Secure multi-execution through static program transformation

Gilles Barthe[1], Juan Manuel Crespo[1], Dominique Devriese[2], Frank Piessens[2], and Exequiel Rivas[1]

[1] IMDEA Software Institute, Madrid, Spain
[2] IBBT-DistriNet Research Group, KU Leuven, Belgium

Abstract. Secure multi-execution (SME) is a dynamic technique to ensure secure information flow. In a nutshell, SME enforces security by running one execution of the program per security level, and by reinterpreting input/output operations w.r.t. their associated security level. SME is sound, in the sense that the execution of a program under SME is non-interfering, and precise, in the sense that for programs that are non-interfering in the usual sense, the semantics of a program under SME coincides with its standard semantics. A further virtue of SME is that its core idea is language-independent; it can be applied to a broad range of languages. A downside of SME is the fact that existing implementation techniques require modifications to the runtime environment, e.g. the browser for Web applications. In this article, we develop an alternative approach where the effect of SME is achieved through program transformation, without modifications to the runtime, thus supporting server-side deployment on the web. We show on an exemplary language with input/output and dynamic code evaluation (modeled after JavaScript's eval) that our transformation is sound and precise. The crux of the proof is a simulation between the execution of the transformed program and the SME execution of the original program. This proof has been machine-checked using the Agda proof assistant. We also report on prototype implementations for a small fragment of Python and a substantial subset of JavaScript.

1 Introduction

Information flow policies are confidentiality and integrity policies that constrain the propagation of data in programs. For instance, such policies can limit how public outputs can depend on confidential inputs, or how high integrity outputs can be influenced by low integrity inputs. A baseline confidentiality policy for information flow security is *non-interference*: given a labeling of input and output channels as either confidential (high, or H) or public (low, or L), a (deterministic) program is non-interferent if there are no two executions with the same public inputs (but possibly different confidential inputs) that lead to different public outputs. This definition of non-interference generalizes from two security levels H and L to an arbitrary partially ordered set of security levels.

Enforcing non-interference and other information flow policies is a challenging problem. Ideally, enforcement mechanisms should achieve potentially conflicting goals, including: i. *soundness:* no illicit flows should arise during execution; ii. *precision:* the execution of secure programs should not be prevented or altered; iii. *practicality:* the cost of the mechanism should be acceptable. Costs can be incurred at development time (for instance additional code annotations), at deployment time (for instance modifications to standard runtime environments) or at run time (for instance performance cost). Despite substantial attention from the research community for several decades, enforcement mechanisms achieving these goals simultaneously have remained elusive.

There are two main classes of enforcement mechanisms for information flow policies. *Static* mechanisms include security type systems [31,17,24], and verification-based approaches [5]. These techniques are sound, and do not incur run time or deployment time costs. However, type-based approaches are not precise, and reject many secure programs. In contrast, verification-based approaches may offer perfect precision (modulo completeness of the underlying program logic). However, both type-based and verification-based approaches have a substantial development time cost as they require annotations in the code. Moreover some language idioms, such as dynamic code evaluation, are not readily amenable to static information flow analysis.

Dynamic techniques, which have received renewed interest in recent years, include run-time monitors [16,29,3,10], and more recently *secure multi-execution (SME)* [14,8]. The cited techniques are sound, and can be more precise than some static techniques. For instance, run-time monitors reject fewer programs than type-based methods[29]; they also require less annotation effort. However, run-time monitors still may reject or alter the behavior of some secure programs. In contrast, SME offers perfect precision (at the cost of potentially modifying the behaviour of insecure programs); it is also practical for developers, since there is no need for security annotations of the code. However, SME is not easy to deploy, as all existing implementations of SME require modifications to the underlying computing infrastructure (OS [8], browser [6,2], virtual machine [14], trusted libraries [18]). Specifically, it is hard to deploy SME for distributed and heterogeneous infrastructures, such as the web.

The key contribution of this paper is a new implementation technique for SME based on static program transformation that eliminates the need to modify the computing infrastructure, while retaining its appealing theoretical properties.

A motivating example: JavaScript advertising

JavaScript code is used in web applications to perform client-side computations. In many scenarios, the fact that scripts run with the same privileges as the website loading the script leads to security problems. One important example are advertisements; these are commonly implemented as scripts and in the absence of security countermeasures, such scripts can leak any information present in the web page that they are part of.

JavaScript advertisements are a challenging application area for information flow security, as they may need some access to the surrounding web page (to be able to provide context-sensitive advertising), and as they can use all of JavaScript's features, including dynamic code evaluation, e.g. in the form of JavaScript's `eval` function, which Richards *et al.*[25] have shown to be widely used on the web. The following code snippet shows a very simple context-sensitive advertisement in JavaScript:

```
1 var keywords = document.getElementById("keywords").textContent;
2 var img = document.getElementById("adimage");
3 img.src = 'http://ads.com/SelectAd.php?keywords='+keywords
```

Line 1 looks up some keywords in the surrounding web page; these keywords will be used by the ad provider to provide a personalized, context-sensitive advertisement. Line 2 locates the element in the document in which the advertisement should be loaded, and finally line 3 generates a request to the advertisement provider site to generate an advertisement (in the form of an image) related to the keywords sent in the request.

Obviously, a malicious advertisement can easily leak any information in the surrounding page to the ad provider or to any third party. Here is a simple malicious ad that leaks the contents of a password field to the ad provider:

```
1 // Malicious: steal a password instead of keywords
2 var password = document.getElementById("password").textContent;
3 var img = document.getElementById("adimage");
4 img.src = 'http://ads.com/SelectAd.php?keywords='+password
```

Information flow security enforcement can mitigate this threat: if one labels the keywords as public information and the password as confidential information, then (treating the network as a public output) enforcing non-interference will permit the non-malicious ad, but block the malicious one.

