Secure Multi-Execution: Fine-grained, Declassification-aware, and Transparent

Willard Rafnsson Andrei Sabelfeld
Chalmers University of Technology, Gothenburg, Sweden

***Abstract*—Recently, much progress has been made on achieving information-flow security via secure multi-execution. Secure multi-execution (SME) is an elegant way to enforce security by executing a given program multiple times, once for each security level, while carefully dispatching inputs and ensuring that an execution at a given level is responsible for producing outputs for information sinks at that level. Secure multi-execution guarantees noninterference, in the sense of no dependencies from secret inputs to public outputs, and transparency, in the sense that if a program is secure then its secure multi-execution does not destroy its original behavior.**

This paper pushes the boundary of what can be achieved with secure multi-execution. First, we lift the assumption from the original secure multi-execution work on the totality of the input environment (that there is always assumed to be input) and on the cooperative scheduling. Second, we generalize secure multi-execution to distinguish between security levels of presence and content of messages. Third, we introduce a declassification model for secure multi-execution that allows expressing what information can be released. Fourth, we establish a full transparency result showing how secure multi-execution can preserve the original order of messages in secure programs. We demonstrate that full transparency is a key enabler for discovering attacks with secure multi-execution.

I. Introduction

As modern attacks are becoming more sophisticated, there is an increasing demand for more advanced protection measures than those offerred by standard security practice. We exemplify an instance of the problem with a motivating scenario from web application security, but note that the problem is of rather general nature.

Motivation: In the context of the web, third-party script inclusion is pervasive. It drives the integration of advertisement and statistics services. As an indicative example, *barackobama.com* at the time of the 2012 US presidential campaign contained 76 different third-party tracking scripts [SD12]. The tracking was used for target political advertisement. Script inclusions extend the trusted computing base to the Internet domains of included scripts. This creates dangerous scenarios of trust-abuse. This can be done either by direct attacks from the included scripts or, perhaps more dangerously, by indirect attacks when a popular service is compromised and its scripts are replaced by the attacker. A recent empirical study [NIK$^+$12] of script inclusion reports high reliance on third-party scripts. It outlines new attack vectors showing how easy it is to get code running in thousands of browsers simply by acquiring some stale or misspelled domains. Access control mechanisms are of limited use because third-party scripts require access to sensitive information for their proper functionality. This is particularly important for statistics and context-aware advertisement services on the web. Similar scenarios arise in the setting of cloud computing where sharing the resources is desirable but without compromising confidentiality and integrity. This motivates the need for fine-grained *information-flow control*.

From static to dynamic information-flow control: Tracking information flow in programs is a popular area of research. Static analysis techniques have been extensively explored, leading to tools like Jif [MZZ$^+$01], FlowCaml [Sim03], and SparkAda Examiner [Bar03] that enhance compilers for Java, Caml, and Ada, respectively. Recently, dynamic monitoring techniques have received increased attention (cf. [LBJS06], [Le 07], [SST07], [SR09], [AF09], [AF10], [HS12]), driven by the demand to analyze dynamic programming languages like JavaScript. While static analysis either accepts or rejects a given program before it is run, dynamic monitors perform checks at run time. There are known fundamental tensions [RS10] between static and dynamic analyses, implying that none is superior to the other. Although dynamic analysis might seem intuitively more permissive, it has to conservatively treat the paths that are not taken by the current execution.

Secure multi-execution (SME): Recently, there has been much progress on SME [DP10], [BDMP11a], [KWH11], [JR11], [AF12], [BCD$^+$12], [GDNP12], a runtime enforcement for information flow. In contrast to the monitoring techniques, the goal is not to prevent insecurities but to "repair" them on the fly. This approach is secure by design: security is achieved by separation of computations at different security levels. The original program is run as many times as there are security levels, where outputs at a given security levels are only allowed if the security level of the program is matched with the security level of the output channel. The handling of inputs is slightly more involved because inputs from less restrictive security levels are allowed to be used in computations at more restrictive levels. Secure multi-execution propagates inputs, once they are received, to the runs of the program that are responsible for the computation of outputs at more restrictive levels.

