}(\hat{y}_1, ..., \hat{y}_n) \\ & x'_{pc} := \texttt{lub}(x_{pc},\ \kappa); \\ & \text{if}\,(e)\ \text{then}\,\{st'_1\}\ \text{else}\,\{st'_2\} \end{array}}$$

Field Delete The compiled code first checks whether or not the NSU constraint associated with the field delete holds. To this end, it computes: (1) the operation-context level, stored in the variable κ_{ctx}, (2) the existence level of the field denoted by x_f, stored in the variable κ'_f, and (3) the relation between the two levels, stored in the variable w_1. If the NSU constraint does not hold, the compiled code throws an information flow exception. If it does hold, the field denoted by x_f is deleted from the object denoted by x_o.

To obtain the operation-context level, the compiled code calls the least-upper-bound function on the variables x_{pc}, $\hat{x_o}$, and $\hat{x_f}$, respectively denoting the context level, the object expression level, and the field expression level. To obtain the existence level of the field denoted by x_f, the compiled code simply reads the corresponding shadow existence field, whose name is computed with the help of the runtime function `fieldExists`.

$$\frac{C_{expr}(x_o) = \kappa_o := \texttt{lub}(\hat{y}_1, ..., \hat{y}_n) \qquad C_{expr}(x_f) = \kappa_f := \texttt{lub}(\hat{z}_1, ..., \hat{z}_n);}{\begin{array}{rl} C_{stmt}(\text{delete}\, x_o[x_f], x_{pc}) = & \kappa_o := \texttt{lub}(\hat{y}_1, ..., \hat{y}_n) \\ & \kappa_f := \texttt{lub}(\hat{z}_1, ..., \hat{z}_n) \\ & \kappa_{ctx} := \texttt{lub}(\kappa_o, \kappa_f, x_{pc}); \\ & x'_f := \texttt{fieldExists}(x_f); \\ & \kappa'_f := x_o[x'_f]; \\ & w_1 := \texttt{leq}(\kappa_{ctx}, \kappa'_f); \\ & \text{if}\,(w_1)\ \text{then}\,\{ \\ & \quad \text{delete}\, x_o[x_f] \\ & \} \\ & \text{else}\,\{ \\ & \quad \text{fail}\,(\texttt{"Illegal Field Delete"}) \\ & \} \end{array}}$$

Variable Upgrade The compiled code first checks whether or not the NSU constraint associated with the variable upgrade holds. To this end, it compares the level of the current context x_{pc} against the level of the variable to be upgraded $\hat{x}$ using the function `leq`. If this constraint does not hold, the compiled code throws an information flow exception. If it does hold, the compiled code updates the level of x. To this end, it uses the function `lub` to compute the least-upper-bound between the level of the context x_{pc} and the parsed level κ_{str}, calculated using the function `parseLev`.

$$\frac{C_{expr}(e) = \kappa := \texttt{lub}(\hat{y}_1, ..., \hat{y}_n);}{C_{stmt}(\text{upgVar}\,(x,\ x_{str}), x_{pc}) = \begin{array}{l} \kappa := \texttt{lub}(\hat{y}_1, ..., \hat{y}_n) \\ \kappa_{str} := \texttt{parseLev}(x_{str}); \\ w_1 := \texttt{leq}(x_{pc}, \hat{x}); \\ \text{if}\ (w_1)\ \text{then}\,\{ \\ \quad \hat{x} := \texttt{lub}(\kappa_{str}, x_{pc}) \\ \} \\ \text{else}\,\{ \\ \quad \text{fail}\,(\texttt{"Illegal UpgVarLab"}) \\ \} \end{array}}$$

4.5 Implementation

We implemented the lock-step monitor and the inlining compiler in OCaml. These implementations were connected to our ECMA-SL interpreter described in Chapter 3 yielding two types of secure execution:

- Monitor + Interpreter: the information flow monitor is plugged into our ECMA-SL interpreter to obtain a monitored interpreter;
- Inlining Compiler + Interpreter: the given program is compiled using the inlining compiler and only then executed in the original interpreter.

Both approaches are equivalent.

Lock-Step Monitor The lock-step monitor was implemented as a functor, comprising 328 lines of code. Its implementation is completely independent of the ECMA-SL interpreter in that it has its own separate representation of security states. The interaction between interpreter and monitor is regulated via the previously described security labels. Hence, one can change the monitor without changing the interpreter and vice-versa, as long as one sticks to the same security labels.

By implementing our information flow monitor as a functor, we make it parametric on the chosen security lattice. This means that one can change the security lattice without modifying the implementation of the monitor. Furthermore, our implementation of the security domains is also parametric on the type of security levels. To this end, we have used polymorphic data types to represent security stores, heaps, and call stacks.

Inlining Compiler The inlining compiler was implemented as an OCaml module and comprises 883 lines of code. It consists of three main functions: `compile_prog`, `compile_func`, and `compile_stmt`, respectively used to compile programs, functions, and statements. These functions precisely follow the compilation rules described is Section 4.3.2.

Besides its main translation functions implemented in OCaml, our inlining compiler comes with a compilation runtime in the form of a set of ECMA-SL functions. This runtime includes an ECMA-SL implementation of the $H - L$ lattice and a set of dedicated functions for upgrading security levels. If one wants to use a different lattice, one simply needs to implement it in ECMA-SL.

Chapter 5

Embedding ECMA-SL in JavaScript

5.1 Deep Embedder

In order to run *ECMA-SL* programs in *JavaScript* engines, we designed a deep embedder of *ECMA-SL* into *JavaScript*, which consists of an *ECMA-SL* interpreter written in *JavaScript*.

Our interpreter has at its core three main abstract classes: Statement, Expression, and Value. These three classes have various subclasses corresponding to each specific syntactic construct. For instance, the Statement class has, among others, the subclasses Block, Assignment, and Conditional Statement. The class diagrams of statements, expressions and values are given in Figures 5.1, 5.2, and 5.3. These classes are arranged according to the Composite pattern (93), which is applicable when dealing with class hierarchies where objects of a given subclass are composed of one or several objects of its superclass(es). The advantage of this pattern is that it allows for the uniform treatment of all subclasses. Following the Composite pattern, the subclasses of Statement may also contain statements, mutatis mutandis for expressions and values.

