Precise Information Flow Control for JavaScript

(extended abstract of the MSc dissertation)

Francisco João do Vale Lopes e Silva Quinaz
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisors: Professors José Fragoso Santos and Ana Almeida Matos

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 the user's resources.

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*.

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* [27], the *Meltdown* [28], and the *Foreshadow* [45], which took advantage of vulnerabilities found in microprocessors; and the Heartbleed Bug [12], a memory related vulnerability found in the OpenSSL's (version 1.0.1) [32] 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. [8, 34, 37]) and instrumentation (e.g. [2–4, 9, 29, 39]). 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 [31], 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. [7, 25]; information flow monitors, (e.g. [2–4, 9, 29, 39]); program logics, (e.g. [5, 18, 24, 38]); abstract interpreters, (e.g. [19]), amongst others, (e.g. [40]) However, none of the proposed mechanisms has been widely adopted in practice. As pointed out by Steve Zdancewic in [46]:

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

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, the information flow analyses for *JavaScript* proposed so far are based on dynamic approaches. Furthermore, they offer no guarantees of capturing all of the *JavaScript*'s implicit information flows.

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 [15] for verifying and testing

JavaScript programs, and the ADsafety tool [35] for verifying isolation properties of *JavaScript* applications.

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*.

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.

2 Related Work

2.1 Secure Information Flow

Programs manipulate information to complete the task at hand. This manipulation creates dependencies between resources. Understanding these dependencies is fundamental to check whether or not an execution is secure. Information flow security [23, 41] 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. 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 [11]), 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.

2.2 Non-Interference

Secure information flow can be defined as a mathematical property called *non-interference* [31]. 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.

2.3 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.

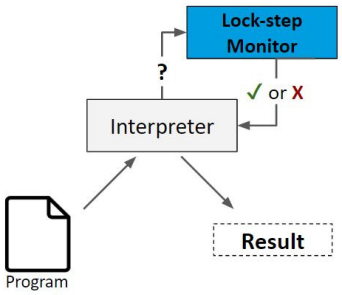

Figure 1. Lock-step monitor architecture

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.

2.4 Monitor Inlining

As shown in Figure 2, 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 [9], 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 [36], SpiderMonkey [13], WebKit [26], and V8 [20]) 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.

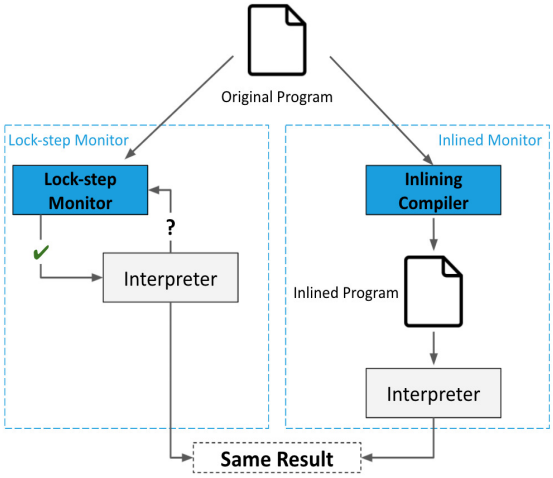

Figure 2. Comparison between Lock-step and Inlined Monitor Architectures

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.

3 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.

3.1 Core ECMA-SL

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).

Syntax of the ECMA-SL Language

$$
\begin{aligned}
v \in \mathcal{V} &:= i \mid f \mid s \mid b \mid l \mid [v_1, \ldots, v_n] \mid \tau \mid (v_1, \ldots, v_n) \mid \mathsf{void} \mid \\
&\quad \mathsf{null} \mid sy \\
e \in \mathcal{E} &:= v \mid x \mid \ominus e \mid e_1 \oplus e_2 \mid \otimes(e_1, \ldots, e_n) \\
st \in \mathcal{S}t &:= st_1; st_2 \mid \mathsf{skip} \mid \mathsf{merge} \mid \mathsf{fail}\,(e) \mid x := e \mid \\
&\quad \mathsf{if}\,(e)\ \mathsf{then}\ \{st_1\}\ \mathsf{else}\ \{st_2\} \mid \mathsf{while}\ (e)\ \mathsf{do}\ \{st\} \mid \\
&\quad \mathsf{return}\,(e) \mid x := e(e_1, \ldots, e_n) \mid x := \{\} \mid \\
&\quad x := e_1\ \mathsf{in}\ e_2 \mid e_1[e_2] := e_3 \mid \mathsf{delete}\, e_1[e_2] \mid \\
&\quad x := \mathsf{fields}\, e \mid x := e_1[e_2] \mid x := e@(e_1, \ldots, e_n)
\end{aligned}
$$