Typically, security levels are drawn from a lattice with the intuition that information from an input source at level ℓ may flow to an output sink at level ℓ' only if $\ell \sqsubseteq \ell'$ [Den76]. For simplicity, we will often use the two-level lattice with a *secret* (*high*) level and a *public* (*low*) level of confidentiality. Figure 1 shows program P with a pair of input sources, labeled high H and low L, and a similarly-labeled pair of sinks. The baseline policy of *noninterference* [GM82] demands that low outputs do not depend on high inputs.

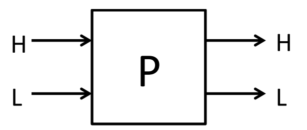

Figure 1: Original execution

Figure 2 shows how secure multi-execution achieves noninterference. Program P is run twice, as P_H at high and as P_L at low levels. The high input is fed into the high run. The low input is fed into both the low and high run. Dummy default values are used whenever the low run asks for high input. High output is produced by the high run, and the low output is produced by the low run, while low output of the high run and high output of the low run are ignored. It is clear from the diagram that noninterference is enforced because the low run, the only producer of low output, never gets access to high input.

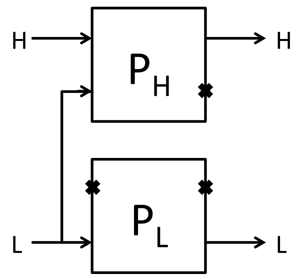

Figure 2: Secure multi-execution

In contrast to the traditional dynamic analysis, there is no concern about executions not taken because the control flow of the low run cannot possibly be affected by high input. Further, secure multi-execution provides *transparency*, in the sense that if a program is secure then its secure multi-execution does not destroy its original behavior.

Contributions: While secure multi-execution gains increased popularity, there are open challenges that need to be addressed before it can be applied widely. We overview the pros and cons of secure multi-executions compared to traditional information-flow control and, among other findings, point out that secure multi-execution (i) lacks support for fine-grained security levels for communication channels, (ii) relies on restrictive scheduling, (iii) lacks support for declassification, (iv) may reorder messages wrt. the original execution, and (v) lacks support for detecting attacks.

We push the boundary of what can be achieved with secure multi-execution. First, we lift the assumption from the original secure multi-execution work on the totality of the input environment (that there is always assumed to be input) and on cooperative scheduling. Second, we generalize secure multi-execution to distinguish between security levels of presence and content of messages. Third, we introduce a declassification model for secure multi-execution that allows expressing what information can be released. Fourth, we establish full transparency showing how secure multi-execution can preserve the original order of messages in secure programs by barrier synchronization. This enables the use of secure multi-execution to discover attacks on runtime.

II. Pros and cons of secure multi-execution

We overview the pros and cons of secure multi-execution with respect to direct information-flow enforcement. The overview has two goals: provide a general basis for deciding on which enforcement mechanism to pick in a particular case and identify the most pressing shortcomings, subject to improvements by this paper. We start by listing of what we view as the pros of secure multi-execution.

Noninterference by design: A significant advantage of secure multi-execution is that it straightforwardly enforces noninterference by a simple access-control discipline: computation responsible for output at a given level never gets access to information at more restrictive or incomparable levels. This provides noninterference guarantees.

Language-independence: A major benefit is that secure multi-execution can be enforced in a blackbox, language-independent, fashion. The enforcement only concerns input and output operations allowing the rest of the language to be arbitrarily complex. This is particularly useful for dynamic languages like JavaScript that are hard to analyze.

Transparency for secure programs: If the original program is secure, there are transparency guarantees that limit ways in which semantics can be modified. The original work on secure multi-execution shows *per-channel* transparency (or *precision* in the terminology of Devriese and Piessens [DP10]). This means that if the original program is secure then, from the viewpoint of each channel, the sequence of I/O events in a given run of a program is the same in the original run and in the multi-executed run.