The example ad script above loads an image from a third-party server. Instead of loading an image, it could also load a script from the server that can then render the ad and further interact with the user (e.g. make the advertisement react to mouse events). In the example below, we illustrate the essence of this technique using the `XMLHttpRequest` API and JavaScript `eval`.

```
1 var keywords = document.getElementById("keywords").textContent;
2 var xmlhttp = new XMLHttpRequest();
3 xmlhttp.open('GET', 'http://ads.com/getAd.php?keywords='+keywords, false);
4 xmlhttp.send(null);
5 eval(xmlhttp.responseText)
```

Lines 2-4 send the keywords to the ad provider, and expect a (personalized) script in response. Line 5 then evaluates the script that was received – and this script could of course be malicious too and try to leak information. Dealing with dynamic generation or loading of new code and its on the fly evaluation further complicates the enforcement of information flow security policies. In particular, since the code to be executed is not available offline, static techniques do not apply.

The enforcement mechanism we develop in this paper will provide effective protection against these security problems of malicious scripts. We propose a program transformation that transforms any script into a script that (1) is guaranteed to be non-interferent, and (2) behaves identically to the original script if that script was non-interferent to begin with.

Summary of contributions

In summary, the main contributions of this paper are:

– We show that standard SME [14] is sound and precise for a language including dynamic code evaluation.
– We propose a program transformation for sequential programs that simulates the effect of SME, and provide a machine-checked correctness proof.
– We report on two prototype implementations of this program transformation.
– We define a variant of the transformation that targets a concurrent programming language, and prove it correct.

The paper is organized as follows: Section 2 introduces our programming language and defines non-interference. Section 3, 4, 5 and 6 each cover one of the contributions above. Related work is discussed in Section 7.

2 Setting

Syntax. Following [14], a program P is simply a command to be executed by the system. The syntax of commands is defined as follows:

$$\begin{aligned} c ::=\ & x := e \mid \textbf{input}\ x\ \textbf{from}\ ic \mid \textbf{output}\ e\ \textbf{to}\ oc \mid c; c \\ \mid\ & \textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ c \mid \textbf{while}\ b\ \textbf{do}\ c \mid \textbf{skip} \mid \textbf{eval}(e) \end{aligned}$$

Most commands are standard, with the exception of $\textbf{input}\ x\ \textbf{from}\ ic$, that assigns the next input from the input channel ic to x, and $\textbf{output}\ e\ \textbf{to}\ oc$, that outputs the value of the expression e to the output channel oc—we assume that input and output channels are disjoint. The main extension w.r.t. [14] is the instruction $\textbf{eval}(e)$, which takes an integer encoding e of a program (in a real language this would be the usual string encoding), decodes it and evaluates it.

Example 1. The command below models a program that exhibits both of the attacks presented in the introduction: the script sends private information (the password) across a public channel (the network) to the ad provider and then receives a (possibly malicious) script which is executed with the same privilege.

$$\begin{array}{l} \textbf{input}\ keys\ \textbf{from}\ LKeys; \\ \textbf{input}\ pass\ \textbf{from}\ HPass; \\ \textbf{output}\ keys + pass\ \textbf{to}\ LReq; \\ \textbf{input}\ res\ \textbf{from}\ LReq'; \\ \textbf{eval}(res) \end{array}$$

Semantics For simplicity, we assume that expressions are side-effect free, and that they are used with their correct types—e.g. guards of branching statements and loops are boolean expressions. The semantics of expressions is defined as a mapping from memories to values or bottom, where a memory is a (well-typed) mapping from variables to values. Formally, we let $[\![e]\!]\ m$ be the evaluation of e in memory m.

The operational behavior of programs is modelled as a transition relation $\rightsquigarrow$ between configurations. Formally, a configuration is a 5-tuple $\langle c, m, p, I, O\rangle$, where c is a command, m is a memory, I and O are program inputs and outputs, i.e. mappings from input and output channels respectively to lists of values, and p is an input pointer, i.e. a mapping from input channels to natural numbers, that points to the next input to be consumed. A configuration is initial if it is of the form $\langle c, m_0, p_0, I, O_0\rangle$, where m_0 maps every variable to a default value, e.g. 0 for integer variables, p_0 maps every input channel to 0, and O_0 maps every output channel to the empty list.

Fig. 1 provides an excerpt of the transition rules that define the operational semantics.The rules make use of an operation $\mathbf{decode}$ that turns an integer into a command, and of primitive operations for reading and writing from a channel (we use the notation $l_1 +\!\!+ l_2$ for appending two lists):

$$\mathbf{read}(I, ic, p) = I(ic)(p(ic)) \qquad\qquad \mathbf{write}(O, oc, v) = O[oc \mapsto O(oc) +\!\!+ [v]]$$

We say that an execution of the program P with input I terminates with input pointer p and program output O, and write $\langle P, I\rangle \rightsquigarrow^* \langle p, O\rangle$, iff $\langle P, m_0, p_0, I, O_0\rangle \rightsquigarrow^* \langle \mathbf{skip}, m, p, I, O\rangle$ for some memory m.