Evaluation of Basic Statements Using the semantics of expressions, we define a small-step semantics for *ECMA-SL* statements in Figure 3. 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 $\mathsf{AssignLab}\,(x, e)$. 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* [15, 21]. Here, we only explain the semantic rules for variable assignments, conditional statements, function calls, field assignments, and field deletions. The remaining rules are analogous.

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.

$$\textsc{Cond. Statement - True}\quad \frac{[\![e]\!]_\rho = \mathsf{true} \qquad o = \mathsf{BranchLab}(e, \mathsf{true})}{\{h, \rho, cs, \mathsf{if}(e)\ \mathsf{then}\ \{st_1\}\ \mathsf{else}\ \{st_2\}\} \xrightarrow{o} \{h, \rho, cs, st_1; \mathsf{merge}\}}$$

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

$$\textsc{Sequential Composition - 1}\quad \frac{st_1 \notin Call \qquad \{h, \rho, cs, st_1\} \xrightarrow{o} \{h', \rho', cs, st_1'\}}{\{h, \rho, cs, st_1; st_2\} \xrightarrow{o} \{h', \rho', cs, st_1'; st_2\}}$$

$$\textsc{Sequential Composition - 2}\quad \{h, \rho, cs, \mathsf{skip}; st\} \xrightarrow{\bullet} \{h, \rho, cs, st\}$$

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

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

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

Figure 3. Semantics of Statements

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.

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 Example

Heap Commands The point of the exampleis 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.

4 ECMA-SL Security Monitor

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:	$\Omega \xrightarrow{o} \Omega'$
Monitored Semantic Transition:	$\Phi \xrightarrow{o} \Phi'$
General Monitor 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 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 definition of the heap low-projection is given in Table 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

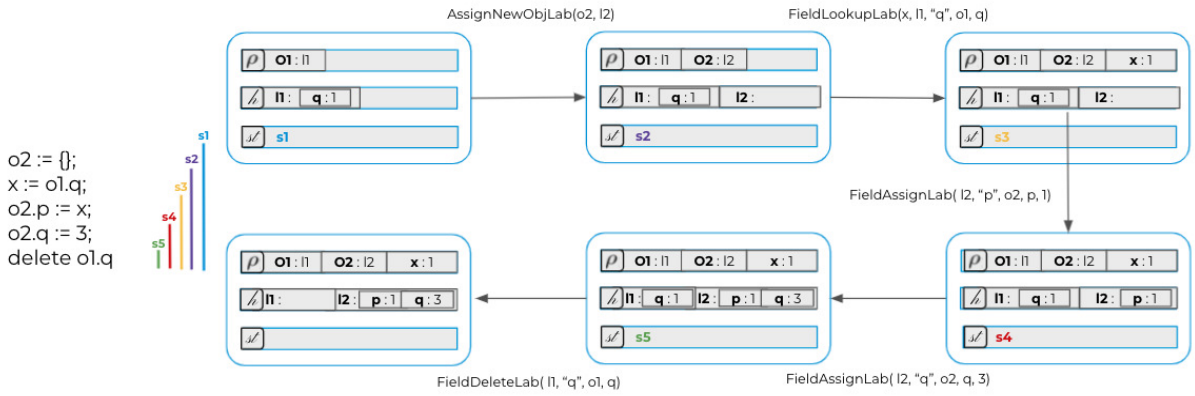

Figure 4. Execution Example

$$h \restriction^{\sigma}_{sh}$$

EMPTY HEAP

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

VISIBLE CELL WITH VISIBLE VALUE

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

INVISIBLE CELL WITH INVISIBLE VALUE

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

VISIBLE CELL WITH INVISIBLE VALUE

$$\frac{sh = sh' \uplus (l,p) \rightarrow (\sigma_1, \sigma_2) \qquad \sigma_1 \sqsubseteq \sigma \qquad \sigma_2 \not\sqsubseteq \sigma}{(h \uplus (l,p) \rightarrow v) \restriction^{\sigma}_{sh} \triangleq (l,p) \rightarrow ? \uplus h \restriction^{\sigma}_{sh'}}$$

Table 1. Low-Projection for Heaps

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 the top of Figure 5. 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 side of each field, while the value level annotates the arrow that connects the field to its corresponding value; for instance, the field **meta** of object **O1** has a low existence level and high value level. The bottom of Figure 5 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.