Transparency at top level: In addition, we note another transparency property, which applies, for example, to programs with no intermediate input: the externally-observable program behavior at the top security level is the same for the original and securely multi-executed runs. This property can be seen from Figures 1 and 2. Clearly, the original run of the program in Figure 1 and the high run of the multi-executed program in Figure 2 get the same inputs. Hence, the high output behaviors are the same no matter whether the original program is secure or not.

We now turn to the cons of secure multi-execution.

Coarse-grained labels: In work on secure multi-execution so far, communication channels are provided with a single security label. This is often too coarse-grained: for example, the presence of a message might be public but the content is secret. This granularity might be useful for statistics services that might be counting different types of events without revealing their content. For example, Google Analytics is routinely used for varios types of counting: how many clicks on the page, how many times a video is played, and how many visitors have viewed a page.

Devriese and Piessens [DP10] assume total input environments: that the input is always present. This does not allow modeling scenarios where the presence of secret input is secret (for example, whether or not the user visits a health web site). Bielova et al. [BDMP11a] allow non-total environments but at the price of ignoring information leaks through termination behavior (targeting termination-insensitive noninterference [VSI96]). This implies that the leaks as in the example with the health site are still ignored.

This motivates the need for fine-grained secure multi-execution. We lift the assumption on total input environments and introduce fine-grained labels for communication channels, where the levels of presence and content of messages are distinguished.

Restrictive scheduling: With the exception of work by Kashyap [KWH11], secure multi-execution heavily relies on the *low-priority* scheduler that lets low computation run until completion before the high run gets a chance to run. The low-priority scheduler is both at the heart of the soundness results by Devriese and Piessens [DP10] and at the heart of FlowFox [GDNP12], an extension of FireFox to enforce secure information flow in JavaScript. The security theorem in the abstract setting of secure multi-execution [DP10] takes advantage of the low-priority scheduler and establishes timing-sensitive security. This is intuitive because the last access of low data occurs before any high data is accessed. Whenever the timing behavior is affected by secrets, there is no possibility for the attacker to inspect the difference.

However, the situation is different in the presence of handlers. The low-priority scheduler does not scale because it is not possible to extend the low-priority discipline over multiple events—simply because it is not possible to run the low handlers that have not yet been triggered. As a compromise, FlowFox [GDNP12] multi-executes JavaScript with the low-priority scheduler on a per-event basis. However, as illustrated by a leak in Appendix A, this strategy is at the cost of timing-sensitive security. All we need to do is to set a low handler to execute after the high run of the main code has finished. Then the low handler can leak via the computation time taken by the high run.

This motivates the need for flexible scheduling strategies and the need for (fair) interleaving of the runs at different levels, as pursued in this paper.

Declassification: Declassification is challenging because secure multi-execution is based on separating information at different security levels. Feeding secret information to a public run might introduce unintended leaks. Coming back to the example of tracking and statistics, we might want to track the popularity of items in a shopping cart or track various average values for transactions.

This motivates the need for declassification in secure multi-execution. The event of declassification should not leak information about the context (branching on a secret and declassifying in the body would leak the boolean value of the secret). It turns out that the support for fine-grained communication channels provides us with a natural treatment of declassification. Indeed, declassification is about communicating a secret value from the high run to the low run, but without leaking through the presence of the communication event. Exactly this is provided by channels with high content and low presence! Hence, a declassification event corresponds to output on a high-content low-presence channel (in view of the high run), and to input on high-content low-presence channel (in the view of the low run).

Order of events modified: The transparency guarantees of secure multi-execution are per channel, allowing the order of events to be modified across different channels. This leads to unexpected results in an interactive setting.

This motivates the need for stronger transparency, where the behavior of secure programs is unmodified across the different levels. We show how to achieve this by careful scheduling of the runs at the different levels.