The most relevant method of statements is the `interpret` method, which receives as an input an ECMA-SL execution context and returns the execution context obtained by interpreting the current statement in that context. An ECMA-SL execution context is implemented as an object that keeps the current heap, variable store, call stack, and statement continuation. The different elements of execution contexts are represented in JavaScript using simple data structures:

- Heap: dictionary mapping locations to ECMA-SL objects;
- ECMA-SL Object: dictionary mapping fields to values;
- Store: dictionary mapping variables to values;
- Call Stack: stack of call stack records;
- Call Stack Record: object containing the various elements which comprise a call stack entry (store, variable, and statement continuation);
- Continuation: stack containing the statements that remain to be executed.

Analogously to statements, the most relevant method of expressions is the `evaluate` method, which receives as input a store and returns the value resulting from the evaluation of the current expression on that store. Unlike statements and expressions, ECMA-SL values do not need to be evaluated. However, we do have to provide JavaScript implementations for all the unary, binary, and n-ary operators included in ECMA-SL. All these implementations are defined in the class Operator, used for the evaluation of

expressions. The class Operator works as a static class in Java, only containing functions that are not attached to any specific object.

5.1.1 Statement Interpretation

The class Statement is extended by the subclasses: Skip, Return, Print, Merge, Loop, Field Lookup, Field Delete, Field Assignment, Fail, Field Collection, New Object, External Call, Call, Assignment, Conditional Statement, and Block. All these subclasses implement the `interpret` method, which interprets the corresponding statement in the given execution context. In the following, we give a brief explanation of the `interpret` method, detailing its implementation for the subclasses Variable Assignment and Conditional Statement.

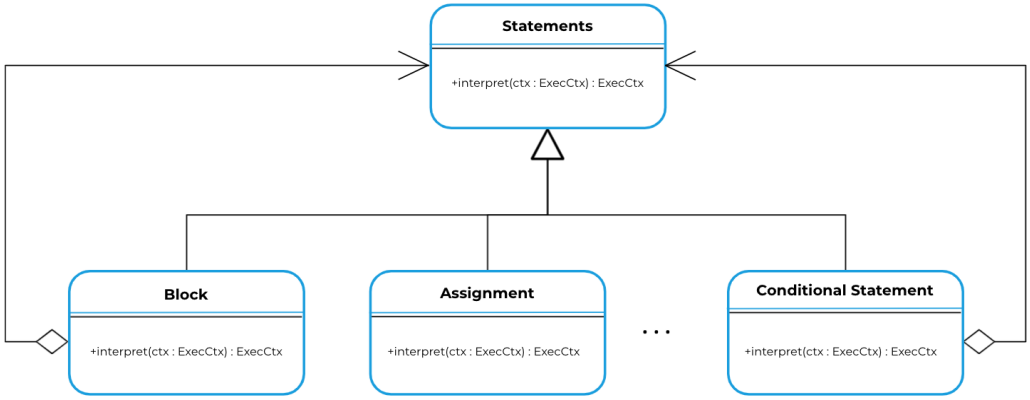


Figure 5.1: Statement Composite

Variable Assignment The `interpret` method of the Variable Assignment class first evaluates the expression to be assigned by calling its method `evaluate` with the current store. Note that every expression knows how to evaluate itself. The obtained value is then set to the left-hand-side variable of the current assignment, `this.x`, with the help of the method `setValue`, defined as part of the Store class. Finally, the method returns an object containing both the updated execution context and the generated monitor label, `AssignLab`.

```
1  //Variable Assignment
2  interpret(ctx){
3    var v = this.expr.evaluate(ctx.store);
4    ctx.store.setValue(this.x, v);
5    return {ctx : ctx, seclabel: new AssignLab(this.stringvar, this.expr)};
6  }
```

Conditional Statement The `interpret` method of the Conditional Statement class first evaluates the test expression by calling its method `evaluate` with the current store. If the resulting value is true, the then branch is appended to the current continuation, followed by the $merge$ statement. If not, the $else$ branch is appended to the current continuation. Note that, some programs might not have an $else$ branch; in those cases, the continuation is left unaltered when the guard evaluates to false. The interpretation of the Conditional Statement always returns the new configuration together with the generated monitor label, `BranchLab`.

```
//Conditional Statement
interpret(ctx){
  var v = this.expr.evaluate(ctx.store);
  if(v){
    ctx.cont = [this.then_block].concat([new Merge()]).concat(ctx.cont);
    return {ctx : ctx, seclabel: new BranchLab(this.expr)};
  } else{
    if(this.else_block){
      ctx.cont = [this.else_block].concat([new Merge()]).concat(ctx.cont);
      return {ctx : ctx, seclabel: new BranchLab(this.expr)};
    }
    else{
      return {ctx : ctx, seclabel: new EmptyLab()};
    }
  }
}
```

5.1.2 Expression Interpretation

The class Expression is extended by the subclasses: Value, Variable, Unary Operation, Binary Operation, and N-Ary Operation. All these subclasses implement the `evaluate` method, which evaluates the corresponding expression in the given store. In the following, we give a brief explanation of the `evaluate` method, detailing its implementation for the subclasses Binary Operation and Variable.

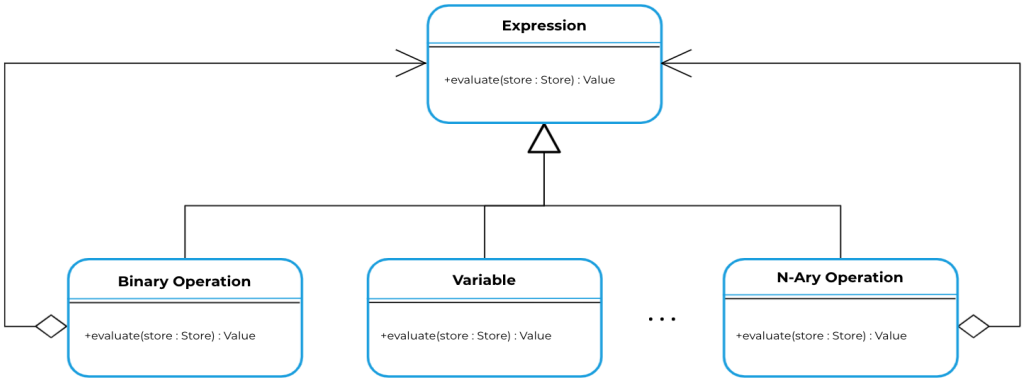


Figure 5.2: Expression Composite

Binary Operation The `evaluate` method of the Binary Operation class recursively evaluates both its left-hand-side and right-hand-side expressions, `this.lhs` and `this.rhs`, and returns the result of applying the corresponding operator, `this.operator`, to the obtained values.

```
//Binary Operation
evaluate(store){
    var v1 = this.lhs.evaluate(store);
    var v2 = this.rhs.evaluate(store);
    return this.operator.interpret(v1,v2);
}
```