$$\frac{}{\langle \mathbf{input}\ x\ \mathbf{from}\ ic, m, p, I, O\rangle \rightsquigarrow \langle \mathbf{skip}, m[x \mapsto \mathbf{read}(I, ic, p)], p[ic \mapsto p(ic)+1], I, O\rangle}$$

$$\frac{}{\langle \mathbf{output}\ e\ \mathbf{to}\ oc, m, p, I, O\rangle \rightsquigarrow \langle \mathbf{skip}, m, p, I, \mathbf{write}(O, oc, [\![e]\!]m)\rangle}$$

$$\frac{\langle c_1, m, p, I, O\rangle \rightsquigarrow \langle c_1', m', p', I, O'\rangle}{\langle c_1; c_2, m, p, I, O\rangle \rightsquigarrow \langle c_1'; c_2, m', p', I, O'\rangle}$$

$$\frac{}{\langle \mathbf{skip}; c_2, m, p, I, O\rangle \rightsquigarrow \langle c_2, m, p, I, O\rangle}$$

$$\frac{}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, m, p, I, O\rangle \rightsquigarrow \langle c; \mathbf{while}\ b\ \mathbf{do}\ c, m, p, I, O\rangle}\ [\![b]\!]m$$

$$\frac{}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, m, p, I, O\rangle \rightsquigarrow \langle \mathbf{skip}, m, p, I, O\rangle}\ \neg[\![b]\!]m$$

$$\frac{}{\langle \mathbf{eval}(e), m, p, I, O\rangle \rightsquigarrow \langle \mathbf{decode}([\![e]\!]m), m, p, I, O\rangle}$$

Fig. 1: Operational semantics (excerpt).

Security The notion of program security is defined relative to a partially ordered set $(\mathcal{L}, \leq)$ of security levels, and mappings σ_{in} and σ_{out} from input and output channels to security levels. The mappings induce equivalence relations on inputs, outputs, and input pointers; informally, two inputs, outputs, and input pointers are equal w.r.t. a security level l if they cannot be distinguished by an adversary that has access to channels of level l and lower. Formally, two program inputs I and I' are equal up to l (written

$I =_l I'$) iff $I(i) = I'(i)$ for all input channels i such that $\sigma_{in}(i) \leq l$. Likewise, two program outputs O and O' are equal up to l (written $O =_l O'$) iff $O(o) = O'(o)$ for all output channels o such that $\sigma_{out}(o) \leq l$. Finally, two input pointers p and p' are equal up to l (written $p =_l p'$) iff $p(i) = p'(i)$ for all input channels i such that $\sigma_{in}(i) \leq l$.

Definition 1 (Non-interference). *A program P is non-interferent with respect to an execution relation $\Rightarrow^*$ (mapping programs and inputs to input pointers and outputs) if for all security levels $l \in \mathcal{L}$, for all l-equal inputs I and I', i.e. $I =_l I'$, we have that $(P, I) \Rightarrow^* (p_f, O_f)$ if and only if $(P, I') \Rightarrow^* (p'_f, O'_f)$ and $p_f =_l p'_f$ and $O_f =_l O'_f$.*

Note that this definition is *termination-sensitive*: it does not allow termination to depend on information at non-minimal levels. The definition of non-interferent program is obtained by instantiating $\Rightarrow^*$ to $\rightsquigarrow^*$. Example 1 is clearly *not* non-interferent.

3 Secure Multi-Execution: the operational approach

We extend the theoretical results of [14] and show that SME remains sound and precise in the presence of dynamic code evaluation.

SME by Example The central insight of SME is that non-interference can be enforced by executing programs once per security level. In order to guarantee non-interference, the execution at security level l only performs inputs and outputs to channels at level l; moreover, inputs from channels with security levels l' such that $l' \not\leq l$ are replaced by default values and inputs from channels of security levels l' such that $l' < l$ are delayed until the execution corresponding to security level l' reads from them—the result is then available to be reused at security level l.

The precision of SME intuitively follows from the fact that for non-interferent programs, the behavior of the program visible at a level l is by definition not influenced by changes to information at levels not lower than l. Therefore, the execution at any level l will still produce the same behavior at level l as the standard execution of the program, since it receives the same input on all levels lower than l.

Figure 2 illustrates the effect of SME on the malicious script from Section 1 and the two-points lattice of security levels $\{L, H\}$, with $L \leq H$. We treat reading the content of the `password` textbox as input at security level H and setting the URL of the image as output at level L. Hence, the SME execution of the program at level L will receive a default value rather than the real content of the `password` textbox. Subsequently, the execution at level L will compute as URL of the image a value that does not contain any information about the real user password. On the contrary, the execution of the script at security level H does receive the real input, and further computations at level H will be performed based on the password; however, the execution does not output to low channels.

Operational semantics of SME. Secure multi-execution is described formally through an operational semantics, and is parametrized by a lattice of security levels $\mathcal{L}$, and mappings σ_{in} and σ_{out} associating input and output channels to security levels respectively.

Execution at L security level.

```
// Malicious: steal a password instead of keywords
var password = ~~document.getElementById("password").textContent~~ undefined;
var img = document.getElementById("adimage");
img.src = 'http://ads.com/SelectAd.php?keywords='+password
```

Execution at H security level.

```
// Malicious: steal a password instead of keywords
var password = document.getElementById("password").textContent
var img = document.getElementById("adimage");
~~img.src = 'http://ads.com/SelectAd.php?keywords='+password~~
```

Fig. 2: Secure Multi-Execution of malicious JavaScript program from Section 1.

The operational semantics combines a local semantics, and a global semantics. The initial configuration includes a local configuration per security level; each local configuration runs independently of the other, except for input/output operations, where synchronization is needed. The global semantics capture the synchronization enforced by SME, and are defined relative to a scheduler **select** that, given a set of local configurations, picks the next one to execute. In their work, Devriese and Piessens [14] focus on a scheduler $\mathbf{select}_{\text{lowprio}}$ which picks the local configuration corresponding to the lowest security level; other schedulers are considered in [19].