4.1 Security 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.

Bellow, we explain three rules of the *ECMA-SL* monitor, which illustrate how our monitor propagates security labels and enforces the no-sensitive-upgrade discipline.

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

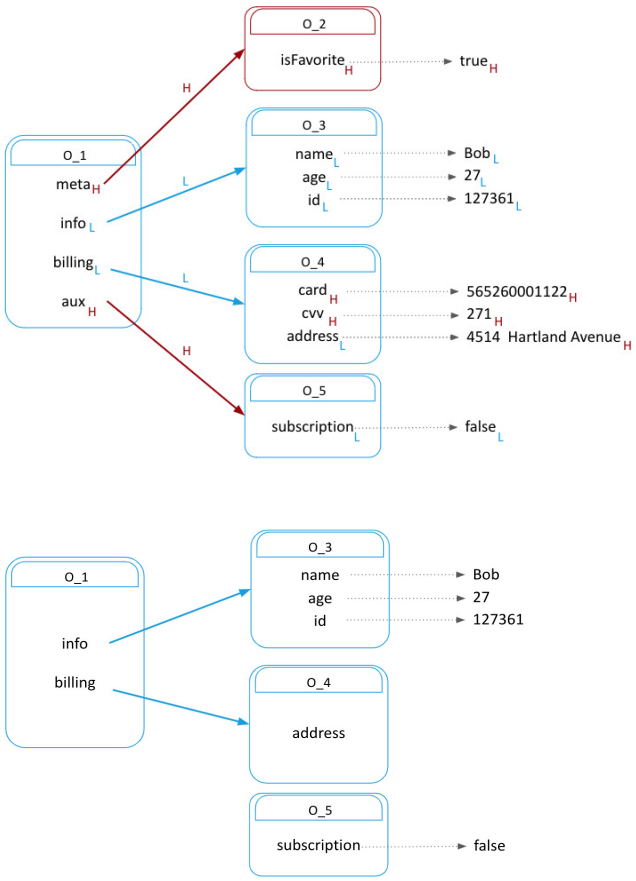

Figure 5. Low-projection example as a low-level observer

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.

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

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, the value level of the object's field is updated to $\sigma_v \sqcup \sigma_{ctx}$. If not, an information flow error is raised.

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

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. If the constraint is satisfied, the object's field is deleted; otherwise, an information flow error is raised.

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

To better understand the inner workings of our information flow monitor, we will now illustrate its application to one simple example. Figure 6 shows the monitored execution of the program given in Table 2. We represent the transitions of the semantics on the left and the transitions of the monitor on the right. Similar to semantic execution contexts, each security execution 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 execution context together with its corresponding security context; more precisely, semantic context on the left and security context on the right.

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.

4.2 Monitor Inlining

The monitor inlining approach achieves the same results as the lock-step monitor approach by compiling the given program

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

Table 2. Type II Program Example

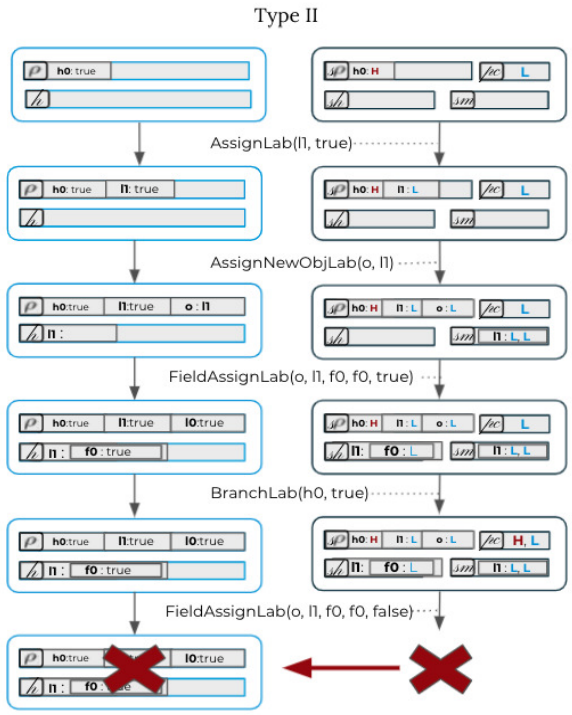

Figure 6. Monitored Execution Example

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 [9, 10, 14, 30, 43], 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 7 represents a labeled heap on the left and its instrumented counterpart on the right.