Variable The `evaluate` method of the Variable class simply returns the value associated with its corresponding variable, `this.variable`, in the given store. If the variable is not defined in that store, the evaluation throws an exception.

```
1   //Variable
2   evaluate(store){
3     var val = store.sto[this.variable];
4     if (val == undefined)
5       throw new Error("Undefined Variable");
6     else
7       return val;
8   }
```

5.1.3 Values

The class Value is extended by the subclasses: Primitive (comprising Integers, Strings, and Floats), Location, List, Tuple, Type, and Symbol. These classes simply represent a value of a certain type. Hence, they do not need to be further evaluated.

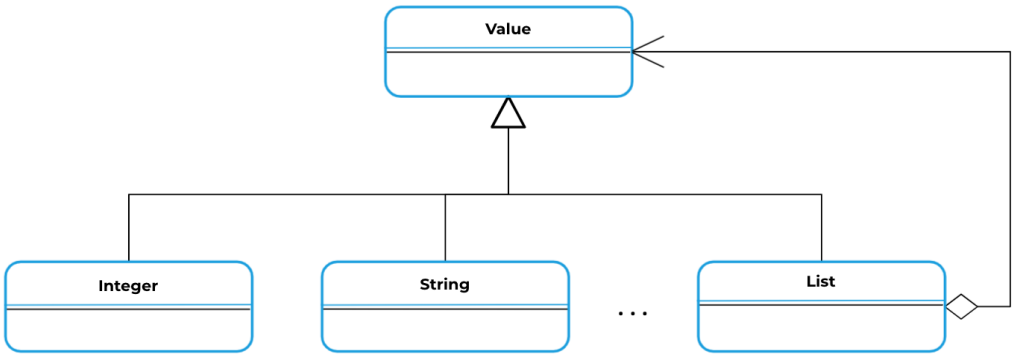


Figure 5.3: Value Composite

5.2 Shallow Embedder

To obtain a more performant implementation of ECMA-SL in JavaScript, we developed a shallow alternative to the deep embedder introduced in the previous section. Our shallow embedder of ECMA-SL into JavaScript consists of a compiler that translates ECMA-SL programs into JavaScript programs, as illustrated in the compilation pipeline depicted below.

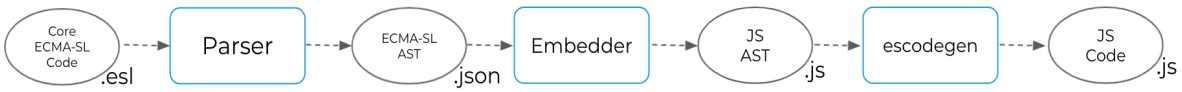


Figure 5.4: Shallow Embedder Pipeline

Our ECMA-SL to JavaScript compiler comprises three main components:

1. The ECMA-SL parser that creates the abstract syntax tree (AST) of the input ECMA-SL program, `Core_ECMA-SL_Code.esl`, and serializes it as a JSON document, `ECMA-SL_AST.json`;
2. The Embedder that transforms the given ECMA-SL AST into a JavaScript AST represented as a JSON document, `JS_AST.json`;
3. The *escodegen* code generator that pretty-prints the final AST as a JavaScript program, `JS_Code.js`.

Data Types Compilation Not every *ECMA-SL* type has a direct equivalent in *JavaScript*. Namely, ECMA-SL has polymorphic tuples and lists, while JavaScript only has arrays. Hence, both types are modeled as specialized JavaScript arrays. At the implementation level, we distinguish an array modeling a tuple from an array modeling a list through a dedicated field, `isTuple`, which is true for the former and false for the latter. All the other ECMA-SL types have a direct JavaScript correspondent, which is summarized in the table below.

ECMA-SL type	JavaScript type
`Integer`	`Number`
`Float`	`Number`
`Boolean`	`Boolean`
`String`	`String`
`List`	`Array`
`Tuple`	`Array`
`Type`	`type`
`Void`	`void`
`Null`	`null`
`Symbol`	`Symbol`
`Object`	`Object`

Table 5.1: Shallow Embedder Type Compilation

Statement Compilation The compilation of ECMA-SL statements to JavaScript statements is straightforward. For each ECMA-SL statement, there is an equivalent JavaScript statement. Note that, since the Skip statement does not exist in JavaScript, it is simply compiled to the expression statement $0;$ which does not have any side effects. The table below summarizes the compilation of ECMA-SL statements to JavaScript statements.

ECMA-SL Statement	JavaScript Statement
`skip`	`0`
`Merge`	`-`
`print e`	`console.log(e)`
`fail e`	`throw e`
`x := e`	`x = e`
`while (e) st`	`while (e) st`
`return e`	`return e`
`x[y] := e`	`x[y] = e`
`delete x[y]`	`delete x[y]`
`x := f(e)`	`x = f(e)`
`x :=`	`x :=`
`x := y[z]`	`x = y[z]`
`st1; st2`	`st1; st2`

Table 5.2: Shallow Embedder Statement Compilation

Expression Compilation The compilation of ECMA-SL expressions is more involved than that of statements, as some ECMA-SL operations do not have a direct JavaScript counterpart. This mismatch is mainly caused by the mismatch between ECMA-SL and JavaScript data types described above. Hence, some ECMA-SL operators are compiled to JavaScript function calls instead of JavaScript operators. For

instance, the ECMA-SL operator for list concatenation is mapped to the `concat` function of the JavaScript Array library. The table below summarizes the compilation of the ECMA-SL operators that do not have a direct JavaScript correspondent.