The local semantics are defined as a relation between pairs of local configurations and global states. Local configurations are of the form $\langle c, m, p\rangle_l$, where c is a command, m is a memory and p is a local input pointer and l is the security level associated to the local configuration. Global states consist of a global input pointer r, a program input I and a program output O. The global input pointer r tracks actual input consumption. I.e. for an input channel ic, $r(ic)$ equals $p(ic)$ where p is the local input pointer of the execution at level $\sigma_{in}(ic)$. For details about the semantics, we refer the reader to the original SME paper [14]. The only novelty is the rule in the local semantics for eval:

$$\frac{[\![e]\!]m = v \qquad \mathbf{decode}(v) = c}{\langle eval(e), m, p\rangle_l, r, I, O \Rightarrow\!\!\triangleright \langle c, m, p\rangle_l, r, I, O}$$

The global semantics are defined as a relation between configurations. The latter are of the form $\langle L, wq, r, I, O\rangle$, where r, I, O form the global state, L is a set of local configurations, and wq is a queue that maps input channels and message numbers to local configurations waiting for that message to be input. Again, the details are in [14].

We say that a set of local configurations C with input I terminates with final input pointer r_f and program output O_f, and write $\langle C, I\rangle \Rightarrow\!\!\triangleright^* \langle r_f, O_f\rangle$, if

$$\langle C, wq_0, r_0, I, O_0\rangle \Rightarrow\!\!\triangleright^* \langle [], wq_f, r_f, I, O_f\rangle$$

for some final waiting queue wq_f and where r_0 is the global input pointer mapping all input channels to position 0.

The secure multi-execution of a program P is defined using the global semantics; specifically, we introduce for every program P and security level l the local configuration $P_l = \langle P, m_0, p_0 \rangle_l$, where m_0 is the default memory—as defined in Section 2—and p_0 maps all input channels to 0. Then, we introduce the set of local configurations $P_{lcinit} = [P_{l_1}, \ldots, P_{l_k}]$ where $l_1 \ldots l_k$ is an enumeration of the security levels. Then, we say that the secure multi-execution of the program P with input I terminates with final input pointer r_f and final program output O_f, and write $\langle P, I \rangle \Rightarrow\!\!\triangleright^* \langle r_f, O_f \rangle$ iff $\langle P_{lcinit}, I \rangle \Rightarrow\!\!\triangleright^* \langle r_f, O_f \rangle$.

Soundness and precision SME provides strong security and operational guarantees.

Theorem 1 (Soundness of SME). *For a totally ordered $\mathcal{L}$, any program P is non-interferent under SME, using the* $\mathbf{select}_{lowprio}$ *scheduler.*

The $\mathbf{select}_{\text{lowprio}}$ scheduler requires a total ordering on security levels. If $\mathcal{L}$ is not totally ordered, then it can be extended to a total order in order to apply SME with the $\mathbf{select}_{\text{lowprio}}$ scheduler. In that case execution of P under SME is termination-*insensitively* non-interferent, but termination information may leak between non-comparable levels of $\mathcal{L}$ [19].

Theorem 2 (Precision of SME). *Let P be a non-interferent program. Then, for all program input I, input pointer p_f and program output, O_f, $\langle P, I \rangle \rightsquigarrow^* \langle p_f, O_f \rangle$ implies $\langle P, I \rangle \Rightarrow\!\!\triangleright^* \langle p_f, O_f \rangle$.*

The proofs follow along the lines of [14]; additional cases for $\mathbf{eval}$ follow by a direct argument.

4 Secure Multi-Execution by program transformation

The instrumented semantics of Section 3 provides a direct, operational interpretation of the effect of secure multi-execution on programs. In this section, we explore an alternative approach in which a program P of the source language is transformed into a program P' whose behavior matches the behavior of P under SME execution. Our results show that one can achieve soundness and precision without modifying the runtime environment.

Informally, one defines for each program P and security level l a transformed program $\mathbf{Tr}(P, l)$ and defines $\mathbf{Tr}(P)$ as the sequential composition of the commands $\mathbf{Tr}(P, l)$, where l ranges over security levels from low to high. This mimicks execution under the SME semantics with the $\mathbf{select}_{\text{lowprio}}$ scheduler. We assume that this sequential composition is done in the same order as the order in which the $\mathbf{select}_{\text{lowprio}}$ scheduler selects executions. For a totally ordered $\mathcal{L}$, this order is fixed, but non-comparable levels can be scheduled in different ways.

SME requires the buffering of inputs so that these inputs can be reused by executions running at higher security levels. We implement these buffers as global lists ($list_{ic}$) and the global input pointer as well as local input pointers are represented as global integer variables ($count_{ic}$ and $count_{ic,l}$ respectively).

For commands that do not perform input/output operations, the command $\mathbf{Tr}(P,l)$ executes P "locally". Specifically, for each variable x of the source program, we introduce variables x_l, where l ranges over security levels; informally, x_l is the local copy of x for the execution corresponding to security level l. Then, we ensure that $\mathbf{Tr}(P,l)$ reads and writes only from/to variables indexed by l. For instance, the transformation of an assignment is defined by the clause:

$$\mathbf{Tr}(x := e, l) = x_l := [e]_l$$

where $[e]_l$ is obtained by replacing occurrences of each variable (say x) by its l-indexed variant (say x_l). The definition of the transformation extends recursively to sequences, branching statements, and loops. In the case of dynamic code evaluation, $\mathbf{Tr}(\mathbf{eval}(e), l)$ should informally compute the value of e locally at level l, decode the resulting value into a command c, compute $c' = \mathbf{Tr}(c, l)$, encode c' into an integer n', and return $\mathbf{eval}(n')$. Hence, $\mathbf{Tr}(\mathbf{eval}(e), l)$ should intuitively be of the form:

$$n := [e]_l; c := \mathbf{decode}(n); c' := \mathbf{Tr}(c, l); n' := \mathbf{encode}(c'); \mathbf{eval}(n')$$