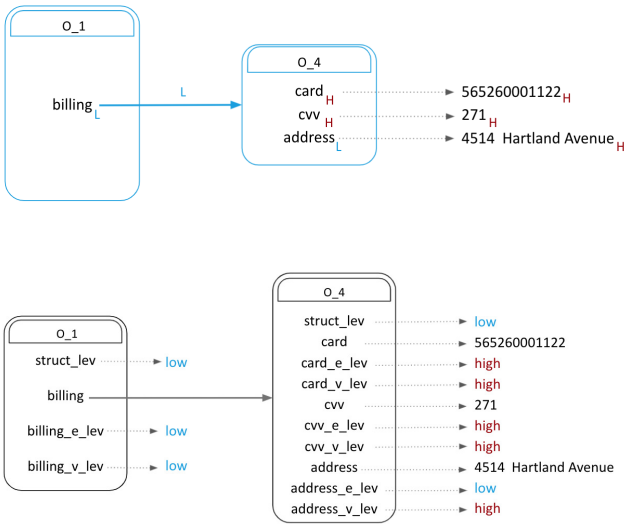

Figure 7. Inlining Trasformation on Projections

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`.

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:

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

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 [9, 10, 14, 30, 43]. Below we explain the *Variable Assignment* and

Conditional Statement 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, $\hat{x}$.

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

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}.

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

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. These classes are arranged according to the Composite pattern [17], which is applicable when dealing with class hierarchies where objects of a given subclass are composed of one or several objects of its super-class(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.

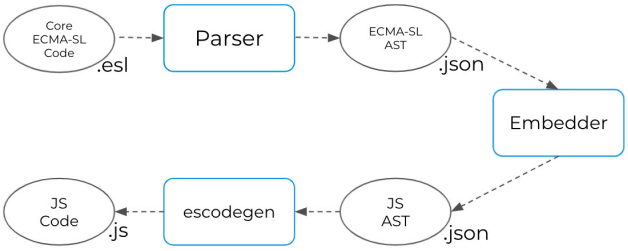

Figure 8. Shallow Embedding Pipeline

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.

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`.