ECMA-SL Statement	JavaScript Statement
`x = y`	`x === y`
`s_concat [x,y]`	`x + y`
`float_to_string x`	`x + ""`
`l_concat(x,y)`	`x.concat(y)`
`l_add(x,y)`	`x.push(y)`
`x in_list y`	`x includes y`
`tl x`	`x.slice()`
`int_to_float x`	`x`
`to_lower_case x`	`x.toLowerCase()`
`int_to_string x`	`x.toString()`
`abs x`	`Math.abs(x)`
`cos x`	`Math.cos(x)`
`acos x`	`Math.acos(x)`
`tan x`	`Math.tan(x)`
`sin x`	`Math.sin(x)`
`asin x`	`Math.asin(x)`
`atan2(x,y)`	`Math.atan2(x,y)`
`ceil x`	`Math.ceil(x)`
`floor x`	`Math.floor(x)`
`sqrt x`	`Math.sqrt(x)`

ECMA-SL Statement	JavaScript Statement
`e x`	`Math.exp(x)`
`log_e x`	`Math.log(x)`
`log_10 x`	`Math.log10(x)`
`random x`	`Math.random(x)`
`max x`	`Math.max(x)`
`min x`	`Math.min(x)`
`x ** y`	`Math.pow(x,y)`
`s_split(x,y)`	`x.split(y)`
`s_substr(x, y)`	`x.substring(y)`
`l_remove_last x`	`x.pop()`
`l_sort x`	`x.sort()`
`int_to_four_hex x`	`x.fx()`
`to_int32 x`	`toInt32(x)`
`trim x`	`x.trim()`
`s_len x`	`x.length()`
`l_nth(x,y)`	`x[y]`
`s_nth(x,y)`	`x[y]`
`t_len x`	`x.length()`
`t_nth(x,y)`	`x[y]`

Table 5.3: Shallow Embedder Operator Compilation

5.3 Implementation

Both the deep embedder and the shallow embedder are implemented in JavaScript and comprise 995 lines of code. Both embedders receive as input a JSON document representing the AST of an ECMA-SL program. While the deep embedder executes the given program starting from the empty state, the shallow embedder transforms it into an equivalent JavaScript program. We represent JavaScript programs using the AST format defined in the ESTree project (94), which originally comes from the Mozilla Parser API project (95). Furthermore, the shallow embedder pretty-prints the obtained JavaScript program using the `escodegen` JavaScript code generator.

Chapter 6

Evaluation

The goal of the evaluation is to confirm that our OCaml lock-step monitor and inlining compiler detect all types of information flow leaks described in Chapter 4, as well as confirm that both our JavaScript embeddings preserve the semantics of ECMA-SL. To accomplish the first goal, we have built a small test suite comprising 24 ECMA-SL programs that cover all the possible types of information flow leaks, including both explicit and implicit leaks. To accomplish the second goal, we resort to the Test262 JavaScript Conformance Test Suite (33). More concretely, we test our shallow embedder against 1848 tests from the official JavaScript test suite, passing 100% of the selected tests. This gives us high confidence that the shallow embedder works as expected. Due to time constraints, we were not able to exhaustively test the deep embedder.

The results shown in the following were all obtained by running the corresponding tests in a machine with an Intel Core i7-4980HQ CPU 2.80 GHz and DDR3 RAM 16GB.

6.1 Unit Tests

We developed a series of basic tests that cover the different types of legal and illegal information flows in ECMA-SL. These tests were designed to cover categories such as basic operations, basic control-flow, heap operations, and logical operations to evaluate how adequately our information flow monitor and inlining compiler analyze these features.

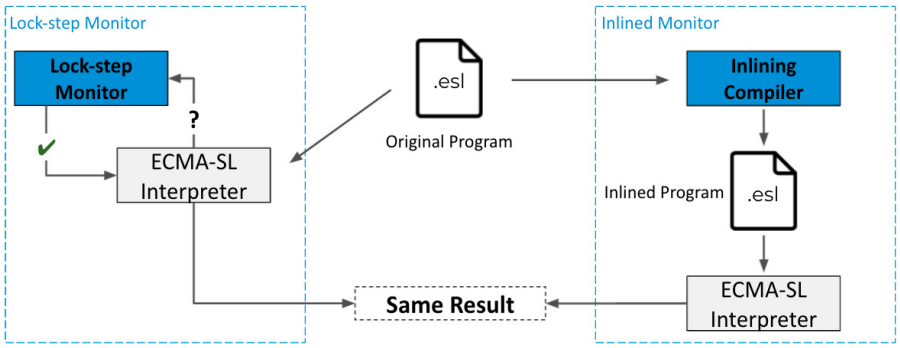


Figure 6.1: ECMA-SL Monitors Unit Test Pipeline

Each ECMA-SL test program was executed with both the lock-step monitor and the inlining compiler. As Figure 6.1 illustrates, each test is expected to have the same result using the two different monitoring

Test	Positive Tests	Successful Positives	Negative Tests	Successful Negative
`Basic Operations`	6	6	6	6
`Control-Flow Operations`	3	3	3	3
`Heap Operations`	8	8	9	9
`Logical Operations`	1	1	1	1
`Total`	18	18	19	19

Table 6.1: ECMA-SL Monitors Unit Tests

approaches. Furthermore, our test suite includes both negative tests, whose execution is supposed to throw an information flow error, and positive tests, whose execution is supposed to complete successfully. The results of running the information flow monitor and the inlining compiler on the unit tests are summarized in Table 6.1. These results allow us to conclude that each test behaves as expected; the tests that were expected to pass do pass and the tests that were expected to fail do fail with the appropriate information flow exception being raised.

6.2 Test 262

To test our shallow embedder, we used the Test262, the official ECMAScript test suite (33). Test262 comprises thousands of non-normative software tests and is routinely used by developers of ECMAScript reference interpreters to check the conformance of their JavaScript implementations with the ECMAScript standard (96; 97). As the ECMAScript language is in constant evolution, Test262 also has to evolve to cover the new features of the language.

Test262 comes with several auxiliary functions to be used in test cases. For instance, the function `assert`(e) is used to check whether or not e evaluates to true; the function `isEqual`(num_1, num_2) tests if two numbers denote the same value; and the function `compareArray`(a, b) checks whether two arrays have the same length and, if so, if they have equal values at equal indexes. These functions are all organized in a single file referred to as the harness of Test262. Hence, to run any Test262 test, one needs to include the code of the harness. In our project, we simply prepend the code of the harness to the code of the test to be executed.