The code snippet is ill-typed and ill-defined in our exemplary language. In a full-fledged language such as JavaScript, one can make the above snippet meaningful, by implementing encoding and decoding functions from strings and abstract syntax trees, and the transformation given by the rules of Fig. 3. For the purpose of this section, we gloss over the details of such implementations and assume the existence for each security level l of a unary operator $\mathbf{trans}_l$ from integers to integers, and define

$$\mathbf{Tr}(\mathbf{eval}(e), l) = \mathbf{eval}(\mathbf{trans}_l([e]_l))$$

Moreover, we assume that $\mathbf{trans}_l$ is correct, i.e. for every integer value k,

$$\mathbf{decode}(\mathbf{trans}_l(k)) = \mathbf{Tr}(\mathbf{decode}(k), l)$$

The most interesting cases of the transformation are for input and output commands. For the latter, $\mathbf{Tr}(P,l)$ is defined by case analysis on the security level of the output channel: a command $\mathbf{output}\ e\ \mathbf{to}\ oc$ is transformed into $\mathbf{output}\ [e]_l\ \mathbf{to}\ oc$ if oc has security level l, and into a $\mathbf{skip}$ statement otherwise. Similarly, for input statements, we define the transformation by case analysis on the security level l' of the input channel—as in the definition of SME. If $l' \not\leq l$, then the input statement is transformed into an assignment of a default value. If $l = l'$, then the transformed command performs the input statement and updates the list of available inputs and the counter representing the number of messages already read from this channel. Finally, if $l' < l$, the transformed command reuses a buffered input value and updates the corresponding counter. Executing the programs $\mathbf{Tr}(P,l)$ sequentially in the order from low to high in an initial memory in which every $count$ variable has value 0 and every $list$ variable is associated with the empty list, will simulate SME execution under the $\mathbf{select}_{\text{lowprio}}$ scheduler.

$$\mathbf{Tr}(P) = \fatsemi\{\mathbf{Tr}(P,l) \mid l \in \mathcal{L}\}$$

Example 2. We apply the transformation to Example 1. The sequential program obtained is shown in Fig. 4.

Formally, we can prove the following theorems.

Theorem 3. *For every program P and program input I:*

1. *if $\langle \mathbf{Tr}(P), I\rangle \rightsquigarrow^* \langle p, O\rangle$ then $\langle P, I\rangle \Rightarrow\!\triangleright^* \langle p, O\rangle$;*
2. *if P is non-interferent and $\langle P, I\rangle \Rightarrow\!\triangleright^* \langle p, O\rangle$ then $\langle \mathbf{Tr}(P)\rangle \rightsquigarrow^* \langle p, O\rangle$.*

We have developed a mechanized proof using Agda, a proof assistant based on the Curry-Howard isomorphism. We refer the reader to the extended version of this paper [4].

The following is an easy corollary of Theorem 3.

Corollary 1. *Statically enforced sequential SME is sound and precise.*

Proof. Soundness follows from Theorem 3, first part and Theorem 1. Precision follows from Theorem 3, second part and Theorem 2. □

5 Implementation

In order to validate our approach, we have developed two prototype implementations. Our first implementation considers a restricted fragment of Python; the fragment essentially corresponds to our exemplary language, with I/O functions `input` and `print` added as built-in functions. It does not support any of Python's more advanced features, but was useful to provide a baseline implementation.

Our second implementation supports a fragment of JavaScript including `eval()`. Both implementations were tested for security and for precision by means of small test scenarios.

We briefly comment on some aspects of the implementations.

Aliasing The soundness of our transformation relies on applying specific rules for I/O operations. In richer languages such as Python or JavaScript, aliasing becomes a major problem as one cannot statically determine where such operations will be called. To avoid this issue, and to be able to identify I/O operations, we proceed in two steps: first, we wrap primitive I/O functions upfront, i.e. the wrapped function will behave according to the security level associated to the context in which is called. Second, programs are only given access to these wrapped functions. This is achieved using Google Caja [23], which guarantees that the translated program only gets access to properly wrapped APIs. Google Caja will rewrite ("cajole") a program in such a way that it can be guaranteed capability secure, i.e. the modified program will only be able to call API functions which it is passed a reference to and otherwise be isolated from other code.

$$
\begin{array}{ll}
\mathbf{Tr}(x := e, l) & = x_l := [e]_l \\[1ex]
\mathbf{Tr}(\mathbf{output}\ e\ \mathbf{to}\ oc, l) & = \begin{cases} \mathbf{output}\ [e]_l\ \mathbf{to}\ oc & \text{if } \sigma_{out}(oc) = l \\ \mathbf{skip} & \text{otherwise} \end{cases} \\[1ex]
\mathbf{Tr}(\mathbf{input}\ x\ \mathbf{from}\ ic, l) & = \begin{cases} x_l := dv & \text{if } \sigma_{in}(ic) \not\leq l \\[1ex] \mathbf{input}\ x\ \mathbf{from}\ ic; \\ list_{ic} := list_{ic} +\!\!+ [x_l]; & \text{if } \sigma_{in}(ic) = l \\ count_{ic} := count_{ic} + 1 \\[1ex] x_l := list_{ic}[count_{ic,l}]; \\ count_{ic,l} := count_{ic,l} + 1\} & \text{if } \sigma_{in}(ic) < l \end{cases} \\[1ex]
\mathbf{Tr}(c_1; c_2, l) & = \mathbf{Tr}(c_1, l); \mathbf{Tr}(c_2, l) \\[1ex]
\mathbf{Tr}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, l) & = \mathbf{if}\ [b]_l\ \mathbf{then}\ \mathbf{Tr}(c_1, l)\ \mathbf{else}\ \mathbf{Tr}(c_2, l) \\[1ex]
\mathbf{Tr}(\mathbf{while}\ b\ \mathbf{do}\ c, l) & = \mathbf{while}\ [b]_l\ \mathbf{do}\ \mathbf{Tr}(c, l) \\[1ex]
\mathbf{Tr}(\mathbf{skip}, l) & = \mathbf{skip} \\[1ex]
\mathbf{Tr}(\mathbf{eval}(e), l) & = \mathbf{eval}(\mathbf{trans}_l([e]_l))
\end{array}
$$

