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, KULeuven, 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. to its 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. Practical deployment of SME is hampered by the fact that existing implementation techniques for SME execute programs in modified runtime environments, 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. 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, which 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 a program. For instance, such policies can limit how public output can depend on confidential input, or how high integrity output can be influenced by low integrity input. 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 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. *transparency:* the execution of secure programs should not be prevented or altered; iii. *practicality:*

the mechanism should not burden developers iv. *applicability:* smooth integration with existing programming languages and widely deployed runtime environments. However, 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 [29,15,22], and verification-based approaches [3]. These techniques are sound, and do not incur run-time efficiency penalty or require a modified runtime environment. However, type-based approaches are not transparent, and reject many secure programs. In contrast, verification-based approaches may offer perfect transparency (modulo completeness of the underlying program logic). However, neither type-based nor verification-based approaches are fully practical, as they require considerable involvement from program developers—who have to provide type annotations, or program annotations in a program logic. 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 [14,27,2,8], and more recently *secure multi-execution (SME)* [12,6]. Such techniques are sound, and can be more transparent than some static techniques. For instance, run-time monitors can be more transparent than type-based methods and reject less programs [27]; they are also more practical, in the sense that they require less annotation effort. However, run-time monitors are not precise, i.e. they may reject or alter the behavior of some secure programs. In contrast, secure multi-execution offers perfect transparency (at the cost of potentially modifying the behavior of insecure programs); it is also practical for developers, since there is no need for security annotations of the code. However, secure multi-execution is not easy to deploy, as all existing implementations of SME require modifications to the underlying computing infrastructure (OS [6], browser [4], virtual machine [12], trusted libraries [16]). Specifically, it is hard to deploy SME-based enforcement techniques for distributed and heterogeneous infrastructures, such as the web.

The key contribution of this paper is a new implementation technique that eliminates the need to modify the computing infrastructure for SME-based enforcement. We start from the essential insight of secure multi-execution, which is to guarantee non-interference by executing multiple copies of the program (one per security level) and by ensuring that the copy at level l only outputs to channels at level l, and that it only gets access to inputs from channels that are below or equal to l. Instead of relying on a purpose-specific semantics, we devise static program transformations that achieve a similar effect. Specifically, we introduce a program transformation that takes a sequential program P and builds a concurrent program $\mathbf{Tr}(P)$ containing multiple suitably synchronized copies of P—one per security level. Then, we sequentialize the resulting program to obtain a program $\mathbf{Tr}^{\mathbf{seq}}(P)$, written in the same language as P. We show that the SME semantics of P are equivalent to the standard semantics of $\mathbf{Tr}(P)$ and $\mathbf{Tr}^{\mathbf{seq}}(P)$, and conclude that the transformations are sound, transparent, and practical. The formal development is based on an exemplary imperative language with input/output and dynamic code evaluation; some of the central results have been

machine-checked using the Agda proof assistant. Moreover, we report on prototype implementations for a small fragment of Python and a realistic subset of JavaScript.

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 happen to be 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 are at liberty to use all of JavaScript's language features, including dynamic code evaluation, e.g. in the form of JavaScript's `eval` function, which Richards *et al.*[23] have shown to be widely used on the web for diverse purposes.

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://adprovider.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://adprovider.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 network output as 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://adprovider.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.

The ability of dynamically generating or loading new code and evaluating it on the fly 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.

Structure of the paper

The paper is organized as follows: Section 2 introduces our programming language and defines non-interference. Section 3 presents secure multi-execution and proves soundness and precision. Section 4 describes the program transformations and shows their equivalence with SME. We report on our prototype implementations in Section 5. Related work is addressed in Section 6 and conclusions are presented in Section 7.

Notation

Throughout the paper, we use some standard notation. Given a relation $\rightarrow$ we note its reflexive and transitive closure as $\rightarrow^*$ and its n-fold composition as $\rightarrow^n$. Given a mapping m, we let $m[x \mapsto v]$ be the mapping that assigns v to x and coincides with m on all other values. We use lists and some standard operations on them. In particular we let $[x]$ be the list containing a single element x and for two lists l_1, l_2 and a natural number n, $l_1 +\!\!+ l_2$ is the result of concatenating l_1 and l_2 and $l_1[n]$ denotes the value at the nth position of the list, or an undefined value in case the list happens to be shorter.

2 Setting

We base our formal development on an exemplary imperative language with input/output statements and dynamic code evaluation.

Syntax Following [12], a program P is simply a command to be executed by the system. The syntax of commands is given in Fig. 1. Most commands are standard, with the exception of $\mathbf{input}\ x\ \mathbf{from}\ ic$, that assigns the next input from the input channel ic to x, and $\mathbf{output}\ e\ \mathbf{to}\ oc$, that outputs the value of the expression e to the output channel oc—we assume that inputs and output channels are disjoint. The sole novelty w.r.t. [12] is the instruction $\mathbf{eval}(e)$, which takes an integer encoding e of a program (instead of the usual string encoding), decodes it and evaluates it.

Example 1. Figure 2 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 script which is executed with the same privilege.

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

Fig. 1: Syntax of programming language.

$$
\begin{aligned}
&\mathbf{input}\ keys\ \mathbf{from}\ LKeys; \\
&\mathbf{input}\ pass\ \mathbf{from}\ HPass; \\
&\mathbf{output}\ keys + pass\ \mathbf{to}\ LReq; \\
&\mathbf{input}\ res\ \mathbf{from}\ LReq'; \\
&\mathbf{eval}(res)
\end{aligned}
$$

Fig. 2: Introduction example modelled in the simple language.

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 are 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. 3 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:

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

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. We say that a program P produces input pointer p and program output O when executed for n steps, written $\langle P, I\rangle \rightsquigarrow^n \langle p, O\rangle$, iff $\langle P, m_0, p_0, I, O_0\rangle \rightsquigarrow^* \langle c, m, p, I, O\rangle$ for some memory m and command c. Note that the second notation does not require the execution of P to be terminated after n steps.

Security The notion of program security is defined relative to a lattice (which for the purpose of this paper we assume is totally ordered[3]) $(\mathcal{L}, \leq)$ of security levels, and mappings σ_{in} and σ_{out} from input and output channels to security levels. The mappings

[3] The results of this paper can be recovered without this assumption, for a weaker notion of non-interference.

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

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

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

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

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

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

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

Fig. 3: Operational semantics (excerpt).

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 $i \in \mathcal{C}_{in}$ 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 $o \in \mathcal{C}_{out}$ 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 $i \in \mathcal{C}_{in}$ 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 $O_f =_l O'_f$.*

We also consider a stronger notion of non-interference, which prevents that a difference in secret inputs may affect the running time of a program—for a simple, step-counting, model of running time.

Definition 2 (Strong Non-interference). *A program P is strongly 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 $0 \leq n$ and for all l-equal inputs I and I', i.e. $I =_l I'$, we have that $(P, I) \Rightarrow^n (p_f, O_f)$ if and only if $(P, I') \Rightarrow^n (p'_f, O'_f)$ and $p_f =_l p'_f$ and $O_f =_l O'_f$.*

The definition of (strong) non-interferent program is obtained by instantiating $\Rightarrow$ to $\rightsquigarrow$. None of these notions are satisfied by the Example 1.

Note that both of these definitions are *termination-sensitive*: it can be checked that they do not allow the program's termination to depend on information at non-minimal levels. Additionally, the latter is *timing-sensitive*: for l-equal inputs, the program is required to behave l-identically after any number of execution steps n (so that an attacker cannot learn additional information from observing timing differences between input or output requests).

3 Secure Multi-Execution: the operational approach

We extend the theoretical results of [12] and show that SME remains sound and precise in presence of dynamic code evaluation. We start with exemplifying SME for our running example, and by defining SME formally.

3.1 Example

The central insight of SME is that non-interference can be enforced by executing programs once per security level. Intuitively, SME can be realized by a multi-threaded program where each thread executes at a specific security level. In order to guarantee non-interference, the thread 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 thread corresponding to security level l' reads from them—the result is then available to be reused at security level l. The threads can be scheduled sequentially (lower security levels first) or in parallel.

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 4 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.

Execution at L security level.

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

Execution at H security level.

```
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://adprovider.com/SelectAd.php?keywords='+password
```

Fig. 4: Secure Multi-Execution of malicious Javascript program from Section 1.

3.2 Operational semantics of SME

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

The operational semantics combines a thread-local semantics, and a global semantics. The thread-local semantics corresponds to the current execution of the program for a security level. 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 $\mathbf{select}$ that, given a set of threads, picks the next thread to execute. In their work, Devriese and Piessens [12] focus on a scheduler $\mathbf{select}_{\text{lowprio}}$ which picks the local configuration corresponding to the lowest security level; other schedulers are considered in [17].

The thread-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 rules are given in Figure 5. We briefly comment on the rules for input and output commands for a local configuration with security level l:

- input and output commands at level l are executed with the expected semantics. Additionally, input commands from a channel ic emit a signal $\odot(ic, n)$ indicating that the n-th input from channel ic is available to threads that are waiting for it;
- output commands at level $l' \neq l$ are ignored;
- input commands at level $l' \not\leq l$ read a default value;
- input commands at level $l' < l$ are treated as follows: if the thread at level l' has already executed the corresponding input command, then the value is reused. Otherwise, the thread emits a signal $\otimes(ic, n)$ indicating that it is waiting for the input—the global semantics will put the configuration in the waiting queue.

Note that rule for sequence must propagate whatever signal has been emitted after the execution of the first instruction, if any. This is achieved by decorating execution steps with a tag $\dagger$, which may be $\otimes(ic, n)$, $\odot(ic, n)$ for some input channel ic and some integer n or nothing.

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 waiting queue that maps input channels and message numbers to local configurations.