Testing Pipeline In order to test our embedder using Test262, we first compile Test262 tests to Core ECMA-SL and then recompile the obtained Core ECMA-SL programs back to JavaScript. The obtained JavaScript programs are then executed using the *Node* engine and their outcomes are checked against the expected outcomes. More concretely, we follow the testing pipeline illustrated in Figure 6.2, which consists of the following main steps:

- The harness file is prepended to the target Test262 JavaScript file; additionally, if the test is to be run in strict mode, we prepend the directive ”use strict” to the code of the test;
- The resulting JavaScript program is compiled to Core ECMA-SL, generating the file `out.cesl`;
- The file `out.cesl` is compiled back to JavaScript, generating the file `out.js`;
- The file `out.js` is executed in *Node*;
- Finally, the outcome of the test is checked against the expected outcome; while positive tests are expected to complete successfully, negative tests are expected to throw an exception of the type given in the metadata of the test.

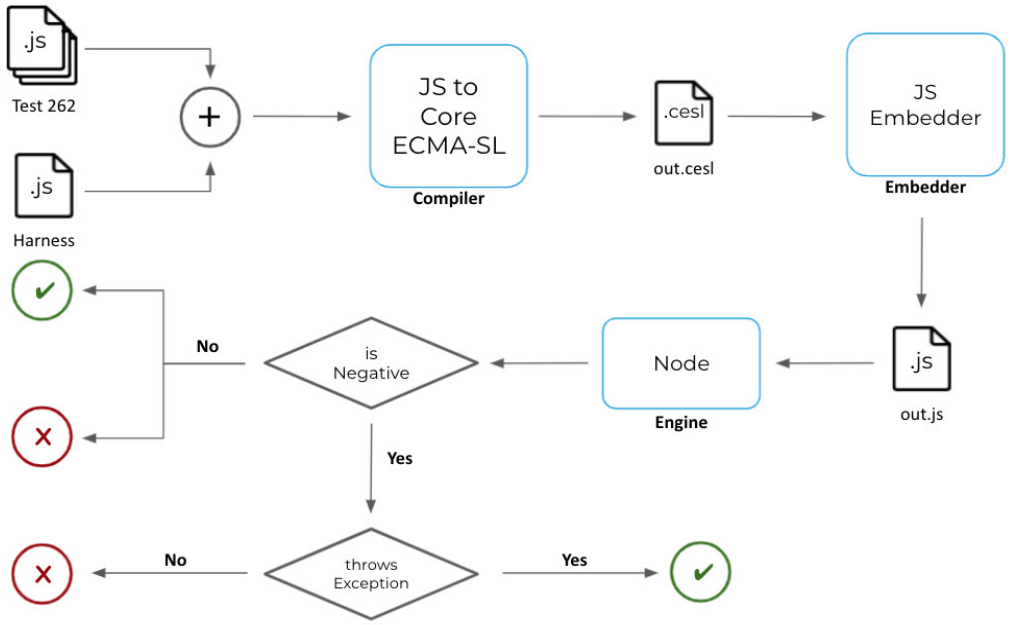


Figure 6.2: Embedding Test Pipeline

Test	Positive Tests	Successful Positives	Negative Tests	Successful Negative
`Arithmetic Expressions`	343	343	0	0
`String Expressions`	5	5	0	0
`Logical Expressions`	397	397	0	0
`Primary Expressions`	35	35	1	1
`Assignment Expressions`	376	376	13	13
`Expressions with Side Effects`	117	117	15	15
`Objects and Properties`	127	127	2	2
`Total`	1400	1400	31	31

Table 6.2: Expression Test Results (Short)

Selection Criteria Due to time constraints, we limited the testing of our shallow embedder to ECMAScript 5 expressions and statements. The documentation about them can be found in Sections 11 and 12 of the ECMAScript 5 Standard.

Testing Results We divided our testing results into two tables, one for Expressions 6.2 and another for Statements 6.3. For clarity, the tests targeting expressions were further divided into 7 sub-groups and the tests targeting statements into 18 sub-groups, one per type of statement. In Appendix A, we give a more fine-grained table for expressions, showcasing the results for each type of expression. In total, we have tested our shallow embedder against 1848 tests, of which 160 were negative tests, passing 100% of all the selected tests.

Test	Positive Tests	Successful Positives	Negative Tests	Successful Negative
Block	10	10	2	2
Break	8	8	10	10
Continue	5	5	10	10
Do While	15	15	8	8
Empty	1	1	0	0
Expressi	2	2	1	1
For	62	62	22	22
For In	20	20	5	5
Function	133	133	8	8
If	17	17	12	12
Labeled	1	1	1	1
Return	5	5	10	10
Switch	5	5	6	6
Throw	14	14	0	0
Try	58	58	16	16
Variable	39	39	10	10
While	15	15	7	7
With	7	7	1	1
Total	417	417	129	129

Table 6.3: Test262 Statements Results

Chapter 7

Conclusion

Information flow control is particularly important on the Web, as web applications often include scripts coming from untrusted origins that execute in the context of the main web page with free access to all of its resources. This security-critical situation has led the research community to propose numerous program analyses (29; 98; 9; 26; 25; 66; 78) to control information flow in *JavaScript* applications. This thesis contributes to this research effort by proposing a new inlining compiler for precisely tracking information flows in JavaScript (ECMAScript 5).

We developed our inlining compiler as part of a wider project, whose goal is to build a tool-suit for *JavaScript* analysis based on a new intermediate language called *ECMA-SL*. We contributed to the overarching *ECMA-SL* project in three different ways: first, we defined the formal semantics of *ECMA-SL* and described its interpreter; second, we designed a new information flow monitor and inlining compiler for *ECMA-SL*; finally, we developed two distinct embedders for running *ECMA-SL* in *JavaScript*. By combining these components together, we obtained a precise information flow inlining compiler for *JavaScript*. We tested the obtained inlining compiler against a subset of Test262, the official JavaScript test suite, showing that it preserves the semantics of the original programs. Furthermore, we have created a set of unit tests to test our monitoring mechanism, confirming that it correctly flags all types of insecure information flows at the *ECMA-SL* level.