Silent failure: The behavior of secure and insecure programs is silently modified. As mentioned above, there are cases when the run at the top security level is immune to such modifications as it never gets dummy values. However, the behavior at less restrictive levels might be modified, leading to loss of important functionality. This directly connects to undiscovered attacks, addressed below.

Undiscovered attacks: Related to the silent failure point above, secure multi-execution "repairs" problematic executions on the fly, with no means to identify if there were any attempted attacks and what caused such attacks.

This motivates an enhancement of secure multi-execution that allows for detecting attacks. Intuitively, we introduce barrier synchronization of the runs at the different security levels and track the consistency of the values they produce. In the two-level lattice, we check if the low output produced by the low run matches the value produced by the high run (which is the same as the low output of the original program). If they are inconsistent, we have found an attack. Full transparency is the key for this result because it guarantees that secure programs must have exactly the same I/O behavior as their securely multi-executed versions.

Nondeterminism: Nondeterminism needs to be reproduced for the executions at different levels. Although this has not been explicitly handled in previous work, a natural possibility is to assign security level to the source of nondeterminism and propagate it to the relevant executions in a fashion similar to propagating inputs.

Dummy values: Dummy values are fed into executions that are not authorized to have access to sensitive input. An unfortunate choice of values might lead to the program crashing. Defensive programming is then needed to ensure that programs are stable under variation of allowed input.

Performance: Executing the program several times implies obvious performance overhead. At the same time, secure multi-execution benefits from multicore architectures, in particular when the number of executions is less than the number of cores [DP10]. Also, as we discuss in Section VII, optimizations are possible for simulating multiple executions by computing on enriched values [AF12].

III. Framework

We lay the foundation for our technical contributions outlined in the introduction by presenting a framework for information-flow security of interactive programs [OCC06], [CH08], [BPS+09], [RS11], [RHS12].

A. *Interactive programs*

Our model of computation is a *labeled transition system* (LTS). An LTS is a triple $(S, L, \rightarrow)$, where S is a set (of *states*), L is a set (of *labels*), and $\rightarrow \subseteq S \times L \times S$ (a *labeled transition relation*). Computation occurs in discrete steps (transitions), each taking a (unspecified) unit of time. $s \xrightarrow{l} s'$ iff $(s, l, s') \in \rightarrow$, and $s \xrightarrow{l}$ iff $s \xrightarrow{l} s'$ for some s'.

The systems we consider in this paper interact with their environment through channel-based message-passing. Such systems have three kinds of effects: (message-)input, output, and silence. The two latter effects are "productions", referred to as output o, and the first effect is a "consumption", referred to as input i. Collectively, these are actions a.

$$a ::= i \mid o \qquad i ::= c?v \qquad o ::= c!v \mid \bullet$$

Here, $c?v$ (resp. $c!v$) denotes a message received (resp. sent) on channel c carrying value v, and $\bullet$ denotes a non-interaction. c and v range over the (nonempty) sets $\mathbb{C}$ and $\mathbb{V}$ resp.. These effects are the *only* external interface to our systems; systems are "black boxes" in *every* other respect.

Definition III.1. An *input-output LTS* (LTS_{IO}) is an LTS $(S, L, \rightarrow)$, with L ranged by a.

Practical languages native to this paradigm include Erlang and JavaScript. Bohannon et al. give the semantics of a JavaScript-like language as an LTS_{IO} in [BPS+09] and Rafnsson et al. give the semantics of an imperative language with I/O (used in our examples) as an LTS_{IO} in [RHS12].

One element $\star \in \mathbb{V}$ (blank) is distinguished. When $s \xrightarrow{c?\star} s'$, then s has waited one time unit for an input on c without receiving one. $c!\star$ has no specific meaning.

Definition III.2. Let $\lambda = (S, L, \rightarrow)$ be an LTS_{IO}.