The rules are given in Fig. 6. They capture formally the intuition behind SME; for instance, if a thread is suspended after attempting to read from input channel ic at position n, it is added to waiting queue $wq(ic, n)$. When the thread at security level $\sigma_{in}(ic)$ performs this read, the local step will produce a $\odot(i, n)$ signal and the waiting queue at that position will be awakened, i.e. added to the list of executing threads. The global input pointer r will, for each input channel ic, keep track of the first position n that the thread at level $\sigma_{in}(ic)$ has not yet read from.

$$\frac{c = \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \qquad [\![b]\!]m}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle c_1, m, p\rangle_l, r, I, O} \qquad \frac{c = \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \qquad \neg[\![b]\!]m}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle c_2, m, p\rangle_l, r, I, O}$$

$$\frac{\langle c_1, m, p\rangle_l, r, I, O \overset{\dagger}{\Rightarrow} \langle c_1', m', p'\rangle_l, r', I, O'}{\langle c_1; c_2, m, p\rangle_l, r, I, O \overset{\dagger}{\Rightarrow} \langle c_1'; c_2, m', p'\rangle_l, r', I, O'} \qquad \frac{}{\langle \textbf{skip}; c_2, m, p\rangle_l, r, I, O \Rightarrow \langle c_2, m, p\rangle_l, r, I, O}$$

$$\frac{c = \textbf{while } b \textbf{ do } c_{\text{loop}} \qquad [\![b]\!]m}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle c_{\text{loop}}; c, m, p\rangle_l, r, I, O} \qquad \frac{c = \textbf{while } b \textbf{ do } c_{\text{loop}} \qquad \neg[\![b]\!]m}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle \textbf{skip}, m, p\rangle_l, r, I, O}$$

$$\frac{c = \textbf{output } e \textbf{ to } oc \qquad \sigma_{out}(oc) = l}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle \textbf{skip}, m, p\rangle_l, r, I, \textbf{write}(O, oc, [\![e]\!]m)} \qquad \frac{c = \textbf{output } e \textbf{ to } oc \qquad \sigma_{out}(oc) \neq l}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle \textbf{skip}, m, p\rangle_l, r, I, O}$$

$$\frac{c = \textbf{input } x \textbf{ from } ic \qquad \sigma_{in}(ic) \not\leq l}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle \textbf{skip}, m[x \mapsto dv], p\rangle_l, r, I, O}$$

$$\frac{c = \textbf{input } x \textbf{ from } ic \qquad \sigma_{in}(ic) = l}{\langle c, m, p\rangle_l, r, I, O \overset{\odot(ic, p(ic))}{\Rightarrow} \langle \textbf{skip}, m[x \mapsto \textbf{read}(I, ic, p)], p[ic \mapsto p(ic) + 1]\rangle_l, r[ic \mapsto p(ic) + 1], I, O}$$

$$\frac{c = \textbf{input } x \textbf{ from } ic \qquad \sigma_{in}(ic) < l \qquad r(i) \leq p(i)}{\langle c, m, p\rangle_l, r, I, O \overset{\otimes(ic, p(ic))}{\Rightarrow} \langle c, m, p\rangle_l, r, I, O}$$

$$\frac{c = \textbf{input } x \textbf{ from } ic \qquad \sigma_{in}(ic) < l \qquad r(i) > p(i)}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle \textbf{skip}, m[x \mapsto \textbf{read}(I, ic, p)], p[ic \mapsto p(ic) + 1]\rangle_l, r, I, O}$$

$$\frac{}{\langle x := e, m, p\rangle_l, r, I, O \Rightarrow \langle \textbf{skip}, m[x \mapsto [\![e]\!]m], p\rangle_l, r, I, O}$$

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

Fig. 5: Thread local semantics for SME.

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 $\langle C, I\rangle \Rightarrow^* \langle r_f, O_f\rangle$, if

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

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 $\langle P, I\rangle \Rightarrow^* \langle r, O\rangle$ iff $\langle P_{lcinit}, I\rangle \Rightarrow^* \langle r, O\rangle$.

$$\frac{\mathbf{select}(L) = lec \qquad lec, r, I, O \Rrightarrow lec', r', I, O'}{\langle L, wq, r, I, O\rangle \Rrightarrow \langle L\backslash\{lec\} \cup \{lec'\}, wq, r', I, O'\rangle} \qquad \frac{\mathbf{select}(L) = lec = \langle \mathbf{skip}, m, p\rangle_l}{\langle L, wq, r, I, O\rangle \Rrightarrow \langle L\backslash\{lec\}, wq, r, I, O\rangle}$$

$$\frac{\mathbf{select}(L) = lec \qquad lec, r, I, O \overset{\odot(i,n)}{\Rrightarrow} lec', r', I, O'}{\langle L, wq, r, I, O\rangle \Rrightarrow \langle L\backslash\{lec\} \cup \{lec'\} \cup wq(i,n), wq[(i,n) \mapsto \{\}], r', I, O'\rangle}$$

$$\frac{\mathbf{select}(L) = lec \qquad lec, r, I, O \overset{\otimes(i,n)}{\Rrightarrow} lec', r', I, O'}{\langle L, wq, r, I, O\rangle \Rrightarrow \langle L\backslash\{lec\}, wq[(i,n) \mapsto wq(i,n) \cup \{lec_i\}], r', I, O'\rangle}$$

Fig. 6: Global Semantics for SME.

3.3 Soundness and precision

Secure multi-execution provides strong security and operational guarantees. The first essential property of SME is soundness: under the $\mathbf{select}_{\text{lowprio}}$ scheduler, the SME execution of programs is non-interferent.

Theorem 1 (Soundness of SME). *Any program P is non-interferent under SME, using the $\mathbf{select}_{lowprio}$ scheduler. If moreover, the set of security levels is totally ordered, then any program P is strongly non-interferent under SME, using the $\mathbf{select}_{lowprio}$ scheduler.*

The second essential property of SME is precision: if P is non-interferent (w.r.t. the standard operational semantics), then the semantics of P under SME coincides with the standard operational semantics.

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 $\langle P, I\rangle \Rrightarrow^* \langle p_f, O_f\rangle$.*

The proofs follow along the lines of [12]; 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. Specifically, we consider two transformations; the first one yields a concurrent program $\mathbf{Tr}(P)$, that can be executed under different choices of schedulers. The second transformation yields a sequential program $\mathbf{Tr}^{\mathbf{seq}}(P)$; intuitively, $\mathbf{Tr}^{\mathbf{seq}}(P)$ corresponds to executing $\mathbf{Tr}(P)$ under the $\mathbf{select}_{\text{lowprio}}$ scheduler. Our results show that one can achieve soundness and precision without modifying the runtime environment.

4.1 Target language

We extend our command language with a synchronization primitive $\mathbf{await}\ b\ \mathbf{then}\ c$; intuitively, the command $\mathbf{await}\ b\ \mathbf{then}\ c$ executes c atomically, provided b holds, and

is stuck otherwise. Then, a program is simply a set of threads; for convenience, we assume that each thread is tagged with a unique identifier. The syntax of the extended language is given in Fig. 7. In what follows, we write $\mathbf{atomic}\ c$ as a shorthand for $\mathbf{await\ true\ then}\ c$.

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

Fig. 7: Syntax of concurrent programming language.

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 8 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

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

4.2 The transformation

Despite its simplicity, our concurrent language is sufficiently expressive to serve as a target for the program transformation $\mathbf{Tr}$. Informally, one defines for each program P and security level l a transformed program $\mathbf{Tr}(P, l)$ and define $\mathbf{Tr}(P)$ as the parallel composition of the commands $\mathbf{Tr}(P, l)$, where l ranges over security levels. The main idea of the transformation is that the synchronization which SME implements within the runtime environment can be mimicked through the use of the synchronization primitives provided by the concurrent language. The buffers used in SME execution to share inputs of a thread to its copies running at higher security levels are implemented as global lists and the global input pointer as well as local input pointers are represented as global integer variables.

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

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

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

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

(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. 8: Extended semantics.

x for the thread 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. 9. 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

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

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—and we use the synchronization primitives to achieve the effect 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 is an $\mathbf{await}$ command that waits for the input value to become available, and upon satisfaction of the guard, reads the input value and updates its counter. Again, the transformed command ensures that the operations are performed atomically.

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

Fig. 9: Syntactic program transformation.

The program $\mathbf{Tr}(P)$ to be executed is the parallel composition of the transformation applied w.r.t. each security level for the original program, i.e. $\parallel \{(l, \mathbf{Tr}(P, l)) \mid l \in \mathcal{L}\}$ and is to be executed in an initial memory in which every $count$ variable has value 0 and every $list$ variable is associated with the empty list.

Example 2. 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. 10a and Fig. 10b respectively. These two programs will be executed concurrently and synchronize on input events. Note that the undesired flow of information is circumvented by replacing high inputs by default values in the low copy and by omitting low