The developed work will be open-sourced and made available online together with the remaining components of the ECMA-SL project.

Future Work We distinguish between two types of future work: immediate and long-term. Due to the extension of this project and its inherent time constraints, we have not concluded the ideal evaluation of the project. Therefore, we define the immediate future work to be the completion of the project's evaluation, which would include:

- Testing our information flow monitoring pipeline against the complete Test262 test suite.
- Testing the combination of our inlining compiler with our shallow embedder by generating a random information flow test suite.
- Proving that the proposed ECMA-SL information flow monitor is non-interferent.

In the long term, we would like to assess the real-world applications of our tool-suit, which can be used to test other, less precise, information flow control tools for JavaScript, as well as to directly find information flow bugs in JavaScript programs. More concretely, we would like to:

- Use our inlining compiler for JavaScript to test other monitors/inlining compilers for securing information flow in JavaScript. We are particularly interested in testing the JSFlow (27) information flow

monitor and the JEST inlining compiler (26). To do this, we would generate a random information flow test suite by annotating Test262 tests with random security levels, and we would compare the results of our tool against the results of other tools for the obtained test-suite.

- Use our monitor together with a symbolic execution engine for JavaScript, such as JaVerT 2.0 (29; 98; 99) to find illegal information flows in JavaScript programs. By instrumenting the program with the information flow analysis, one can find inputs that trigger illegal information flows.
- Create other ECMA-SL information flow monitors with different levels of granularity and overhead. We are particularly interested in implementing a taint monitor for ECMA-SL, as it was pointed out in current research that implicit flows do not lead to serious security vulnerabilities for the majority of web applications (100).
- Set up a streamlined procedure for automatically obtaining the implementation of a new security lattice from its declarative specification and integrating the resulting implementation in our tool-chain. This would greatly ease the process of defining and implementing new security lattices, which are often application-specific.

Bibliography

[1] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[3] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium*, USENIX Association, August 2018.

[4] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, (New York, NY, USA), p. 475–488, Association for Computing Machinery, 2014.

[5] OpenSSL, "Openssl 1.0.1 implementation." `https:///openssl.org/`, 2011.

[6] H. Chen, A. Tiu, Z. Xu, and Y. Liu, "A permission-dependent type system for secure information flow analysis," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pp. 218–232, 2018.

[7] B. C. Pierce, *Types and Programming Languages*. The MIT Press, 1st ed., 2002.

[8] V. Rajani, I. Bastys, W. Rafnsson, and D. Garg, "Type systems for information flow control: The question of granularity," *ACM SIGLOG News*, vol. 4, p. 6–21, Feb. 2017.

[9] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, (New York, NY, USA), p. 113–124, Association for Computing Machinery, 2009.

[10] T. H. Austin and C. Flanagan, "Permissive dynamic information flow analysis," in *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, (New York, NY, USA), Association for Computing Machinery, 2010.

[11] T. H. Austin, T. Schmitz, and C. Flanagan, "Multiple facets for dynamic information flow with exceptions," *ACM Trans. Program. Lang. Syst.*, vol. 39, May 2017.

[12] A. Chudnov and D. A. Naumann, "Information flow monitor inlining," in *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, (USA), p. 200–214, IEEE Computer Society, 2010.

[13] Y. Liu and A. Milanova, “Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows,” in *2010 14th European Conference on Software Maintenance and Reengineering*, pp. 146–155, 2010.

[14] A. Russo and A. Sabelfeld, “Dynamic vs. static flow-sensitive security analysis,” in *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF ’10, (USA), p. 186–199, IEEE Computer Society, 2010.

[15] H. Mantel, *Information Flow and Noninterference*, pp. 605–607. Boston, MA: Springer US, 2011.

[16] L. Cardelli, “Type systems,” *ACM Comput. Surv.*, vol. 28, p. 263–264, Mar. 1996.

[17] H. Jiang, D. Lin, X. Zhang, and X. Xie, “Type system in programming languages,” *Journal of Computer Science and Technology*, vol. 16, pp. 286–292, May 2001.

[18] L. Beringer and M. Hofmann, “Secure information flow and program logics,” in *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF ’07, (USA), p. 233–248, IEEE Computer Society, 2007.

[19] P. A. Gardner, S. Maffeis, and G. D. Smith, “Towards a program logic for javascript,” *SIGPLAN Not.*, vol. 47, p. 31–44, Jan. 2012.

[20] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Commun. ACM*, vol. 12, p. 576–580, Oct. 1969.

[21] J. C. Reynolds, “Separation logic: a logic for shared mutable data structures,” in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74, 2002.

[22] R. Giacobazzi and I. Mastroeni, “Abstract non-interference: Parameterizing non-interference by abstract interpretation,” in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’04, (New York, NY, USA), p. 186–197, Association for Computing Machinery, 2004.

[23] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[24] S. Zdancewic, “Challenges for information-flow security,” in *In Proc. Programming Language Interference and Dependence (PLID*, 2004.

[25] J. F. Santos and T. Rezk, “An information flow monitor-inlining compiler for securing a core of javascript,” in *ICT Systems Security and Privacy Protection* (N. Cuppens-Boulahia, F. Cuppens, S. Jajodia, A. Abou El Kalam, and T. Sans, eds.), (Berlin, Heidelberg), pp. 278–292, Springer Berlin Heidelberg, 2014.

[26] A. Chudnov and D. A. Naumann, “Inlined information flow monitoring for javascript,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS ’15, (New York, NY, USA), p. 629–643, Association for Computing Machinery, 2015.

[27] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, “Jsflow: Tracking information flow in javascript and its apis,” in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC ’14, (New York, NY, USA), p. 1663–1671, Association for Computing Machinery, 2014.

[28] WHATWG, “Dom living standard.” `https://dom.spec.whatwg.org`, 2021.

[29] J. Fragoso Santos, P. Maksimović, D. Naudžiūnienundefined, T. Wood, and P. Gardner, "Javert: Javascript verification toolchain," *Proc. ACM Program. Lang.*, vol. 2, Dec. 2017.

[30] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, "Adsafety: Type-based verification of javascript sandboxing," in *20th USENIX Security Symposium (USENIX Security 11)*, (San Francisco, CA), USENIX Association, Aug. 2011.