Fig. 3: Syntactic program transformation.

```
input $keys_L$ from $LKeys$;
$list_{LK} := list_{LK} +\!\!+ [keys_L]$;
$count_{LK} := count_{LK} + 1$;
$pass_L := dv$;
output $keys_L + pass_L$ to $LReq$;
input $res_L$ from $LReq'$;
$list_{LR'} := list_{LR'} +\!\!+ [res_L]$;
$count_{LR'} := count_{LR'} + 1$;
eval(trans$_L(res_L)$);
$keys_H := list_{LK}[count_{LK,H}]$;
$count_{LK,H} := count_{LK,H} + 1$;
input $pass_H$ from $HPass$;
$list_{HP} := list_{HP} +\!\!+ [pass_H]$;
$count_{HP} := count_{HP} + 1$;
$res_H := list_{LR'}[count_{LR',H}]$;
$count_{LR',H} := count_{LR',H} + 1$;
eval(trans$_H(res_H)$)
```

Fig. 4: Static transformation applied to malicious ad.

Dynamic code evaluation Our prototype supports an `eval` function (JavaScript's well-known dynamic code evaluation primitive). Since Google Caja does not support dynamic code evaluation, we have developed our own *ad hoc* solution. Our `eval` takes as input a string of code, and sends it to a remote Caja cajoling service; the transformed code is then executed with the same wrapped APIs as the calling code. This proof-of-concept implementation is admittedly inefficient but arguably secure (assuming the calls to Google's cajoling service are reliable) and supports the entire subset of JavaScript that Google Caja supports.

Document Object Model (DOM) The Document Object Model (DOM) APIs that a browser exposes to scripts is structured as a tree corresponding to the HTML structure of the document. The DOM tree can be inspected and modified from within JavaScript. Our prototype supports a limited, read-only, version of the DOM. In particular, it allows the hosting page to assign security levels to parts of the document. The scripts can access the hosting document according to this policy and perform synchronous XML-HttpRequests. Our coverage of the DOM is sufficient for our examples.

Many DOM APIs allow web applications to register callback functions, which will be executed when certain (network, user or other) events occur; Bielova *et al.* [6] discuss how events and callbacks can be supported under secure multi-execution. Extending our transformation to address events and callbacks, and provide support for the full DOM is a significant engineering challenge, which we regard as future work.

6 Transformation to a concurrent language

The transformation defined earlier simulates SME with the $\mathbf{select}_{\text{lowprio}}$ scheduler. Kashyap et al. [19] have shown that other scheduling strategies can be useful too. In this section, we present a variant of our transformation towards a language that supports concurrency in order to enable the use of more scheduling strategies.

This revised transformation still takes programs in the sequential subset of the language as input. The concurrency features are only used in the output of the transformation.

Target language. We extend our command language with the following syntax:

$$\begin{aligned} c &::= \ldots \mid \mathbf{await}\ b\ \mathbf{then}\ c \\ P &::= \parallel (id, c)^* \end{aligned}$$

Intuitively, the command $\mathbf{await}\ b\ \mathbf{then}\ c$ executes c atomically, provided b holds, and blocks otherwise. Then, a program is simply a set of threads; for convenience, we assume that each thread is tagged with a unique identifier. In what follows, we write $\mathbf{atomic}\ c$ as a shorthand for $\mathbf{await}\ \mathbf{true}\ \mathbf{then}\ c$.

The operational behavior of programs is modelled as a transition between configurations. A configuration is a 5-tuple consisting of a program P, a waiting queue wq mapping guards to commands, an input pointer p, a program input I and a program output O. Figure 5 presents the semantics of the language. The thread-local semantics is similar to our sequential language; note however that we introduce another rule for sequence in order to propagate the emission of signals induced by **await** commands. The rules for the latter are standard; if the guard holds, then the body of the command is executed atomically. Otherwise, the command blocks and emits a signal, namely the guard in which its blocked. Upon the emission of a signal, the global semantics then inserts the blocked thread associated with the guard into the waiting queue. Further changes in global state trigger the re-evaluation of guards, and threads associated with guards that become true are moved back to the ready list.

We say that an execution of the program P with input I terminates with input pointer p and program output O, and write $\langle P, I\rangle \rightsquigarrow^* \langle p, O\rangle$, if there exists some memory m such that

$$\langle P, wq_0, m_0, p_0, I, O_0\rangle \rightsquigarrow^* \langle [], wq_0, m, p, I, O\rangle$$

$$\frac{\langle c_1, m, p, I, O\rangle \overset{b}{\rightsquigarrow} \langle c_1', m, p, I, O\rangle}{\langle c_1; c_2, m, p, I, O\rangle \overset{b}{\rightsquigarrow} \langle c_1'; c_2, m, p, I, O\rangle}$$

$$\frac{\langle c, m, p, I, O\rangle \rightsquigarrow^* \langle \mathbf{skip}, m', p', I, O'\rangle}{\langle \mathbf{await}\ b\ \mathbf{then}\ c, m, p, I, O\rangle \rightsquigarrow \langle \mathbf{skip}, m', p', I, O'\rangle}\ [\![b]\!]m$$

$$\frac{}{\langle \mathbf{await}\ b\ \mathbf{then}\ c, m, p, I, O\rangle \overset{b}{\rightsquigarrow} \langle \mathbf{await}\ b\ \mathbf{then}\ c, m, p, I, O\rangle}\ \neg[\![b]\!]m$$