outputs in the high copy. Also, the transformation is propagated across eval statements as described previously.

```
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. 10: Static transformation applied to malicious ad.

Secure multi-execution through static transformation yields executions equivalent to secure multi-execution. This proof relies on a simulation result and hinges on the assumption that (informally) schedulers pick the same threads to execute.

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

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

Corollary 1. *Statically enforced secure multi-execution is sound (w.r.t. $\mathbf{select}_{lowprio}$ scheduler) and precise.*

Proof. Soundness follows from Theorem 3, first part and soundness of SME (Theorem 1). Precision follows from Theorem 3, second part and precision of SME (Theorem 2).

4.3 Sequentializing static SME

Although the results in the previous section are interesting, they only apply to languages that provide support for concurrency. In this section we show how to carry these results to a non-concurrent setting by statically picking a way to schedule the threads. In particular, we focus on the $\mathbf{select}_{lowprio}$ scheduler.

The changes required for the static transformation to work on the sequential case are rather simple and local to the way input operations are handled:

$$\mathbf{Tr}^{\mathbf{seq}}(\mathbf{input}\ x\ \mathbf{from}\ ic, l) = \begin{cases} x_l := dv & \text{if } (ic) > l \\ \mathbf{atomic}\ \{\mathbf{input}\ x\ \mathbf{from}\ ic; \\ list_{ic} := list_{ic} +\!\!+ [x_l]; \\ count_{ic} := count_{ic} + 1\} & \text{if } (ic) = l \\ \mathbf{atomic}\ \{x_l := list_{ic}[count_{ic,l}]; \\ count_{l,ic} := count_{l,ic} + 1\} & \text{otherwise} \end{cases}$$

Note that the transformation is performed keeping the atomicity statements. These are of course superfluous in a non-concurrent setting, but we keep them as an artifact to ease the proofs: we need to prove equivalence of executions in the same number of steps.[4] However, we do omit **atomic** statements in examples.

We define the sequential transformation of an entire program as the sequencing of the translations at each level, respecting the order of the security level.

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

Example 3. Again, we apply the transformation to our running example. The sequential program obtained is shown in Fig. 11.

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

Fig. 11: Sequential static transformation applied to malicious ad.

Informally, this way of sequentializing SME mirrors the way in which threads are scheduled using the $\mathbf{select}_{lowprio}$ scheduler. This is given a formal status through the following theorems.

[4] An alternative approach would dispense of the atomicity statements and prove that the number of steps in the sequential execution is a function of the number of the different kinds of steps in the other execution, but this would make some of the proofs more difficult.

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

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

Corollary 2. *For every program P and program input I:*

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

Proof. Part one, follows from Theorem 3, part one and Theorem 4, part one. Part two, follows from Theorem 3, part two and Theorem 4, part two.

Corollary 3. *Statically enforced sequential SME is sound and precise.*

Proof. Soundness follows from the first part of Corollary 2, first part and soundness of SME (Theorem 1). Precision follows from Corollary 2, second part and precision of SME (Theorem 2).

We have developed a mechanized proof covering an important part of the results in this text using Agda, a proof assistant based on the Curry-Howard isomorphism. Because of both technical reasons and time constraints, our mechanized proof is structured differently than the proofs in the appendix, and focuses on the sequentialized transformation. The proof covers both parts of Corollary 2. The proof scripts are available online[5].

5 Implementation

In order to validate our approach, we have developed two prototype implementations of the sequential transformation described in Section 4.3. 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. The implementation can be tested online[6]. In this section, 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 presence of 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,

[5] `http://people.cs.kuleuven.be/~dominique.devriese/permanent/sme-transfo-mechanized.tar.gz`

[6] `http://dcc.fceia.unr.edu.ar/~erivas/sme/demo.html`

using Google Caja [21] which guarantees that the translated program only get 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.

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. It has the technical limitation that the executed code is always executed in a new, empty scope (but the eval'd code can still communicate with the outside world via arguments passed in and through its result value).

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 XMLHttpRequests. 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.* [4] discuss how events and callbacks can be supported under secure multi-execution. Extending our transformation to address events and callbacks, and support for the full DOM is a significant engineering challenge, which we regard as future work.

6 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 [26] give an excellent survey on static techniques for information flow enforcement. Le Guernic's PhD thesis [14] 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. Sabelfeld *et al.* have proposed monitoring algorithms that can handle DOM-like structures [25], dynamic code evaluation [1] and timeouts [24]. Austin and Flanagan [2] have developed 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.* [6] proposed *shadow executions*: they propose to run two executions of operating system processes for the H (secret) and L (public) security level to provide strong confidentiality guarantees. Cristiá and Mata [10] independently formalize and prototype a similar system for secure multi-execution at operating system level. Devriese and Piessens [12] 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[7] 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.* [17], 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 [16] propose a monadic library to realize secure multi-execution in Haskell. Bielova *et al*. [4] propose a variant of secure multi-execution suitable for reactive systems such as browsers.

Finally, some other authors have considered program transformations for information flow security. Chudnov and Naumann [8] propose an inlined information flow monitor, and Birgisson *et al.* [5] propose a transformation towards a capability secure target language. Both approaches share the advantage of not requiring modifications to operating system or virtual machine, but as 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 An important motivating example for the technique proposed in this paper is providing security for JavaScript script inclusion. Many authors have proposed a variety of alternative security mechanisms. Chugh *et al.*[9] have developed 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 API's. Several practical systems have been proposed, including ADSafe [11], Caja [21] and Facebook JavaScript [13]. Maffeis *et al.* [19] formalize the key mechanisms underlying these sandboxes and prove they can be used to create secure sandboxes. They also discuss several other existing proposals, and 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.* [20] have formalized capability safety, and have proved a Caja-like subset of JavaScript capability safe. Taly *et al.* [28] propose an approach to verify if API's offered to sandboxed code are secure.

Ter Louw *et al.* have proposed AdJail [18], specifically 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 code.

7 Conclusion

Secure Multi Execution is an appealing approach to enforce information flow policies: it is sound and precise, an can be applied to a variety of programming languages. In this paper, we have shown that the effect of SME can be achieved through static program transformation, and without the need to modify the underlying computing infrastructure. Our work opens the way for practical deployment of SME for Web applications. Our next objective is to extend our implementation to handle the full DOM.

8 Acknowledgments

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by the Research Fund K.U.Leuven, and by the EU-funded FP7-projects HATS and WebSand. Dominique Devriese holds a Ph. D. fellowship of the Research Foundation - Flanders (FWO). Juan Manuel Crespo holds an FPI Ph. D. fellowship funded by the Ministry of Science and Innovation of the Spanish Government.

References

1. Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF*, pages 43–59, 2009.
2. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS*, 2010.
3. Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114, 2004.
4. N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *NSS*, 2011.
5. Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Capabilities for information flow. In *PLAS*, 2011.
6. R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In *ACSAC*, 2008.
7. Salvador Cavadini. Secure slices of insecure programs. In *ASIACCS*, pages 112–122, 2008.
8. Andrey Chudnov and David A. Naumann. Information flow monitor inlining. In *CSF*, pages 200–214, 2010.
9. R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for Javascript. In *PLDI*, 2009.
10. Maximiliano Cristiá and Pablo Mata. Runtime enforcement of noninterference by duplicating processes and their memories. In *Workshop de Seguridad Informática WSEGI 2009*, 2009.