[31] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of javascript," in *ECOOP 2010 – Object-Oriented Programming* (T. D'Hondt, ed.), (Berlin, Heidelberg), pp. 126–150, Springer Berlin Heidelberg, 2010.

[32] S. A. Zdancewic and A. Myers, *Programming Languages for Information Security*. PhD thesis, USA, 2002. AAI3063751.

[33] "Test262 - official ecmascript conformance test suite." Accessed on 2020-06-07.

[34] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J.Sel. A. Commun.*, vol. 21, p. 5–19, Sept. 2006.

[35] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in *Software Safety and Security*, 2012.

[36] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, p. 236–243, May 1976.

[37] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, p. 504–513, July 1977.

[38] A. Chudnov, G. Kuan, and D. A. Naumann, "Information flow monitoring as abstract interpretation for relational logic," in *2014 IEEE 27th Computer Security Foundations Symposium*, pp. 48–62, 2014.

[39] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," pp. 238–252, 01 1977.

[40] G. Kuan, A. Chudnov, and D. Naumann, "Information flow monitoring as abstract interpretation for relational logic," vol. 2014, 07 2014.

[41] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, (New York, NY, USA), p. 238–252, Association for Computing Machinery, 1977.

[42] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *In 6th POPL*, pp. 269–282, ACM Press, 1979.

[43] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, CSFW '04, (USA), p. 100, IEEE Computer Society, 2004.

[44] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, May 2018.

[45] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, p. 82–90, Feb. 2013.

[46] S. S. Ishtiaq and P. W. O'Hearn, "Bi as an assertion language for mutable data structures," *SIGPLAN Not.*, vol. 36, p. 14–26, Jan. 2001.

[47] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, (Berlin, Heidelberg), p. 1–19, Springer-Verlag, 2001.

[48] A. Aguirre, G. Barthe, M. Gaboardi, D. Garg, and P.-Y. Strub, "A relational logic for higher-order programs," *Proc. ACM Program. Lang.*, vol. 1, Aug. 2017.

[49] T. Amtoft and A. Banerjee, "Information flow analysis in logical form," in *Static Analysis* (R. Giacobazzi, ed.), (Berlin, Heidelberg), pp. 100–115, Springer Berlin Heidelberg, 2004.

[50] A. Birgisson, D. Hedin, and A. Sabelfeld, "Boosting the permissiveness of dynamic information-flow tracking by testing," 09 2012.

[51] G. Le Guernic and T. Jensen, "Monitoring information flow," 05 2006.

[52] J. Magazinius, A. Russo, and A. Sabelfeld, "On-the-fly inlining of dynamic security monitors," in *Security and Privacy – Silver Linings in the Cloud* (K. Rannenberg, V. Varadharajan, and C. Weber, eds.), (Berlin, Heidelberg), pp. 173–186, Springer Berlin Heidelberg, 2010.

[53] F. Besson, N. Bielova, and T. Jensen, "Hybrid information flow monitoring against web tracking," pp. 240–254, 06 2013.

[54] S. Moore and S. Chong, "Static analysis for efficient hybrid information-flow control," in *2011 IEEE 24th Computer Security Foundations Symposium*, pp. 146–160, 2011.

[55] J. Fragoso Santos, T. Jensen, T. Rezk, and A. Schmitt, "Hybrid typing of secure information flow in a javascript-like language," pp. 63–78, 01 2016.

[56] K. Knowles and C. Flanagan, "Hybrid type checking," *ACM Trans. Program. Lang. Syst.*, vol. 32, Feb. 2010.

[57] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, "Generalizing permissive-upgrade in dynamic information flow analysis," in *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*, PLAS'14, (New York, NY, USA), p. 15–24, Association for Computing Machinery, 2014.

[58] M. Ngo, N. Bielova, C. Flanagan, T. Rezk, A. Russo, and T. Schmitz, "A Better Facet of Dynamic Information Flow Control," in *WWW '18 Companion: The 2018 Web Conference Companion*, (Lyon, France), pp. 1–9, Apr. 2018.

[59] P. Ohmann and B. Liblit, "Lightweight control-flow instrumentation and postmortem analysis in support of debugging," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 378–388, 2013.

[60] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *2010 IEEE Symposium on Security and Privacy*, pp. 109–124, 2010.

[61] W. Rafnsson and A. Sabelfeld, "Secure multi-execution: Fine-grained, declassification-aware, and transparent," in *2013 IEEE 26th Computer Security Foundations Symposium*, pp. 33–48, 2013.

[62] C. Project, "The blink web engine." `https://chromium.org/blink/`, 2013.

[63] M. Foundation, “The spidermonkey engine.” https://spidermonkey.dev/, 1996.

[64] KDE, “The webkit engine.” https:///webkit.org/, 1998.

[65] Google, “The v8 javascript engine.” https://v8project.blogspot.ie/, 2017.

[66] A. L. Scull Pupo, L. Christophe, J. Nicolay, C. de Roover, and E. Gonzalez Boix, “Practical information flow control for web applications,” in *Runtime Verification* (C. Colombo and M. Leucker, eds.), (Cham), pp. 372–388, Springer International Publishing, 2018.

[67] P. Ratanaworabhan, B. Livshits, and B. Zorn, “Jsmeter: Comparing the behavior of javascript benchmarks with real web applications,” 01 2010.

[68] D. Hedin and A. Sabelfeld, “Information-flow security for a core of javascript,” in *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pp. 3–18, 2012.

[69] C.-A. Staicu, M. Pradel, and B. Livshits, “Synode: Understanding and automatically preventing injection attacks on node.js,” 01 2018.

[70] F. Gauthier, B. Hassanshahi, and A. Jordan, “A¡span class=”smallcaps smallercapital”¿ffogato¡/span¿: Runtime detection of injection attacks for node.js,” in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA ’18, (New York, NY, USA), p. 94–99, Association for Computing Machinery, 2018.

[71] R. Karim, F. Tip, A. Sochůrková, and K. Sen, “Platform-independent dynamic taint analysis for javascript,” *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1364–1379, 2020.

[72] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, “Nodest: Feedback-driven static analysis of node.js applications,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 455–465, Association for Computing Machinery, 2019.