6 Evaluation

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

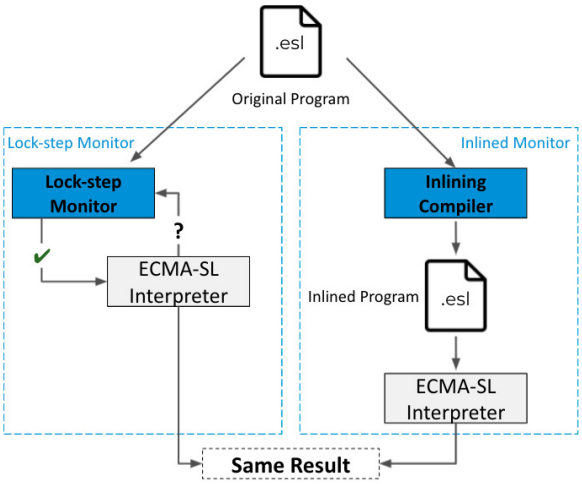

Figure 9. ECMA-SL Monitors Unit Test Pipeline

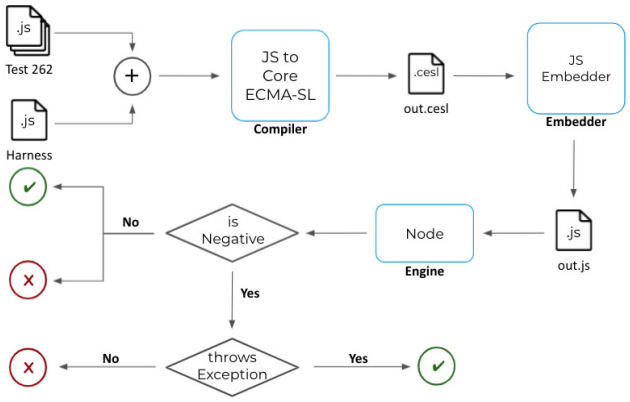

Figure 10. Embedding Test Pipeline

7 Test262

To test our shallow embedder, we used the Test262, the official ECMAScript test suite [1]. 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 [6, 33]. As the ECMAScript language is in constant evolution, Test262 also has to evolve to cover the new features of the language. Test262 comprises thousands of test files, often including multiple test cases.

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.

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.

To validate the correctness of the embedded file, it is important to confirm the expected result of its execution. Some programs must run in strict mode, therefore, the embedder test script must also signal the resulting program to run in strict mode. Another point worth mentioning is that the interpretation of a Test262 program can result in an exception. Consequently, the test passes if the original Test262 program is supposed to throw an exception and the interpretation embedded program also throws an exception.

7.1 Results

We divided our testing results into two tables, one for Expressions 3 and another for Statements 4.

Expressions Results These tests were split into 1400 positive tests, indicating that the execution is expected to end normally, and 31 negative tests, indicating that the execution is expected to throw an exception. Every test from the 1431 files behaved as expected, covering 100% of the Expressions test suite.

Statements Results These tests were split into 417 positive tests, indicating that the execution is expected to end normally, and 129 negative tests, indicating that the execution is expected to throw an exception. Every test from the 546 files behaved as expected, covering 100% of the Statements test suite.

8 Conclusion

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.

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 3. Expression Test Results

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
Expression	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
Whith	7	7	1	1
Total	417	417	129	129

Table 4. Test262 Statements Results

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 [22] information flow monitor and the JEST inlining compiler [10]. 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 [15, 16, 42] 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 [44].
- 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.

References

[1] [n.d.]. Test262 - Official ECMAScript Conformance Test Suite. https://github.com/tc39/test262/ Accessed on 2020-06-07.

[2] Thomas H. Austin and Cormac Flanagan. 2009. Efficient Purely-Dynamic Information Flow Analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS '09)*. Association for Computing Machinery, New York, NY, USA, 113–124. https://doi.org/10.1145/1554339.1554353

[3] Thomas H. Austin and Cormac Flanagan. 2010. Permissive Dynamic Information Flow Analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '10)*. Association for Computing Machinery, New York, NY, USA, Article Article 3, 12 pages. https://doi.org/10.1145/1814217.1814220

[4] Thomas H. Austin, Tommy Schmitz, and Cormac Flanagan. 2017. Multiple Facets for Dynamic Information Flow with Exceptions. *ACM Trans. Program. Lang. Syst.* 39, 3, Article Article 10 (May 2017), 56 pages. https://doi.org/10.1145/3024086

[5] Lennart Beringer and Martin Hofmann. 2007. Secure Information Flow and Program Logics. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF '07)*. IEEE Computer Society, USA, 233–248. https://doi.org/10.1109/CSF.2007.30

[6] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. Association for Computing Machinery, New York, NY, USA, 87–100. https://doi.org/10.1145/2535838.2535876

[7] Luca Cardelli. 1996. Type Systems. *ACM Comput. Surv.* 28, 1 (March 1996), 263–264. https://doi.org/10.1145/234313.234418

[8] H. Chen, A. Tiu, Z. Xu, and Y. Liu. 2018. A Permission-Dependent Type System for Secure Information Flow Analysis. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 218–232.

[9] Andrey Chudnov and David A. Naumann. 2010. Information Flow Monitor Inlining. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium (CSF '10)*. IEEE Computer Society, USA, 200–214. https://doi.org/10.1109/CSF.2010.21

[10] Andrey Chudnov and David A. Naumann. 2015. Inlined Information Flow Monitoring for JavaScript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. Association for Computing Machinery, New York, NY, USA, 629–643. https://doi.org/10.1145/2810103.2813684

[11] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243. https://doi.org/10.1145/360051.360056

[12] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC '14)*. Association for Computing Machinery, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755

[13] Mozilla Foundation. 1996. The SpiderMonkey Engine. https://spidermonkey.dev/.