11. Douglas Crockford. Adsafe. `http://www.adsafe.org/`, December 2009.
12. Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, pages 109–124, 2010.
13. Facebook. Fbjs. `http://developers.facebook.com/docs/fbjs/`, 2011.
14. G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

15. 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.
16. Mauro Jaskelioff and Alejandro Russo. Secure multi-execution in haskell. In *PSI*, 2011.
17. 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.
18. 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.
19. Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating javascript with filters, rewriting, and wrappers. In *ESORICS*, pages 505–522, 2009.
20. 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.
21. 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.
22. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.
23. G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *ECOOP*, 2011.
24. Alejandro Russo and Andrei Sabelfeld. Securing timeout instructions in web applications. In *CSF*, pages 92–106, 2009.
25. Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, pages 86–103, 2009.
26. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 21:5–19, 2003.
27. 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.
28. 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.
29. 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.

A Proofs

In this section, we develop the necessary notions to provide the proofs of the theorems of the previous section. The proofs follow by defining relations between the execution states and then:

- proving these relations hold initially;
- proving these relations are preserved throughout executions, i.e. the relations are invariants of the executions;
- proving that the relations entail state equality.

For clarity, we provide an abstract formal inductive definition of $\Rightarrow^n$ in terms of an abstract relation $\Rightarrow$ between configurations γ in Fig. 12.

$$\frac{}{\gamma \Rightarrow^0 \gamma} \qquad \frac{\gamma_1 \Rightarrow^n \gamma_2 \qquad \gamma_2 \Rightarrow \gamma_3}{\gamma_1 \Rightarrow^{n+1} \gamma_3}$$

Fig. 12: Formal definition of n-step execution.

We instantiate this definition for execution relations $\rightsquigarrow$ and $\Rightarrow$ and their corresponding configurations, and we use its associated induction principle in our proofs.

We have mechanized the proof of equivalence between sequentialized static SME and SME in Agda. The proof script is available online[7]. In this paper, and for the purpose of clarity of exposition we provide a proof of equivalence of concurrent static SME and traditional SME, followed by a proof of sequentialized static SME and concurrent static SME. The mechanized result is then obtained as a consequence of these two theorems.

A.1 Proofs of Subsection 4.1

We start by providing some auxiliary definitions and the relation witnessing the simulation between concurrent static SME and traditional SME.

Definition 3. *Let m and m' be two memories and l a security level, we say that m is l-similar to m' iff:*

$$m \sim_l m' \stackrel{def}{=} \forall x, x \in \mathbf{dom}(m') \Rightarrow m'(x) = m(x_l)$$

Definition 4. *Let m be a memory, let p be an input pointer and l a security level, we say that m l-simulates p iff:*

$$m \sim_l p \stackrel{def}{=} \qquad \forall i, i \in \mathcal{C}_{in} \Rightarrow p(i) = [\![count_{i,l}]\!]m$$

[7] `http://people.cs.kuleuven.be/~dominique.devriese/permanent/sme-transfo-mechanized.tar.gz`

Definition 5. *Let m be a memory, let r be an input pointer and let I be a program input, we say that m r-simulates I iff*

$$m \sim_r I \stackrel{def}{=} \qquad \forall i, i \in \mathcal{C}_{in} \Rightarrow \forall k, k < r(i) \Rightarrow I(i,k) = [\![list_i[k]]\!]m$$

Definition 6. *Let P be a ready list for configurations of traditional execution, L be a ready list for configuration of SME, and m be a memory, we will say that P m-simulates L, noted $P \sim_m L$ iff:*

$$\begin{aligned} P \sim_m L \stackrel{def}{=} \forall l, c, (l,c) \in P \Rightarrow \exists m', p', c', \\ \langle c', m', p' \rangle_l \in L \wedge c = \mathbf{Tr}_{\sigma_{in}, \sigma out}(c', l) \wedge m \sim_l m' \wedge m \sim_l p \wedge \\ |P| = |L| \end{aligned}$$

Note that a key requirement for this proof is the fact that the scheduler of SME and the scheduler of standard executions will "make the same choices". Formally, for any ready list of SME L and any ready list of NE P such that $P \sim_m L$ if $\mathbf{select}(P) = (l, c)$ and $\mathbf{select}(L) = \langle c', m', p' \rangle'_l$ then $l' = l$.

We call this the "equal scheduler election hypothesis".

Definition 7. *Let wq be a waiting list for configurations of traditional execution, wq' be a waiting list for configuration of SME, and m be a memory, we will say that wq m-simulates wq', noted $wq \sim_m wq'$ iff:*

$$\begin{aligned} wq \sim_m wq' \stackrel{def}{=} \forall b, l, c, (b, (l, c)) \in wq \Rightarrow \exists i, n, m', p', c', \\ ((i,n) \mapsto \langle c', m', p' \rangle_l) \in wq' \wedge c = \mathbf{Tr}_{\sigma_{in}, \sigma out}(c', l) \wedge \\ m \sim_l m' \wedge m(count_{i,l}) = n \wedge m \sim_l p' \wedge \\ b = (count_i > count_{i,l}) \wedge |wq| = |\bigcup_{(i,n) \in \mathbf{dom}(wq')} wq'(i,n)| \end{aligned}$$

Definition 8. *Let $\langle P, wq, m, p, I, O \rangle$ and $\langle L, wq', r, I, O' \rangle$ be a configurations for normal execution and SME execution respectively, then we define: R as follows:*

$$\begin{aligned} R\langle P, wq, m, p, I, O \rangle \langle L, wq', r, I', O' \rangle \stackrel{def}{=} O = O' \wedge p = r \wedge I = I' \wedge \\ P \sim_l L \wedge wq \sim_l wq' \wedge m \sim_r I' \end{aligned}$$

Lemma 1. *R satisfies the following properties:*

- *Initial execution states are related by R, i.e.:*

$$R\langle \| \{(l, \mathbf{Tr}_{\sigma_{in}, \sigma_{out}}(P, l)) | l \in \mathcal{L}\}, \{\}, m_{0, \sigma_{in}, \sigma_{out}}, p_0, I, O_0 \rangle \langle L_{P,0}, wq_0, r_0, I, O \rangle$$

- *If two configurations are related by R, one is final iff the other one is final, i.e.:*

$$\begin{aligned} R\langle P, wq, m, p, I, O \rangle \langle L, wq', r, I', O' \rangle \Rightarrow \\ (P = \{\} \wedge wq = \{\} \Leftrightarrow L = \{\} \wedge \forall i, n, wq'(i,n) = \{\}) \end{aligned}$$

Lemma 2. *For any configuration $\langle c, m, p, I, O\rangle$ of the standard execution model such that $\langle c, m, p, I, O\rangle \rightsquigarrow \langle c', m', p', I, O'\rangle$ and any security level l, memory m_2, command c_2, input pointer p_2 and r_2 and program output O such that $\mathbf{Tr}(c_2, l) = c$, $m_2 \sim_l m$, $m \sim_l p_2$, $r_2 = p$ and $O_2 = O$ then there exist $c_2', m_2', p_2', r_2', O_2'$ such that:*

$$\langle c_2, m_2, p_2\rangle_l, r_2, I, O_2 \overset{\dagger}{\Rightarrow} \langle c_2', m_2', p_2'\rangle_l, r_2', I, O_2'$$

and $\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c_2', l) = c'$, $m_2' \sim_l m'$, $m' \sim_l p_2'$, $r_2' = p'$ and $O_2' = O'$.

Proof. The proof follows by induction on the structure of command c_2. We treat the most interesting case, namely, inputs.

Case $c_2 = \mathbf{input}\ x\ \mathbf{from}\ ic$ This case has three subcases according to $\sigma_{in}(ic)$.

- $l < l'$. In this case $\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c_2, l) = x_l := dv$. By inversion in standard one step execution, we know that $c' = \mathbf{skip}$, $m' = m[x_l \mapsto dv]$, $p' = p$ and $O' = O$. We then set $c_2' = \mathbf{skip}$, $m_2' = m_2[x \mapsto dv]$, $p_2' = p_2$, $r_2' = r_2$ and $O_2' = O_2$. $\langle c_2, m_2, p_2\rangle_l, r_2, I, O_2 \overset{\dagger}{\Rightarrow} \langle c_2', m_2', p_2'\rangle_l, r_2', I, O_2'$ follows from thread-local semantics of SME (rule 9) and checking the relation between final states is straightforward.
- $l = l'$. In this case

$$\begin{aligned}\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c_2, l) = \mathbf{await}\ (true)\ \mathbf{then}\{&\\ &\mathbf{input}\ x_l\ \mathbf{from}\ ic;\\ &list_{ic} := list_{ic} +\!\!+ [x_l];\\ &count_{ic} := count_{ic} + 1\}\end{aligned}$$

  By inversion in standard one step execution, we know that $c' = \mathbf{skip}$, $m' = m[x_l \mapsto \mathbf{read}(I, ic, p)][list_{ic} \mapsto (m\ list_{ic}) +\!\!+ [x_l]][count_{ic} \mapsto (m\ count_{ic}) + 1]$, $p' = p[ic \mapsto (p\ ic) + 1]$ and $O' = O$. We then set $c_2' = \mathbf{skip}$, $m_2' = m_2[x \mapsto \mathbf{read}(I, ic, p_2')]$, $p_2' = p_2[ic \mapsto (p_2\ ic) + 1]$, $r_2' = r_2[ic \mapsto (r_2\ ic) + 1]$ and $O_2' = O_2$. $\langle c_2, m_2, p_2\rangle_l, r_2, I, O_2 \overset{\odot(ic, p_2(ic))}{\Rightarrow} \langle c_2', m_2', p_2'\rangle_l, r_2', I, O_2'$ follows from thread-local semantics of SME (rule 10). Checking the relation for final states is routine.
- $l' < l$. In this case

$$\begin{aligned}\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c_2, l) = \mathbf{await}\ (count_{ic,l} < count_{ic})\ \mathbf{then}\{&\\ &x_l := list_{ic}[count_{ic,l}];\\ &count_{ic,l} := count_{ic,l} + 1\}\end{aligned}$$

  Applying inversion on the standard one step execution, there are two subcases.

  - $\langle c, m, p, I, O\rangle \overset{count_{ic,l} < count_{ic}}{\rightsquigarrow} \langle c, m, p, I, O\rangle$ and $count_{ic} \leq count_{ic,l}$. We then set $c_2' = c_2$, $m_2' = m_2$, $r_2' = r_2$ and $O_2' = O_2$. From thread-local semantics of SME (rule 11) we know that

$$\langle c_2, m_2, p_2\rangle_l, r_2, I, O_2 \overset{\otimes(ic, p_2(ic))}{\Rightarrow} \langle c_2', m_2', p_2'\rangle_l, r_2', I, O_2'$$

    It remains to show that $r(i) \leq p(i)$, which follows from hypothesis on initial configuration. Relation on final configurations is established trivially.