[73] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, “Riding out DOMsday: Toward detecting and preventing DOM cross-site scripting,” in *Proceedings of the 25th Network and Distributed System Security Symposium*, 2018.

[74] S. Lekies, B. Stock, and M. Johns, “25 million flows later - large-scale detection of dom-based xss,” pp. 1193–1204, 11 2013.

[75] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, “Dexterjs: Robust testing platform for dom-based xss vulnerabilities,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), p. 946–949, Association for Computing Machinery, 2015.

[76] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, “Taj: Effective taint analysis of web applications,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’09, (New York, NY, USA), p. 87–97, Association for Computing Machinery, 2009.

[77] A. Bichhawat, V. Rajani, J. Jain, D. Garg, and C. Hammer, “Webpol: Fine-grained information flow policies for web browsers,” pp. 242–259, 06 2017.

[78] B. Sayed, I. Traoré, and A. Abdelhalim, “If-transpiler: Inlining of hybrid flow-sensitive security monitor for javascript,” *Computers & Security*, vol. 75, 02 2018.

[79] J. Fragoso Santos, T. Rezk, and A. Almeida Matos, “Modular Monitor Extensions for Information Flow Security in JavaScript,” in *Trustworthy Global Computing*, (Madrid, Spain), 2015.

[80] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, “Information flow control in webkit’s javascript bytecode,” in *Principles of Security and Trust* (M. Abadi and S. Kremer, eds.), (Berlin, Heidelberg), pp. 159–178, Springer Berlin Heidelberg, 2014.

[81] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer, “Information flow control for event handling and the dom in web browsers,” in *2015 IEEE 28th Computer Security Foundations Symposium*, pp. 366–379, July 2015.

[82] W. Groef, D. Devriese, N. Nikiforakis, and F. Piessens, “Flowfox: A web browser with flexible and precise information flow control,” pp. 748–759, 10 2012.

[83] D. Hedin, A. Sjösten, F. Piessens, and A. Sabelfeld, “A principled approach to tracking information flow in the presence of libraries,” in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, (Berlin, Heidelberg), p. 49–70, Springer-Verlag, 2017.

[84] T. H. Austin and C. Flanagan, “Multiple facets for dynamic information flow,” in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’12, (New York, NY, USA), p. 165–178, Association for Computing Machinery, 2012.

[85] B. Eich, “Narcissus.” `https://github.com/mozilla/narcissus`, 2012.

[86] L. Loureiro, “Ecma-sl - a platform for specifying and running the ecmascript standard,” Master’s thesis, Instituto Superior Técnico, July 2021.

[87] A. Sabelfeld and A. Russo, “From dynamic to static and back: Riding the roller coaster of information-flow control research,” in *Perspectives of Systems Informatics* (A. Pnueli, I. Virbitskaite, and A. Voronkov, eds.), (Berlin, Heidelberg), pp. 352–365, Springer Berlin Heidelberg, 2010.

[88] “Ocaml - general-purpose, multi-paradigm programming language.” `"https://ocaml.org/"`. Accessed on 2020-06-07.

[89] F. Pottier and Y. Régis-Gianas, “Menhir reference manual.” `http://gallium.inria.fr/~fpottier/menhir/manual.html`, 2021.

[90] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[91] X. Leroy, “A modular module system,” *J. Funct. Program.*, vol. 10, p. 269–303, May 2000.

[92] J. Fragoso Femenin dos Santos, *Enforcing secure information flow in client-side Web applications*. Theses, Université Nice Sophia Antipolis, Dec. 2014.

[93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[94] S. McKenzie, K. Simpson, M. Sherov, A. Hidayat, A. Heine, D. Herman, and M. Ficarra, “Estree.” `https://github.com/estree/estree`, 2021.

[95] A. Chudnov, “Mozilla js parser api.” `https://github.com/jswebtools/mozilla-js-parser-api`, 2015.

[96] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, "A trusted mechanised javascript specification," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, (New York, NY, USA), p. 87–100, Association for Computing Machinery, 2014.

[97] D. Park, A. Ştefănescu, and G. Roşu, "Kjs: A complete formal semantics of javascript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, (New York, NY, USA), p. 346–356, Association for Computing Machinery, 2015.

[98] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner, "Javert 2.0: Compositional symbolic execution for javascript," *Proc. ACM Program. Lang.*, vol. 3, Jan. 2019.

[99] J. F. Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner, "Symbolic execution for javascript," in *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[100] C. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, "An empirical study of information flows in real-world javascript," *CoRR*, vol. abs/1906.11507, 2019.

Appendix A

Appendix A

Test	Positive Tests	Successful Positives	Negative Tests	Successful Negative
Addition	39	39	0	0
Array	11	11	0	0
Assignment	35	35	1	1
Bitwise And	22	22	0	0
Bitwise Not	14	14	0	0
Bitwise Or	22	22	0	0
Bitwise Xor	22	22	0	0
Call	33	33	1	1
Comma	5	5	0	0
Compound Assignment	341	341	12	12
Concatenation	5	5	0	0
Conditional	15	15	0	0
Delete	62	62	0	0
Division	34	34	0	0
Does Not Equals	29	29	0	0
Equals	30	30	0	0
Function	2	2	1	1
Greater Then	76	76	0	0
Grouping	8	8	0	0
In	13	13	0	0
InstanceOf	38	38	0	0
Left Shift	37	37	0	0
Less Than	36	36	0	0
Less Than Or Equal	41	41	0	0
Logical And	15	15	0	0
Logical Not	17	17	0	0
Logical Or	15	15	0	0
Modulus	31	31	0	0
Multiplication	32	32	0	0
New	13	13	0	0
Object	23	23	1	1
Postfix Decrement	21	21	5	5
Postfix Increment	21	21	6	6
Prefix Decrement	21	21	2	2
Prefix Increment	21	21	1	1
Property Accessors	19	19	0	0
Relational	1	1	0	0
Right-shift	29	29	0	0
Strict Does Not Equals	21	21	0	0
Strict Equals	21	21	0	0
Subtraction	31	31	0	0
This	2	2	1	1
TypeOf	8	8	0	0
Unary Minus	12	12	0	0
Unary Plus	10	10	0	0
Unsigned Right Shift	9	9	0	0
Void	37	37	0	0
Total	1400	1400	31	31

Table A.1: Test262 Expression Results