(a) Thread-local semantics (excerpts)

$$\frac{\mathbf{select}(P) = (id, \mathbf{skip})}{\langle P, wq, m, p, I, O\rangle \rightsquigarrow \langle P\backslash\{(id, \mathbf{skip})\}, wq, m, p, I, O\rangle}$$

$$\frac{\mathbf{select}(P) = (id, c) \qquad \langle c, m, p, I, O\rangle \overset{b}{\rightsquigarrow} \langle c, m, p, I, O\rangle}{\langle P, wq, m, p, I, O\rangle \rightsquigarrow \langle P\backslash\{(id, c)\}, wq \cup \{(b, (id, c))\}, m, p, I, O\rangle}$$

$$\frac{\begin{array}{c}\mathbf{select}(P) = (id, c) \qquad \langle c, m, p, I, O\rangle \rightsquigarrow \langle c', m', p', I, O'\rangle \\ P' = P\backslash\{(id, c)\} \cup \{(id, c')\} \cup \{(id^*, c^*) | (b, (id^*, c^*)) \in wq \wedge [\![b]\!]m'\} \\ wq' = \{(b, (id^*, c^*)) | (b, (id^*, c^*)) \in wq \wedge \neg[\![b]\!]m'\}\end{array}}{\langle P, wq, m, p, I, O\rangle \rightsquigarrow \langle P', wq', m', p', I, O'\rangle}$$

(b) Global semantics

Fig. 5: Extended semantics.

The transformation. Adapting our transformation to target the concurrent case requires only two changes. First, input will now perform synchronization:

$$\mathbf{Tr^{con}}(\mathbf{input}\ x\ \mathbf{from}\ ic, l) = \begin{cases} x_l := dv & \text{if } \sigma_{in}(ic) \not\leq l \\ \mathbf{atomic}\ (\mathbf{input}\ x_l\ \mathbf{from}\ ic; \\ list_{ic} := list_{ic} +\!\!+ [x_l]; count_{ic} := count_{ic} + 1) & \text{if } \sigma_{in}(ic) = l \\ \mathbf{await}\ count_{ic,l} < count_{ic}\ \mathbf{then} \\ (x_l := list_{ic}[count_{ic,l}]; count_{ic,l} := count_{ic,l} + 1) & \text{if } \sigma_{in}(ic) < l \end{cases}$$

$$\mathbf{Tr^{con}}(\mathbf{output}\ e\ \mathbf{to}\ oc, l) = \begin{cases} \mathbf{output}\ [e]_l\ \mathbf{to}\ oc & \text{if } \sigma_{out}(oc) = l \\ \mathbf{skip} & \text{otherwise} \end{cases}$$

Second, instead of defining the overall transformation as a sequential composition, we define it as a parallel one, i.e. $\mathbf{Tr}^{\mathbf{con}}(P) = \| \{(l, \mathbf{Tr}^{\mathbf{con}}(P, l)) \mid l \in \mathcal{L}\}$.

Example 3. Consider our running example, the malicious ad. Applying the transformations to the example w.r.t. security levels L and H yields the two programs shown in Fig. 6a and Fig. 6b respectively.

```
**atomic** {
  **input** $keys_L$ **from** $LKeys$;
  $list_{LK} := list_{LK} +\!\!+ [keys_L]$;
  $count_{LK} := count_{LK} + 1$};
$pass_L := dv$;
**output** $keys_L + pass_L$ **to** $LReq$;
**atomic** {
  **input** $res_L$ **from** $LReq'$;
  $list_{LR'} := list_{LR'} +\!\!+ [res_L]$;
  $count_{LR'} := count_{LR'} + 1$};
**eval**(**trans**$_L(res_L)$);
```

(a) Security level L.

```
**await** $count_{LK,H} < count_{LK}$ **then** {
  $keys_H := list_{LK}[count_{LK,H}]$;
  $count_{LK,H} := count_{LK,H} + 1$};
**atomic** {
  **input** $pass_H$ **from** $HPass$;
  $list_{HP} := list_{HP} +\!\!+ [pass_H]$;
  $count_{HP} := count_{HP} + 1$};
**await** $count_{LR',H} < count_{LR'}$ **then** {
  $res_H := list_{LR'}[count_{LR',H}]$;
  $count_{LR',H} := count_{LR',H} + 1$};
**eval**(**trans**$_H(res_H)$);
```

(b) Security level H.

Fig. 6: Static transformation applied to malicious ad.

The revised transformation again yields executions equivalent to secure multi-execution, now for any scheduling strategy. The proof relies on a simulation result and hinges on the assumption that (informally) schedulers pick the same threads to execute.

Theorem 4. *For every program P, and program input I:*

1. *if* $\langle \mathbf{Tr}^{\mathbf{con}}(P), I\rangle \rightsquigarrow^* \langle p, O\rangle$ *then* $\langle P, I\rangle \Rrightarrow^* \langle p, O\rangle$*;*
2. *if* $\langle P, I\rangle \Rrightarrow^* \langle p, O\rangle$ *then* $\langle \mathbf{Tr}^{\mathbf{con}}(P), I\rangle \rightsquigarrow^* \langle p, O\rangle$*.*

For the proof, we refer to the extended version of this paper [4].

7 Related work

The work reported on in this paper is related to information flow security, a research area that has received significant attention for many decades. We point the reader to two broad surveys, and then zoom in to recent research that is closely related to our work. Sabelfeld and Myers [28] give an excellent survey on static techniques for information flow enforcement. Le Guernic's PhD thesis [16] surveys dynamic techniques.

Dynamic techniques for information flow security Several recent works propose run time monitors for information flow security, often with a particular focus on JavaScript, or on the Web context. These include monitoring algorithms that can handle DOM-like structures [27], dynamic code evaluation [1] and timeouts [26]. Austin and Flanagan [3] develop alternative, more permissive techniques. These run time monitoring based techniques are likely more efficient than the technique proposed in this paper, but they lack the precision of secure multi-execution: such monitors will block the execution of some non-interferent programs.