- We know that $c' = \mathbf{skip}$, $m' = m[x_l \mapsto [\![list_{ic}[count_c]]\!]m][count_{ic,l} \mapsto (m\ count_{ic,l}) + 1]$, $p' = p$ and $O' = O$. We then set $c'_2 = \mathbf{skip}$, $m'_2 = m_2[x \mapsto \mathbf{read}(I, ic, p'_2)]$, $p'_2 = p_2[ic \mapsto (p_2\ ic)+1]$, $r'_2 = r_2[ic \mapsto (r_2\ ic)+1]$ and $O'_2 = O_2$. The fact that $\langle c_2, m_2, p_2\rangle_l, r_2, I, O_2 \Rrightarrow \langle c'_2, m'_2, p'_2\rangle_l, r'_2, I, O'_2$ follows from thread-local semantics of SME (rule 12). Checking the relation for final states is routine.

Lemma 3. *Let $\langle P, wq, m, p, I, O\rangle$ and $\langle L, wq', r, I', O'\rangle$ be two configurations such that $R\langle P, wq, m, p, I, O\rangle\langle L, wq', r, I', O'\rangle$. For every configuration $\langle P_1, wq_1, m_1, p_1, I\rangle$ such that $\langle P, wq, m, p, I, O\rangle \rightsquigarrow^n \langle P_1, wq_1, m_1, p_1, I, O_1\rangle$ then there exists L_2, wq'_2, r_2, I', O'_2 such that $\langle L, wq', r, I', O'\rangle \Rrightarrow^n \langle L_2, wq'_2, r_2, I', O'_2\rangle$ and*

$$R\langle P_1, wq_1, m_1, p_1, I, O_1\rangle\langle L_2, wq'_2, r_2, I', O'_2\rangle$$

Proof. The proof follows by induction on the derivation of $\langle P, wq, m, p, I, O\rangle \rightsquigarrow^n \langle P_1, wq_1, m_1, p_1, I, O_1\rangle$.

The first case, in which there is no execution step, is straightforward. For the second case we know the following:

$$\langle P, wq, m, p, I, O\rangle \rightsquigarrow^n \langle P_1, wq_1, m_1, p_1, I, O_1\rangle$$
$$\langle P_1, wq_1, m_1, p_1, I, O_1\rangle \rightsquigarrow \langle P_3, wq_3, m_3, p_3, I, O_3\rangle$$

Also, by induction hypothesis we know there is a configuration $\langle L_2, wq'_2, r_2, I', O'_2\rangle$ such that:

$$\langle L, wq', r, I', O'\rangle \Rrightarrow^n \langle L_2, wq'_2, r_2, I', O'_2\rangle$$
$$R\langle P_1, wq_1, m_1, p_1, I, O_1\rangle\langle L_2, wq'_2, r_2, I', O'_2\rangle \ (1)$$

It then suffices to show that there is a configuration $\langle L_4, wq'_4, r_4, I', O'_4\rangle$ such that:

$$\langle L_2, wq'_2, r_2, I', O'_2\rangle \Rrightarrow \langle L_4, wq'_4, r_4, I', O'_4\rangle$$
$$R\langle P_3, wq_3, m_3, p_3, I, O_3\rangle\langle L_4, wq'_4, r_4, I', O'_4\rangle$$

We proceed by case analysis on the derivation of

$$\langle P_1, wq_1, m_1, p_1, I, O_1\rangle \rightsquigarrow \langle P_3, wq_3, m_3, p_3, I, O_3\rangle$$

.

Case (1). We know that $\mathbf{select}(P_1) = (l, \mathbf{skip})$, $P_3 = P_1 \backslash \{(l, \mathbf{skip})\}$, $wq_3 = wq_1$, $m_3 = m_1$, $p_3 = p_1$ and $O_3 = O_1$. From the definition of R and (1) we know that there is $\langle c_4, m_4, p_4\rangle_l$ in L_4 such that $\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c_4, l) = \mathbf{skip}$. Moreover, by the equal scheduler choice hypothesis we know $\mathbf{select}(L_2) = \langle c_4, m_4, p_4\rangle_l$. Using the former, we know $c_4 = \mathbf{skip}$. From this and the scheduler choice we conclude using the second rule of global SME semantics

$$\langle L_2, wq'_2, r_2, I', O'_2\rangle \Rrightarrow \langle L_2 \backslash \{\langle c_4, m_4, p_4\rangle_l\}, wq'_2, r_2, I', O'_2, O'_4\rangle$$

It remains to show that:

$$R\langle P_1 \backslash \{(l, \mathbf{skip})\}, wq_1, m_1, p_1, I, O_1\rangle\langle L_2 \backslash \{\langle c_4, m_4, p_4\rangle_l\}, wq_2', r_2, I', O_2', O_4'\rangle$$

which follows from the definition of R and (1).

Case (2). We know that $\neg[\![count_{i,l} \leq count_i]\!]m_1$, $P_3 = P_1 \backslash \{(l, c)\}$, $wq_3 = wq_1 \cup \{(l, c)\}$, $m_3 = m_1$, $p_3 = p_1$, $O_3 = O_1$ and

$$\mathbf{select}(P_1) = (l, \mathbf{await}\ count_{i,l} \leq count_i\ \mathbf{then}\ x_l := list_i[count_{i,l}]; x_l := x_l{+}1; c_{seq})$$

Using (1) and from the definition of R we know that there are c_4, m_4, p_4 such that:

$$\langle c, m_4, p_4\rangle_l \in L_2 \wedge c = \mathbf{input}\ x\ \mathbf{from}\ i; c'_{seq} \wedge \mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c'_{seq}, l) = c_{seq}$$

Also, note that $p_4(i) > r_2(i)$ and $\sigma_{in}(i) < l$. Using the rule

By the equal scheduler election hypothesis we know that:

$$\mathbf{select}(L_2) = \langle \mathbf{input}\ x\ \mathbf{from}\ i; c'_{seq}, m_4, p_4\rangle_l$$

It then follows from thread local semantics of SME that:

$$\frac{\langle \mathbf{input}\ x\ \mathbf{from}\ i; c'_{seq}, m_4, p_4\rangle_l, r_2, I', O_2\rangle_l \overset{\otimes(i,[\![count_i]\!]m_1)}{\Rrightarrow}}{\langle \mathbf{input}\ x\ \mathbf{from}\ i; c'_{seq}, m_4, p_4\rangle_l, r_2, I', O_2\rangle_l}$$

Applying global semantics of SME, last rule, we get:

$$\langle L_2, wq_2', r_2, I', O_2'\rangle \Rrightarrow \langle L_4, wq_4', r_4, I', O_4'\rangle$$

with $r_4 = r_2$, $O_4' = O_2$ and

$$\begin{array}{l} L_4 = L_2 \backslash \{\langle \mathbf{input}\ x\ \mathbf{from}\ i, m_4, p_4\rangle_l, m_2, I', O_2\rangle_l\} \\ wq_4' = wq_2'[(i, [\![count_i]\!]) \mapsto \langle \mathbf{input}\ x\ \mathbf{from}\ i, m_4, p_4\rangle_l, m_2, I', O_2\rangle] \end{array}$$

Checking that $R\langle P_3, wq_3, m_3, p_3, I, O_3\rangle\langle L_4, wq_4', r_4, I', O_4'\rangle$ is straightforward.

Case (3). We know that:

$$\begin{array}{l} \mathbf{select}(P_1) = (l, c) \\ \langle c, m_1, p_1, I, O_1\rangle \rightsquigarrow \langle c_3, m_3, p_3, I, O_3\rangle \end{array}$$

From R, we know that $\langle c_2, m_2, p_2\rangle_l \in L$ with $m_1 \sim_l m_2$, $m_1 \sim_l p_2$ and $\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c_2, l) = c_3$, and using the equal scheduler hypothesis we know $\mathbf{select}(L) = \langle c_2, m_2, p_2\rangle_l$.

Using Lemma 2 we obtain that there exists c_4, m_4, p_4, r_4, O_4 such that:

$$\langle c_2, m_2, p_2\rangle_l, r_2, I, O \overset{\dagger}{\Rrightarrow} \langle c_4, m_4, p_4\rangle_l, r_4, I, O_4$$

such that $m_4 \sim_l m_3$, $m_3 \sim_l p_4$, $r_4 = p_3$ and $O_4 = O_3$.

We proceed by case analysis on †.

– If $\dagger = \bullet$ then there has been no signal, i.e. no reads from a channel. Since command c is a translation of a command that includes no inputs, we are sure that none of the guards of the blocked threads become valid, so $wq_3 = wq_1$. On the other hand, $P_3 = P_1 \backslash \{(l, c)\} \cup \{l, c_3\}$.
Using rule 1 of SME global semantics, we conclude:

$$\langle L_2, wq_2, I, O_2\rangle \Rightarrow \triangleright \langle L_2 \backslash \{\langle c_2, m_2, p_2\rangle_l\} \cup \{\langle c_4, m_4, p_4\rangle_l\}, wq_4, I, O_4\rangle$$

Checking that the relation holds in this case is straightforward.

– If $\dagger = \odot(i, n)$ there has been a signal. This means that either $c_2 = \mathbf{input}\ x\ \mathbf{from}\ i; c_2'$ or $c_2 = \mathbf{input}\ x\ \mathbf{from}\ i$ with $l = \sigma_{in}(i)$. We prove it for the former case, in which $\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(\mathbf{input}\ x\ \mathbf{from}\ i; c_2', l) = \mathbf{await}\ true\ \mathbf{then}\ (\mathbf{input}\ x\ \mathbf{from}\ i; count_i := count_i + 1; list_i := list_i +\!\!+ [x_l]); c'$ with $\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c_2', l) = c'$.
Using SME global semantics, rule (3), we obtain:

$$\langle L_2, wq_2, I, O_2\rangle \Rightarrow \triangleright$$
$$\langle L_2 \backslash \{\langle c_2, m_2, p_2\rangle_l\} \cup \{\langle c_4, m_4, p_4\rangle_l\} \cup wq_2(i, n), wq_2[(i, n) \mapsto \{\}], I, O_4\rangle$$

It remains to check:

$$P_3 \sim_{m_3} L_2 \backslash \{\langle c_2, m_2, p_2\rangle_l\} \cup \{\langle c_4, m_4, p_4\rangle_l\} \cup wq_2(i, n)$$
$$wq_3 \sim_{m_3} wq_2[(i, n) \mapsto \{\}]$$

This follows by analysing the derivation of $\langle c, m_1, p_1, I, O_1\rangle \rightsquigarrow \langle c_3, m_3, p_3, I, O_3\rangle$, taking into account c's definition.

Proof of Theorem 3, part one. For every program P, and program input I if

$$\langle \| \{(l, \mathbf{Tr}_{\sigma_{in},\sigma_{out}}(P, l)) | l \in \mathcal{L}\}, I\rangle \rightsquigarrow^n \langle p, O\rangle$$

then

$$\langle P, I\rangle \Rightarrow \triangleright^n \langle p, O\rangle$$

Proof. By definition, we know that there is an m such that:

$$\langle \| \{(l, \mathbf{Tr}_{\sigma_{in},\sigma_{out}}(P, l)) | l \in \mathcal{L}\}, \{\}, m_{0,\sigma_{in},\sigma out}, p_o, I, O_0\rangle \rightsquigarrow^n \langle \{\}, \{\}, m, p, I, O\rangle$$

Using Lemma 1, first part, we know:

$$R\langle \| \{(l, \mathbf{Tr}_{\sigma_{in},\sigma_{out}}(P, l)) | l \in \mathcal{L}\}, \{\}, m_{0,\sigma_{in},\sigma_{out}}, p_0, I, O_0\rangle \langle L_{P,0}, wq_0, r_0, I, O\rangle$$

Using Lemma 3 we know that there exists L_2, wq_2', r_2, O_2' such that:

$$\langle L_{P,0}, wq_0, r_0, I, O\rangle \Rightarrow \triangleright^n \langle L_2, wq_2', r_2, I, O_2'\rangle$$
$$R\langle \{\}, \{\}, m, p, I, O\rangle \langle L_2, wq_2', r_2, I, O_2'\rangle$$

and we know it is a final configuration, using the last part of Lemma 1 and the fact that $\langle \{\}, \{\}, m, p, I, O\rangle$ is a final configuration. Hence, we conclude:

$$\langle P, I\rangle \Rightarrow \triangleright^n \langle r_2, O_2'\rangle$$

From the definition of R, we know that $r_2 = p$ and $O_2' = O$.

Now we will prove the converse result for non-interferent programs.

Lemma 4. *For any configuration* $\langle c_2, m_2, p_2\rangle_l, r_2, I, O_2$ *of SME such that*

$$\langle c_2, m_2, p_2\rangle_l, r_2, I, O_2 \overset{\dagger}{\Rrightarrow} \langle c_2', m_2', p_2'\rangle_l, r_2', I, O_2'$$

, and any memory m*, command* c*, input pointer* p *and program output* O *such that* $\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c_2, l) = c$*,* $m_2 \sim_l m$*,* $m \sim_l p_2$*,* $r_2 = p$ *and* $O_2 = O$ *there exist* c'*,* m'*,* p' *and* O *such that* $\langle c, m, p, I, O\rangle \rightsquigarrow \langle c', m', p', I, O'\rangle$*,* $\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c_2', l) = c'$*,* $m_2' \sim_l m'$*,* $m' \sim_l p_2'$*,* $r_2' = p'$ *and* $O_2' = O'$*.*