[14] José Fragoso Femenin dos Santos. 2014. *Enforcing secure information flow in client-side Web applications*. Theses. Université Nice Sophia Antipolis. https://tel.archives-ouvertes.fr/tel-01135001

[15] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienundefined, Thomas Wood, and Philippa Gardner. 2017. JaVerT: JavaScript Verification Toolchain. *Proc. ACM Program. Lang.* 2, POPL, Article 50 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158138

[16] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *Proc. ACM Program. Lang.* 3, POPL, Article 66 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290379

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA.

[18] Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. 2012. Towards a Program Logic for JavaScript. *SIGPLAN Not.* 47, 1 (Jan. 2012), 31–44. https://doi.org/10.1145/2103621.2103663

[19] Roberto Giacobazzi and Isabella Mastroeni. 2004. 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)*. Association for Computing Machinery, New York, NY, USA, 186–197. https://doi.org/10.1145/964001.964017

[20] Google. 2017. The V8 JavaScript Engine. https://v8project.blogspot.ie/.

[21] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 126–150.

[22] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking Information Flow in JavaScript and Its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14)*. Association for Computing Machinery, New York, NY, USA, 1663–1671. https://doi.org/10.1145/2554850.2554909

[23] Daniel Hedin and A. Sabelfeld. 2012. A Perspective on Information-Flow Control. In *Software Safety and Security*.

[24] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

[25] Hui Jiang, Dong Lin, Xingyuan Zhang, and Xiren Xie. 2001. Type system in programming languages. *Journal of Computer Science and Technology* 16, 3 (01 May 2001), 286–292. https://doi.org/10.1007/BF02943207

[26] KDE. 1998. The Webkit Engine. https:///webkit.org/.

[27] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[28] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[29] Y. Liu and A. Milanova. 2010. 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*. 146–155.

[30] Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. 2010. On-the-fly Inlining of Dynamic Security Monitors. In *Security and Privacy – Silver Linings in the Cloud*, Kai Rannenberg, Vijay Varadharajan, and Christian Weber (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 173–186.

[31] Heiko Mantel. 2011. *Information Flow and Noninterference*. Springer US, Boston, MA, 605–607. https://doi.org/10.1007/978-1-4419-5906-5{_}874

[32] OpenSSL. 2011. OpenSSL 1.0.1 implementation. https:///openssl.org/.

[33] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th*

ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15). Association for Computing Machinery, New York, NY, USA, 346–356. https://doi.org/10.1145/2737924.2737991

[34] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[35] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. 2011. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *20th USENIX Security Symposium (USENIX Security 11)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/usenix-security-11/adsafety-type-based-verification-javascript-sandboxing

[36] Chromium Project. 2013. The Blink Web Engine. https://chromium.org/blink/.

[37] Vineet Rajani, Iulia Bastys, Willard Rafnsson, and Deepak Garg. 2017. Type Systems for Information Flow Control: The Question of Granularity. *ACM SIGLOG News* 4, 1 (Feb. 2017), 6–21. https://doi.org/10.1145/3051528.3051531

[38] J. C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

[39] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium (CSF '10)*. IEEE Computer Society, USA, 186–199. https://doi.org/10.1109/CSF.2010.20

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

[41] A. Sabelfeld and A. C. Myers. 2006. Language-Based Information-Flow Security. *IEEE J.Sel. A. Commun.* 21, 1 (Sept. 2006), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[42] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. 2018. Symbolic Execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18)*. Association for Computing Machinery, New York, NY, USA, Article 11, 14 pages. https://doi.org/10.1145/3236950.3236956

[43] José Fragoso Santos and Tamara Rezk. 2014. An Information Flow Monitor-Inlining Compiler for Securing a Core of JavaScript. In *ICT Systems Security and Privacy Protection*, Nora Cuppens-Boulahia, Frédéric Cuppens, Sushil Jajodia, Anas Abou El Kalam, and Thierry Sans (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–292.

[44] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. 2019. An Empirical Study of Information Flows in Real-World JavaScript. *CoRR* abs/1906.11507 (2019). arXiv:1906.11507 http://arxiv.org/abs/1906.11507

[45] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. 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.

[46] Steve Zdancewic. 2004. Challenges for information-flow security. In *In Proc. Programming Language Interference and Dependence (PLID*.