The idea underlying secure multi-execution was developed independently by several researchers. Capizzi *et al.* [8] propose *shadow executions*: they propose to run two executions of processes for the H (secret) and L (public) security level to provide strong confidentiality guarantees. Cristiá and Mata [12] independently formalize and prototype a similar system for secure multi-execution at operating system level. Devriese and Piessens [14] were the first to prove the strong soundness and precision guarantees that SME offers. They also report on a JavaScript implementation that requires a modified virtual machine. In a somewhat related line of work, Cavadini[9] proposes a technique based on program slicing to obtain secure fragments of insecure programs.

Several authors have improved on these initial results. Kashyap *et al.* [19], generalize the technique of secure multi-execution to a family of techniques that they call *the scheduling approach to non-interference*, and they analyze how the scheduling strategy can impact the security properties offered. Jaskelioff and Russo [18] propose a monadic library to realize secure multi-execution in Haskell. Bielova *et al.* [6] propose a variant of secure multi-execution suitable for reactive systems such as browsers. Finally, Austin and Flanagan [2] develop a more efficient implementation technique.

Finally, some other authors have considered program transformations for information flow security. Chudnov and Naumann [10] propose an inlined information flow monitor, and Birgisson *et al.* [7] propose a transformation towards a capability secure target language. Both approaches share the advantage of not requiring modifications to the operating system or virtual machine, but as with other classical run time monitors, they lack the precision of SME based approaches. In a sense, the approach proposed in this paper combines the advantages of these existing program-transformation based approaches with the advantages of SME (at the same performance cost as SME).

Other security techniques for JavaScript A motivating example for the technique proposed in this paper is providing security for JavaScript script inclusion. Many authors have proposed alternative security mechanisms. Chugh *et al.*[11] develop a novel multi-stage static technique for enforcing information flow security in JavaScript.

Most authors focus on *isolation* or *sandboxing* rather than information flow security: how can scripts be included in web pages without giving them full access to the surrounding page and the browser APIs. Several practical systems have been proposed, including ADSafe [13], Caja [23] and Facebook JavaScript [15]. Maffeis *et al.* [21] formalize the key mechanisms underlying these systems and prove they can be used to create secure sandboxes. They also discuss several other existing proposals; we point the reader to their paper for a more extensive discussion of work in this area.

The capability security approach is of particular relevance to this paper, as we build on the isolation provided by a capability secure language to develop our prototype implementation for JavaScript. Maffeis *et al.* [22] formalize capability safety, and prove a Caja-like subset of JavaScript capability safe. Taly *et al.* [30] propose an approach to verify if APIs offered to sandboxed code are secure.

Ter Louw *et al.* propose AdJail [20], targeted at sandboxing advertisements by isolating them in a separate iframe, and by providing a stub in the original web page that communicates in a controlled way with the sandboxed advertisement.

8 Conclusion

Secure multi-execution is an appealing approach to enforce information flow policies: it is sound and precise, and can be applied to a variety of programming languages. In this paper, we have shown that the effect of SME can be achieved through program transformation, and without the need to modify the underlying computing infrastructure.

Acknowledgments This research is partially funded by the Research Fund K.U.Leuven, by the IWT-SBO project SPION, and by the EU-funded FP7-projects HATS and WebSand. Dominique Devriese holds a Ph. D. fellowship of the Research Foundation - Flanders (FWO).

The authors are grateful to the anonymous reviewers for their useful and detailed comments on the paper.

References

1. Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF*, pages 43–59, 2009.
2. T. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, 2012.
3. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS*, 2010.
4. Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. Secure multi-execution through static program transformation: extended version. Technical Report CW620, Department of Computer Science, Katholieke Universiteit Leuven, 2012.
5. Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114, 2004.
6. N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *NSS*, 2011.
7. Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Capabilities for information flow. In *PLAS*, 2011.
8. R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In *ACSAC*, 2008.
9. Salvador Cavadini. Secure slices of insecure programs. In *ASIACCS*, pages 112–122, 2008.
10. Andrey Chudnov and David A. Naumann. Information flow monitor inlining. In *CSF*, pages 200–214, 2010.
11. R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for Javascript. In *PLDI*, 2009.
12. Maximiliano Cristiá and Pablo Mata. Runtime enforcement of noninterference by duplicating processes and their memories. In *WSEGI 2009*, 2009.

13. Douglas Crockford. Adsafe. `http://www.adsafe.org/`, December 2009.
14. Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, pages 109–124, 2010.
15. Facebook. Fbjs. `http://developers.facebook.com/docs/fbjs/`, 2011.
16. G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.
17. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, January 1998.
18. Mauro Jaskelioff and Alejandro Russo. Secure multi-execution in haskell. In *PSI*, 2011.
19. Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 413–428, Washington, DC, USA, 2011. IEEE Computer Society.
20. Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *USENIX Security Symposium*, pages 371–388, 2010.
21. Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating javascript with filters, rewriting, and wrappers. In *ESORICS*, pages 505–522, 2009.
22. Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *IEEE Symposium on Security and Privacy*, pages 125–140, 2010.
23. M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript. `http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf`, January 2008.
24. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.
25. G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *ECOOP*, 2011.
26. Alejandro Russo and Andrei Sabelfeld. Securing timeout instructions in web applications. In *CSF*, pages 92–106, 2009.
27. Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, pages 86–103, 2009.
28. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 21:5–19, 2003.
29. Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Ershov Memorial Conference*, pages 352–365, 2009.
30. Ankur Taly, Ulfar Erlingsson, Mark S. Miller, John C. Mitchell, and Jasvir Nagra. Automated analysis of security-critical javascript apis. In *IEEE Symposium on Security and Privacy*, 2011.
31. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2/3):167–188, 1996.