Proof. The proof follows by induction on c_2. We treat the most interesting case, namely, inputs.

Case $c_2 = \mathbf{input}\ x\ \mathbf{from}\ ic$. By inversion on thread-local SME execution we know that there are four cases, i.e. any rule from 9 to 12 can be applied to derive this judgement.

- Assume this derivation is obtained by application of rule 9. In such a case, we know that $\sigma_{in}(ic) > l$. Hence $\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c_2, l) = x_l := dv$. Moreover, we know that $c_2' = \mathbf{skip}$, $m_2' = m_2[x \mapsto dv]$, $r_2' = r_2$, $p_2' = p_2$, $r_2' = r_2$ and $O_2' = O_2$. Take $c' = \mathbf{skip}$, $m' = m[x_l \mapsto dv]$ $p' = p$ and $O' = O$. From standard execution semantics, rule for assignments we know $\langle c, m, p, I, O\rangle \rightsquigarrow \langle c', m', p', I, O'\rangle$. Checking that the relation holds for final states is straightforward.
- Assume this derivation is obtained by application of rule 10. In such a case, we know that $\sigma_{in}(ic) = l$ and

$$\begin{aligned}\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c_2, l) = \mathbf{await}\ (true)\ \mathbf{then}\{&\\ &\mathbf{input}\ x_l\ \mathbf{from}\ ic;\\ &list_{ic} := list_{ic} +\!\!+ [x_l];\\ &count_{ic} := count_{ic} + 1\}\end{aligned}$$

  Also, we know that $c_2' = \mathbf{skip}$, $m_2' = m_2[x \mapsto \mathbf{read}(I, ic, p_2)]$, $p_2' = p_2[ic \mapsto (p_2\ ic) + 1]$, $r_2' = r_2[ic \mapsto (r_2\ ic) + 1]$ and $O_2' = O_2$. Take $c' = \mathbf{skip}$, $m' = m[x_l = \mathbf{read}(I, ic, p)][list_{ic} \mapsto [\![list_{ic} +\!\!+ x_l]\!]m[x_l = \mathbf{read}(I, ic, p)]][count_{ic} \mapsto (mcount_{ic}) + 1]$, $p' = p[ic \mapsto (pic) + 1]$ and $O' = O$. From standard execution semantics, rules for await, sequence, input and assignment we know $\langle c, m, p, I, O\rangle \rightsquigarrow \langle c', m', p', I, O'\rangle$. Checking that the relation holds for final states is straightforward.
- Assume this derivation is obtained by application of rule 11 or 12. In such a case, we know that $\sigma_{in}(ic) < l$ and

$$\begin{aligned}\mathbf{Tr}_{\sigma_{in},\sigma_{out}}(c_2, l) = \mathbf{await}\ (count_{ic,l} < count_{ic})\ \mathbf{then}\{&\\ &x_l := list_{ic}[count_{ic,l}];\\ &count_{ic,l} := count_{ic,l} + 1\}\end{aligned}$$

  There are two subcases, according to the rule applied.

- If the rule 11 was applied, then we know $r_2(ic) \leq p_2(ic)$, $c'_2 = c_2$, $m'_2 = m_2$, $p'_2 = p_2$, $r'_2 = r_2$, $O'_2 = O_2$ and $\dagger = \otimes(ic, p_2(i, c))$. From the relation on initial states we know then that $m(count_{ic}) \leq m(count_{ic,l})$. Take $c' = c$, $m' = m$, $p' = p$ and $O' = O$. $\langle c, m, p, I, O\rangle \overset{count_{ic,l} \leq count_{ic}}{\rightsquigarrow} \langle c', m', p', I, O'\rangle$ follows from standard execution semantics, rule for await. The relation on final states can be checked trivially.
- If the rule 12 was applied, then we know $p_2(ic) < r_2(ic)$, $c'_2 = \mathbf{skip}$, $m'_2 = m_2[x \mapsto readcIicp_2]$, $p'_2 = p_2[ic \mapsto (p_2\ ic) + 1]$, $r'_2 = r_2$ and $O'_2 = O_2$. Take $m' = m[x_l \mapsto [\![list_{ic}[count_c]]\!]m][count_{ic,l} \mapsto (m\ count_{ic,l}) + 1]$, $p' = p$ and $O' = O$. $\langle c, m, p, I, O\rangle \rightsquigarrow \langle c', m', p', I, O'\rangle$ follows from standard execution semantics, rules for await, sequence and assignment. The relation on final states follows from relation on initial states and can be checked easily.

Lemma 5. *Let $\langle P, wq, m, p, I, O\rangle$ and $\langle L, wq', r, I', O'\rangle$ be two configurations such that:*

$$R\langle P, wq, m, p, I, O\rangle\langle L, wq', r, I', O'\rangle$$

If there is a configuration $\langle L_2, wq'_2, r_2, I', O'_2\rangle$ such that:

$$\langle L, wq', r, I', O'\rangle \Rrightarrow^* \langle L_2, wq'_2, r_2, I', O'_2\rangle$$

then there exists P_1, wq_1, m_1, p_1, O_1 such that:

$$\begin{aligned}\langle P, wq, m, p, I, O\rangle &\rightsquigarrow^* \langle P_1, wq_1, m_1, p_1, I, O_1\rangle\\ R\langle P_1, wq_1, m_1, p_1, I, O_1\rangle&\langle L_2, wq'_2, r_2, I', O'_2\rangle\end{aligned}$$

Proof. Follows by induction on the derivation of

$$\langle L, wq', r, I', O'\rangle \Rrightarrow^* \langle L_2, wq'_2, r_2, I', O'_2\rangle$$

The first case, in which there are no executions steps is straightforward.
For the second case we know that:

$$\begin{aligned}\langle L, wq', r, I', O'\rangle &\Rrightarrow^* \langle L_2, wq'_2, r_2, I', O'_2\rangle\\ \langle L_2, wq'_2, r_2, I', O'_2\rangle &\Rrightarrow \langle L_4, wq'_4, r_4, I'_4, O'_4\rangle\end{aligned}$$

Also, by induction hypothesis we know there is a configuration $\langle P_1, wq_1, m_1, p_1, I, O_1\rangle$ such that:

$$\begin{aligned}\langle P, wq, m, p, I, O\rangle &\rightsquigarrow^* \langle P_1, wq_1, m_1, p_1, I, O_1\rangle\\ R\langle P_1, wq_1, m_1, p_1, I, O_1\rangle&\langle L_2, wq'_2, r_2, I', O'_2\rangle\end{aligned}$$

It then suffices to show that there exists a configuration $\langle P_3, wq_3, m_3, p_3, I, O_3\rangle$ such that

$$\langle P_1, wq_1, m_1, p_1, I, O_1\rangle \rightsquigarrow \langle P_3, wq_3, m_3, p_3, I, O_3\rangle$$
$$R\langle P_3, wq_3, m_3, p_3, I, O_3\rangle\langle L_4, wq_4', r_4, I_4', O_4'\rangle$$

We proceed by case analysis in the derivation of

$$\langle L_2, wq_2', r_2, I', O_2'\rangle \Rrightarrow \langle L_4, wq_4', r_4, I_4', O_4'\rangle$$

There are four cases, each corresponding to the application of one of the rules of global SME semantics.

- Case (1) corresponds to a local step for a given thread. In such a case, we know that $\mathbf{select}(L_2) = \langle c, m, p\rangle$ and $\langle c, m, p\rangle_l, r_2, I', O_2' \Rrightarrow \langle c', m', p'\rangle_l, r_4, I', O_4'$. Also, we conclude $L_4 = L_2\ \{\langle c, m, p\rangle_l\} \cup \{\langle c, m, p\rangle_l\}$ and $wq_4' = wq_2'$. From $P_1 \sim_{m_1} L_2$ and the equal scheduler hypothesis we know that $\mathbf{select}(P_1) = \mathbf{select}(L_2)$. Take $\mathbf{select}(P_1) = (l, c_2)$. From Lemma 4, we know that there exists c^*, m^*, p^*, O^* such that $\langle c_2, m_1, p_1, I, O_1\rangle \rightsquigarrow \langle c^*, m^*, p^*, I, O^*\rangle$. Take $P_3 = P_1/\{(l, c_2)\} \cup \{(l, c^*)\}$, $wq_3 = wq_1$, $m_3 = m^*$, $p_3 = p^*$ and $O_3 = O^*$. R ensures that if a step on SME has not emitted a read channel signal then $\{(id, c)|(b, (id, c)) \in wq_1 \wedge [\![b]\!]m_3)\}$ is empty. Hence, we apply the third rule of global semantics for standard execution. Checking that R relates post-states is then straightforward.
- Case (2) corresponds to the situation in which the picked thread has already already finished. The result follows from the fact that R ensures $P_1 \sim_{m_1} L_2$., the equal scheduler choice hypothesis and the first rule of the global semantics for standard execution.
- Case (3) corresponds to the in which a thread reads from a channel whose security level is the same in which the thread is running. We know that $\mathbf{select}(L_2) = \langle c, m, p\rangle$ and $\langle c, m, p\rangle_l, r_2, I', O_2' \overset{\odot(ic,n)}{\Rrightarrow} \langle c', m', p'\rangle_l, r_4, I', O_4'$. Also, we conclude $L_4 = L_2\ \{\langle c, m, p\rangle_l\} \cup \{\langle c, m, p\rangle_l\} \cup wq_2'(ic, n)$ and $wq_4' = wq_2'[(i, n) \mapsto \{\}]$. From $P_1 \sim_{m_1} L_2$ and the equal scheduler hypothesis we know that $\mathbf{select}(P_1) = \mathbf{select}(L_2)$. Take $\mathbf{select}(P_1) = (l, c_2)$. From Lemma 4, we know that there exists c^*, m^*, p^*, O^* such that $\langle c_2, m_1, p_1, I, O_1\rangle \rightsquigarrow \langle c^*, m^*, p^*, I, O^*\rangle$. Take $P_3 = P_1/\{(l, c_2)\} \cup \{(l, c^*)\} \cup \{(id, c)|(b, (id, c)) \in wq_1 \wedge [\![b]\!]m_3)\}$,

  $$wq_3 = \{(b, (id, c))|(b, (id, c)) \in wq_1 \wedge \neg[\![b]\!]m_3$$

  , $m_3 = m^*$, $p_3 = p^*$ and $O_3 = O^*$. R ensures that if a step on SME has emitted a read channel signal then $\{(id, c)|(b, (id, c)) \in wq_1 \wedge [\![b]\!]m_3)\} \sim_m wq_2'(i, n)$. Hence, we apply the third rule of global semantics for standard execution. Checking that R relates post-states is then straightforward.
- Case (4) corresponds to the case in which the picked thread cannot progress because a lower thread has to read input on a channel. We know that $\mathbf{select}(L_2) = \langle c, m, p\rangle$ and $\langle c, m, p\rangle_l, r_2, I', O_2' \overset{\otimes(ic,n)}{\Rrightarrow} \langle c, m, p\rangle_l, r_2, I', O_2'$. We know then that $L_4 = L_2\ \{\langle c, m, p\rangle_l\}$ and $wq_4' = wq_2'[(i, n) \mapsto \{\langle c, m, p\rangle_l\}]$. From $P_1 \sim_{m_1} L_2$ and the equal scheduler hypothesis we know that $\mathbf{select}(P_1) = \mathbf{select}(L_2)$.

Take $\mathbf{select}(P_1) = (l, c_2)$. From Lemma 4, we know that $\langle c_2, m_1, p_1, I, O_1\rangle \overset{b}{\rightsquigarrow} \langle c_2, m_1, p_1, I, O_1$. Take $P_3 = P_1/\{l, c_2\}$ and $wq_3 = wq_1 \cup \{l, c_2\}$. Using the second rule of global standard execution we conclude the desired result. Checking that R relates the post-states is straightforward.

Proof of Theorem 3, part two. For every program P, and program input I if

$$\langle P, I\rangle \Rightarrow^* \langle p, O\rangle$$

then

$$\langle \| \{(l, \mathbf{Tr}_{\sigma_{in},\sigma_{out}}(P, l))|l \in \mathcal{L}\}, I\rangle \rightsquigarrow^* \langle p, O\rangle$$

Proof. By definition, we know that

$$\langle L_{P,0}, wq_0, r_0, I, O_0\rangle \Rightarrow^* \langle L_f, wq_f, p, I, O\rangle$$

Using Lemma 1, first part, we know:

$$R\langle \| \{(l, \mathbf{Tr}_{\sigma_{in},\sigma_{out}}(P, l))|l \in \mathcal{L}\}, \{\}, m_{0,\sigma_{in},\sigma_{out}}, p_0, I, O_0\rangle \langle L_{P,0}, wq_0, r_0, I, O\rangle$$

Using Lemma 5 we know that there exists L_2, wq_2', r_2, O_2' such that:

$$\begin{array}{c}\langle L_{P,0}, wq_0, r_0, I, O\rangle \Rightarrow^n \langle L_2, wq_2', r_2, I, O_2'\rangle \\ R\langle\{\}, \{\}, m, p, I, O\rangle\langle L_2, wq_2', r_2, I, O_2'\rangle\end{array}$$

we know that it is a final configuration, using the last part of Lemma 1 and the fact that $\langle\{\}, \{\}, m, p, I, O\rangle$ is a final configuration. Hence, we conclude:

$$\langle P, I\rangle \Rightarrow^* \langle r_2, O_2'\rangle$$

From the definition of R, we know that $r_2 = p$ and $O_2' = O$.

A.2 Proofs of Subsection 4.3

We start by defining a relation between configurations of sequential standard execution and global concurrent executions.

Definition 9. *Let $\langle c, m_1, p_1, I_1, O_2\rangle$ be a configuration of the sequential language and $\langle P, wq_2, m_2, p_2, I_2, O_2\rangle$ we define a relation R between such configurations as follows:*

$$\begin{array}{rl} R\langle c, m_1, p_1, I_1, O_2\rangle\langle P, wq_2, m_2, p_2, I_2, O_2\rangle \overset{def}{=} & c \sim P \wedge m_1 = m_2 \wedge p_1 = p_2 \wedge \\ & I_1 = I_2 \wedge O_1 = O_2 \wedge wq_2 = [] \wedge \\ & \forall ic, n, [\![list_{ic}[n]]\!]m_1 \neq \perp \Leftrightarrow \\ & \quad [\![n < count_{ic}]\!]m_1 \\ \textit{where } P \sim c \overset{def}{=} & \exists k, c = c_k; \ldots; c_n \wedge \forall i \in [k \ldots n], \\ & \exists c', \mathbf{Tr}^{\mathbf{seq}}(c', i) = c_i \\ & \wedge (i, \mathbf{Tr}(c', i)) \in P \end{array}$$

Lemma 6. *R satisfies the following properties:*

- *Initial execution states are related by R, i.e.:*

$$R\langle \| \{(l, \mathbf{Tr}_{\sigma_{in},\sigma_{out}}(P,l)) | l \in \mathcal{L}\}, \{\}, m_{0,\sigma_{in},\sigma_{out}}, p_0, I, O_0\rangle \langle L_{P,0}, wq_0, r_0, I, O\rangle$$

- *If two configurations are related by R, one is final iff the other one is final, i.e.:*

$$R\langle P, wq, m, p, I, O\rangle\langle L, wq', r, I', O'\rangle \Rightarrow$$
$$(P = \{\} \wedge wq = \{\} \Leftrightarrow L = \{\} \wedge \forall i, n, wq'(i,n) = \{\})$$

Lemma 7. *For any configuration $\langle c_1, m_1, p_1, I, O_1\rangle$ of sequential execution and any thread local configuration of concurrent execution $\langle c_2, m_2, p_2, I, O_2\rangle$ such that $m_1 = m_2$, $p_1 = p_2$, $O_1 = O_2$ and there exists c and l such that $\mathbf{Tr}^{\mathbf{seq}}(c,l) = c_1$ and $\mathbf{Tr}(c,l) = c_2$ if $\langle c_1, m_1, p_1, I, O_1\rangle \rightsquigarrow \langle c_1', m_1', p_1', I, O_1'\rangle$ then there exists c_2', m_2', p_2', O_2' such that $\langle c_2, m_2, p_2, I, O_2\rangle \rightsquigarrow \langle c_2', m_2', p_2', I, O_2'\rangle$ and $m_1' = m_2'$, $p_1' = p_2'$, $O_1' = O_2'$ and there exists c' such that $\mathbf{Tr}^{\mathbf{seq}}(c', l) = c_1'$ and $\mathbf{Tr}(c', l) = c_2'$.*

Proof. The proof follows by induction on c, i.e. the statement of the original program from which c_1 and c_2 are obtained by transformation. Cases corresponding to statements such as sequence, while and if follow from direct application of the induction hypothesis and the fact that the transformations coincide when applied to the same command in all cases but inputs. We focus on the latter.

- Case $c = \mathbf{input}\ x\ \mathbf{from}\ ic$. The result of the transformation in this case depends on the relationship between l and $\sigma_{in}(ic)$. We proceed by case analysis.
  - Case $\sigma_{in}(ic) > l$. We know that $c_1 = x_l := dv$ and $c_2 = x_l := dv$. The result is then immediate.
  - Case $\sigma_{ic}(ic) < l$. We know that

$$c_1 = \mathbf{atomic}\ (x_l := list_{ic}[count_{ic,l}]; count_{ic,l} := count_{ic,l} + 1)$$

    and

$$c_2 = \mathbf{await}\ count_{ic,l} < count_{ic}\ \mathbf{then}$$
$$(x_l := list_{ic}[count_{ic,l}]; count_{ic,l} := count_{ic,l} + 1)$$

    Since $\langle c_1, m_1, p_1, I, O_1\rangle \rightsquigarrow \langle c_1', m_1', p_1', I, O_1'\rangle$, $[\![list_{ic}[count_{ic,l}]]\!]m_1 \neq \perp$. Hence, we know that $[\![count_{ic,l} < count_{ic}]\!]m_1$ and since $m_1 = m_2$, follows that $[\![count_{ic,l} < count_{ic}]\!]m_2$. We also know that $p_1' = p_1$, $c_1' = \mathbf{skip}$, $O_1' = O_1$ and

$$m_1' = m_1[count_{ic,l} \mapsto [\![count_{ic,l}]\!]m_1 + 1][x_l \mapsto [\![list_{ic}[count_{ic,l}]]\!]m_1]$$

    The result follows by setting $c_2' = \mathbf{skip}$, $m_2' = m_1'$, $p_2' = p_1'$ and $O_2' = O_1'$ from the rule for await commands of the concurrent thread local semantics.
  - Case $\sigma_{ic}(ic) < l$. We know

$$c_1 = \mathbf{atomic}\ (\mathbf{input}\ x\ \mathbf{from}\ ic; list_{ic} := list_{ic} +\!\!+ [x]; count_{ic} := count_{ic} + 1)$$

    and $c_2 = c_1$. The result is then immediate.

Lemma 8. *Let* $\langle c_1, m_1, p_1, I, O_1\rangle$ *and* $\langle P_2, wq_2, m_2, p_2, I, O_2\rangle$ *be configurations of sequential execution and concurrent execution respectively such that*

$$R\langle c_1, m_1, p_1, I, O_1\rangle\langle P_2, wq_2, m_2, p_2, I, O_2\rangle$$

If $\langle c_1, m_1, p_1, I, O_1\rangle \rightsquigarrow^n \langle c_1', m_1', p_1', I, O_1'\rangle$ *then there exists* $P_2', wq_2', m_2', p_2', O_2'$ *such that* $\langle P_2, wq_2, m_2, p_2, I, O_2\rangle \rightsquigarrow^n \langle P_2', wq_2', m_2', p_2', I, O_2'\rangle$ *and*

$$R\langle c_1', m_1', p_1', I, O_1'\rangle\langle P_2', wq_2', m_2', p_2', I, O_2'\rangle$$

Proof. Follows by induction on the derivation of

$$\langle c_1, m_1, p_1, I, O_1\rangle \rightsquigarrow^n \langle c_1', m_1', p_1', I, O_1'\rangle$$

The case in which there is no step is straightforward.
For the second case, taking $m + 1 = n$ we know that:

$$\begin{aligned}\langle c_1, m_1, p_1, I, O_1\rangle &\rightsquigarrow^m \langle c_1'', m_1'', p_1'', I, O_1''\rangle\\ \langle c_1'', m_1'', p_1'', I'', O_1''\rangle &\rightsquigarrow \langle c_1', m_1', p_1', I, O_1'\rangle\end{aligned}$$

Also, by inductive hypothesis, we know there exists $P_2'', wq_2'', m_2'', p_2'', I, O_2''$ such that

$$\begin{aligned}\langle P_2, wq_2, m_2, p_2, I, O_2\rangle &\rightsquigarrow^m \langle P_2'', wq_2'', m_2'', p_2'', I, O_2''\rangle\\ R\langle c_1'', m_1'', p_1'', I, O_1''\rangle\langle P_2'', wq_2'', m_2'', p_2'', I, O_2''\rangle&\end{aligned}$$

Then it suffices to show that there is $P_2', wq_2', m_2', p_2', I, O_2'$ such that

$$\begin{aligned}\langle P_2'', wq_2'', m_2'', p_2'', I, O_2''\rangle &\rightsquigarrow \langle P_2', wq_2', m_2', p_2', I, O_2'\rangle\\ R\langle c_1', m_1', p_1', I, O_1'\rangle\langle P_2', wq_2', m_2', p_2', I, O_2'\rangle&\end{aligned}$$

From the definition of R, we know there is k such that $c_1'' = c_k''; \ldots; c_n''$ such that for all $i \in [k, \ldots, n]$ there is c'' such that $\mathbf{Tr}^{\mathbf{seq}}(c'', i) = c_i''$ and $(i, \mathbf{Tr}(c'', i)) \in P_2''$. In particular, for we know that there is c'' such that $\mathbf{Tr}^{\mathbf{seq}}(c'', k) = c_k''$ and $(k, \mathbf{Tr}(c'', k)) \in P_2''$. Since we are dealing with the $lowprio$ scheduler, $\mathbf{select}(P_2'') = (k, \mathbf{Tr}(c'', k))$. Also, by inversion on sequential execution, from

$$\langle c_k''; \ldots; c_n'', m_1'', p_1'', I'', O_1''\rangle \rightsquigarrow \langle c_k''^*; \ldots; c_n'', m_1', p_1', I, O_1'\rangle$$

we know that $\langle c_k'', m_1'', p_1'', I'', O_1''\rangle \rightsquigarrow \langle c_k''^*, m_1', p_1', I, O_1'\rangle$. Using this fact, and R we fulfill the hypothesis of Lemma 7, hence we conclude that there is $c_2''^*$, m_2', p_2', O_2' such that $\langle \mathbf{Tr}(c'', k), m_2'', p_2'', I, O_2''\rangle \rightsquigarrow \langle c_2''^*, m_2', p_2', I, O_2'\rangle$ and $m_2' = m_1'$, $p_2' = p_1'$, $O_2' = O_1'$ and there exists c''^* such that $\mathbf{Tr}(c''^*, k) = c_2''^*$ and $\mathbf{Tr}^{\mathbf{seq}}(c''^*, k) = c_1''^*$. Then, we take $P_2' = P_2'' \cup \{(k, c_2''^*)\}/\{(k, c_k'')\}$ and $wq_2'' = 0$ and using rule 3 of global semantics of concurrent execution conclude the desired result. Checking that R holds for final configurations is direct.

Proof of Theorem 4, part one. From definition of $\langle \mathbf{Tr}^{\mathbf{seq}}(P), I\rangle \rightsquigarrow^n \langle p, O\rangle$ we know that there exists c'_1, m'_1 such that $\langle \mathbf{Tr}^{\mathbf{seq}}(P), m_{0,P}, I, O_0\rangle \rightsquigarrow^n \langle c'_1, m'_1, p, I, O\rangle$. From Lemma 6, we know that if $\langle \mathbf{Tr}(P), wq_0, m_{0,p}, p_0, I, O_0\rangle$ is an initial configuration of concurrent execution then $R\langle \mathbf{Tr}^{\mathbf{seq}}(P), m_{0,P}, I, O_0\rangle\langle \mathbf{Tr}(P), wq_0, m_{0,p}, p_0, I, O_0\rangle$. From Lemma 8, we know that there is P'_2, wq'_2, m'_2, O'_2 such that

$$\langle \mathbf{Tr}(P), wq_0, m_{0,P}, I, O_0\rangle \rightsquigarrow^n \langle P'_2, wq'_2, p'_2, m'_2, I, O'_2\rangle$$

and that R holds on the final states. By definition, we know $\langle \mathbf{Tr}(P), I\rangle \rightsquigarrow^n \langle p'_2, O'_2\rangle$. From R we conclude that $p'_2 = p$ and $O'_2 = O$.

Now will prove the second part of Theorem 4. To do so, we rely on a property proved in [12], which we state next.

Proposition 1. *If P is non-interferent and $\langle L_{P,0}, wq_0, r_0, I, O_0\rangle \Rightarrow^* \langle L', wq', r', I, O'\rangle$ using the $lowprio$ scheduler, then $wq' = \{\}$. Moreover, any other intermediate configuration $\langle L'', wq'', r'', I, O''\rangle$ such that $\langle L_{P,0}, wq_0, r_0, I, O_0\rangle \Rightarrow \langle L'', wq'', r'', I, O''\rangle$ and $\langle L'', wq'', r'', I, O''\rangle \Rightarrow^* \langle L', wq', r', I, O'\rangle$ has the same property, i.e. $wq'' = \{\}$.*

Using this lemma, we state a new invariant of concurrent execution using $lowprio$ scheduler, namely, the fact that waiting queues remain empty during execution.

Lemma 9. *If P is non-interferent and*

$$\langle \mathbf{Tr}(P), wq_0, m_0, p_0, I, O_0\rangle \rightsquigarrow^* \langle P', wq', m', p', I, O'\rangle$$

then $wq' = \{\}$. Moreover, any configuration intermediate $\langle P'', wq'', m'', p'', I, O''\rangle$ such that $\langle \mathbf{Tr}(P), wq_0, m_0, p_0, I, O_0\rangle \rightsquigarrow^* \langle P'', wq'', m'', p'', I, O''\rangle$ and such that $\langle P'', wq'', m'', p'', I, O''\rangle \rightsquigarrow^* \langle P', wq', m', p', I, O'\rangle$ has the same property, i.e. $wq'' = \{\}$.

Proof. We know that if $\langle \mathbf{Tr}(P), wq_0, m_0, p_0, I, O_0\rangle \rightsquigarrow^* \langle P', wq', m', p', I, O'\rangle$ then there exists n such that $\langle \mathbf{Tr}(P), wq_0, m_0, p_0, I, O_0\rangle \rightsquigarrow^n \langle P', wq', m', p', I, O'\rangle$. Then, we apply Lemma 3 and obtain an execution of SME where initial and final configurations are related by R. Since executions of SME for non-interferent programs always feature empty queues in intermediate configurations and R only relates SME configurations with empty queues with concurrent configuration with empty queues, we conclude that the intermediate configurations of concurrent executions also have empty queues.

Corollary 4. *During the concurrent execution of transformed programs that terminate with empty waiting queues under the lowprio scheduler, local executions never emit blocking guards.*

Proof. By contradiction. Assume there is a global configuration $\langle P, wq, m, p, I, O\rangle$ such that $\mathbf{select}(P) = (id, c)$ and $\langle c, m, p, I, O\rangle \overset{b}{\rightsquigarrow} \langle c, m, p, I, O\rangle$. Using global semantics, rule 2, we obtain $\langle P, wq, m, p, I, O\rangle \rightsquigarrow \langle P', wq', m, p, I, O\rangle$ with $wq \neq \{\}$ which contradicts Lemma 9.

Lemma 10. *For any thread local configuration $\langle c_2, m_2, p_2, I, O_2\rangle$ of concurrent execution and any configuration of sequential execution $\langle c_1, m_1, p_1, I, O_1\rangle$ such that $m_1 = m_2$, $p_1 = p_2$, $O_1 = O_2$, if there exists c and l such that $\mathbf{Tr}^{\mathbf{seq}}(c, l) = c_1$, $\mathbf{Tr}(c, l) = c_2$ and if $\langle c_2, m_2, p_2, I, O_2\rangle \rightsquigarrow \langle c_2', m_2', p_2', I, O_2'\rangle$ (without emitting blocking guards) then there exists c_1', m_1', p_1', O_1' such that $\langle c_1, m_1, p_1, I, O_1\rangle \rightsquigarrow \langle c_1', m_1', p_1', I, O_1'\rangle$ and $m_1' = m_2'$, $p_1' = p_2'$, $O_1' = O_2'$ and there exists c' such that $\mathbf{Tr}^{\mathbf{seq}}(c', l) = c_1'$ and $\mathbf{Tr}(c', l) = c_2'$.*

Proof. The proof follows by induction on c, i.e. the statement of the original program from which c_1 and c_2 are obtained by transformation. Cases corresponding to statements such as sequence, while and if follow from direct application of the induction hypothesis and the fact that the transformations coincide when applied to the same command in all cases but inputs. We focus on the latter.

- Case $c = \mathbf{input}\ x\ \mathbf{from}\ ic$. The result of the transformation in this case depends on the relationship between l and $\sigma_{in}(ic)$. We proceed by case analysis.
  - Case $\sigma_{in}(ic) > l$. We know that $c_1 = x_l := dv$ and $c_2 = x_l := dv$. The result is then immediate.
  - Case $\sigma_{ic}(ic) < l$. We know that

    $$c_2 = \mathbf{await}\ count_{ic,l} < count_{ic}\ \mathbf{then}\\ (x_l := list_{ic}[count_{ic,l}]; count_{ic,l} := count_{ic,l} + 1)$$

    and $c_1 = \mathbf{atomic}\ (x_l := list_{ic}[count_{ic,l}]; count_{ic,l} := count_{ic,l} + 1)$ Also, note that $\langle c_2, m_2, p_2, I, O\rangle \rightsquigarrow \langle c_2', m_2', p_2', I, O_2'\rangle$ cannot emit a signal since we are assuming it does not. Hence we now that the guard $count_{ic,l} < count_{ic}$ holds and that $list_{ic}[count_{ic,l}]$ is defined. Applying thread local execution, we know $m_2' = m_2[count_{ic,l} \mapsto [\![count_{ic,l}]\!]m_2 + 1][x_l \mapsto [\![list_{ic}[count_{ic,l}]]\!]m_2]$, $c_2' = \mathbf{skip}$, $p_2' = p_2$ and $O_2' = O_2$. The result then follows by setting $c_1' = \mathbf{skip}$, $m_1' = m_2'$, $p_1' = p_2'$, $O_1' = O_2'$. from the rule of sequence, force and assignment of the sequential semantics.
  - Case $\sigma_{ic}(ic) < l$. We know

    $$c_1 = \mathbf{atomic}\ (\mathbf{input}\ x\ \mathbf{from}\ ic; list_{ic} := list_{ic} +\!\!+ [x]; count_{ic} := count_{ic} + 1)$$

    and $c_2 = c_1$. The result is then immediate.

Lemma 11. *Let $\langle c_1, m_1, p_1, I, O_1\rangle$ and $\langle P_2, wq_2, m_2, p_2, I, O_2\rangle$ be configurations of sequential execution and concurrent execution respectively such that*

$$R\langle c_1, m_1, p_1, I, O_1\rangle\langle P_2, wq_2, m_2, p_2, I, O_2\rangle$$

Also, assume that during any concurrent execution starting from $\langle P_2, wq_2, m_2, p_2, I, O_2\rangle$, no local steps emit blocking guards. Then it is the case that if $\langle P_2, wq_2, m_2, p_2, I, O_2\rangle \rightsquigarrow^* \langle P_2', wq_2', m_2', p_2', I, O_2'\rangle$ there exists $c_1', m_1', p_1', I, O_1'$ such that $\langle c_1, m_1, p_1, I, O_1\rangle \rightsquigarrow^* \langle c_1', m_1', p_1', I, O_1'\rangle$ and $R\langle c_1', m_1', p_1', I, O_1'\rangle\langle P_2', wq_2', m_2', p_2', I, O_2'\rangle$.

Proof. Follows by induction on the derivation of

$$\langle P_2, wq_2, m_2, p_2, I, O_2\rangle \rightsquigarrow^* \langle P_2', wq_2', m_2', p_2', I, O_2'\rangle$$

The case in which there is no step is straightforward.
For the second case we know that:

$$\langle P_2, wq_2, m_2, p_2, I, O_2\rangle \rightsquigarrow^* \langle P_2'', wq_2'', m_2'', p_2'', I, O_2''\rangle$$
$$\langle P_2'', wq_2'', m_2'', p_2'', I, O_2''\rangle \rightsquigarrow \langle P_2', wq_2', m_2', p_2', I, O_2'\rangle$$

Also, by inductive hypothesis, we know there exist $c_1'', m_1'', p_1'', I, O_1''$ such that

$$\langle c_1, m_1, p_1, I, O_1\rangle \rightsquigarrow^* \langle c_1'', m_1'', p_1'', I, O_1''\rangle$$
$$R\langle c_1'', m_1'', p_1'', I, O_1''\rangle\langle P_2'', wq_2'', m_2'', p_2'', I, O_2''\rangle$$

Then, it suffices to show that there exist $c_1', m_1', p_1', I, O_1'$ such that

$$\langle c_1'', m_1'', p_1'', I'', O_1''\rangle \rightsquigarrow \langle c_1', m_1', p_1', I, O_1'\rangle$$
$$R\langle c_1', m_1', p_1', I, O_1'\rangle\langle P_2', wq_2', m_2', p_2', I, O_2'\rangle$$

We now perform case analysis on the derivation of

$$\langle P_2'', wq_2'', m_2'', p_2'', I, O_2''\rangle \rightsquigarrow \langle P_2', wq_2', m_2', p_2', I, O_2'\rangle$$

- Case 1. For this case, we know $\mathbf{select}(P_2'') = (l, \mathbf{skip})$. Since we are using the $lowprio$ scheduler, we know that l is the minimal security level with a local configuration associated in P_2''. From R, we know that then $c_1'' = \mathbf{skip}; c_k; \ldots; c_n$. Also, from the execution we know that $P_2' = P_2''/\{(id, \mathbf{skip})\}$, $wq_2' = wq_2''$, $p_2' = p_2''$ and $O_2' = O_2''$. Take $c_1' = c_k; \ldots; c_n$, $m_1' = m_1''$, $p_1' = p_1''$ and $O_1' = O_2''$. Then, from rule of sequence we conclude $\langle c_1'', m_1'', p_1'', I'', O_1''\rangle \rightsquigarrow \langle c_1', m_1', p_1', I, O_1'\rangle$. Checking that R relates final configurations is straightforward.
- Case 2. Local executions do not emit guards, by assumption.
- Case 3. From R we know that there is a k such that $c_1'' = c_k; \ldots; c_n$ and that for any i between k and n, there is a c, such that $\mathbf{Tr}_c^{\mathbf{seq}}(i, =)c_k$ and $(i, \mathbf{Tr}(c, i)) \in P_1''$. The $lowprio$ scheduler will pick the thread corresponding to k, since it is minimal, i.e. $\mathbf{select}(P_2'') = (k, c_2''^*)$ with $c_2^* = \mathbf{Tr}_c(k,)$. Assume then, that we have a local step $\langle c_2''^*, m_2'', p_2'', I, O_2''\rangle \rightsquigarrow \langle c_2'^*, m_2', p_2', I, O_2'\rangle$. From Lemma 10, we know that there exists c_1', m_1', p_1', O_1' such that $\langle c_1'', m_1'', p_1'', I, O_1''\rangle \rightsquigarrow \langle c_1', m_1', p_1', I, O_1'\rangle$ and $m_1' = m_2'$, $p_1' = p_2'$, $O_1' = O_2'$ and there exists c' such that $\mathbf{Tr}^{\mathbf{seq}}(c', l) = c_1'$ and $\mathbf{Tr}(c', l) = c_2'^*$. Also, we know that $P_2' = P_2''/\{(k, c_2''^*)\} \cup \{(k, c_2'^*)\}$ and $wq_2'' = \{\} = wq_1'$. Using sequence rule of sequential execution we conclude $\langle c_1'', m_1'', p_1'', I'', O_1''\rangle \rightsquigarrow \langle c_1', m_1', p_1', I, O_1'\rangle$. Checking that R relates final configurations is straightforward.

Proof of Theorem 4, part two.

Proof. From definition of $\langle \mathbf{Tr}(P), I\rangle \rightsquigarrow^* \langle p, O\rangle$ we know that there exists P_2', wq_2', m_2' such that $\langle \mathbf{Tr}(P), wq_0, m_{0,P}, I, O_0\rangle \rightsquigarrow^* \langle P_2', wq_2', p, m_2', I, O\rangle$. Since P is non-interferent, we know that $wq_2' = \{\}$. From Lemma 6, we know that for the initial configuration of sequential SME execution $\langle \mathbf{Tr^{seq}}(P), m_{0,P}, I, O_0\rangle$,

$$R\langle \mathbf{Tr^{seq}}(P), m_{0,P}, I, O_0\rangle\langle \mathbf{Tr}(P), wq_0, m_{0,p}, p_0, I, O_0\rangle$$

. Also, from Corollary 4, we know local steps in the concurrent execution never emit blocking guards. From Lemma 11, we know that there is c_1', m_1', p_1', O_1' such that $\langle \mathbf{Tr^{seq}}(P), m_{0,P}, I, O_0\rangle \rightsquigarrow^* \langle c_1', m_1', p_1', I, O_1'\rangle$ and R holds on final configurations. Hence, by definition, we conclude $\langle \mathbf{Tr^{seq}}(P), I\rangle \rightsquigarrow^* \langle p_1', O_1'\rangle$. From R we know that $p_1' = p$ and $O_1' = O$.