1) λ is *input-neutral* iff
   $\forall s \in S, c \,.\, (\exists v \,.\, s \xrightarrow{c?v}) \implies (\forall v \,.\, s \xrightarrow{c?v})$.
2) λ is *input-blocking* iff
   $\forall \{s, s'\} \subseteq S, c \,.\, s \xrightarrow{c?\star} s' \implies s' = s$.
3) λ is *deterministic* iff
   a) $\forall s \in S, a_1, a_2 \,.\, s \xrightarrow{a_1} \wedge s \xrightarrow{a_2} \wedge a_1 \neq a_2 \implies$
      $\exists c, v_1, v_2 \,.\, a_1 = c?v_1 \wedge a_2 = c?v_2$, and
   b) $\forall \{s, s_1, s_2\} \subseteq S, a \,.\, s \xrightarrow{a} s_1 \wedge s \xrightarrow{a} s_2 \implies s_1 = s_2$.

Pt. 1 states that if s is ready to perform input, s be receptive to any v. Pt. 2 states that input is a blocking operation. Pt. 3a says if $s \xrightarrow{c?v}$, then $s \xrightarrow{a}$ iff $a \in \{a?v \mid v \in \mathbb{V}\}$, and implicitly, if $s \xrightarrow{o}$, then $s \xrightarrow{a}$ iff $a = o$. Pt. 3b says s has no internal nondeterminism. Unless stated otherwise, any s we consider in this paper is from a λ satisfying pt. 1, 2 and 3. We discuss the assumption of pt. 2 further in Section IV.

B. *Traces*

A trace is a (finite) *list* of actions, denoted $\bar{a}$. We write $s \xrightarrow{\bar{a}} s_n$ if $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ for some $s_1, \ldots, s_n$ and $\bar{a} = a_1. \cdots .a_n$. Let $\bar{a}\restriction_?$, $\bar{a}\restriction_!$ and $\bar{a}\restriction_c$ denote the projection of $\bar{a}$ to its input-, output- and c-messages, respectively. E.g., if $\bar{a} = c?1.c'!2.c'?\star.c!4$, then $\bar{a}\restriction_? = c?1.c'?\star$, $\bar{a}\restriction_! = c'!2.c!4$, and $\bar{a}\restriction_c = c?1.c!4$. $\bar{a}\restriction_c$ extends to $\bar{a}\restriction_C$ for $C \subseteq \mathbb{C}$ in the obvious way. We write $\bar{a}\restriction_{x_1,\ldots,x_n}$ as short for $\bar{a}\restriction_{x_1} \cdots \restriction_{x_n}$; we refer to each x_j as a projection predicate. With $\bar{a}$ defined as above, $\bar{a}\restriction_{c,?} = \bar{a}\restriction_c\restriction_? = c?1$. We write $\bar{x} \leq \bar{x}''$ when, for some $\bar{x}'$, $\bar{x}'' = \bar{x}.\bar{x}'$. For a relation R, $\bar{a}\, R_{x_1,\ldots,x_n}\, \bar{a}'$ is short for $(\bar{a}\restriction_{x_1,\ldots,x_n})\, R\, (\bar{a}'\restriction_{x_1,\ldots,x_n})$. Note that $(R_{x_1,\ldots,x_n})_{x_0} = R_{x_0,\ldots,x_n}$. With $\bar{a}$ defined as above, $\bar{a} \leq_{!,c} c?3.c!4.c!5$.

C. *Observables*

The observables of our programs are its effects. The observability of a message is given by the *security level* associated with the channel carrying the message. As foreshadowed earlier, we assume a lattice $(\mathcal{L}, \sqsubseteq)$, with $\mathcal{L}$ ranged by ℓ, of security levels which express levels of *confidentiality*. Each channel is labeled with two security levels; $\delta(c)$ is the level of the *presence* of a message on c, and $\gamma(c)$ is the level of the *content* or *value* of a message on c. In examples, we frequently represent a channel by its security labels; we then write $\gamma(c)^{\delta(c)}$ in place of c (in code, $\gamma(c)\delta(c)$). A classic example is the two-level lattice $\mathcal{L}_{\texttt{LH}} = \{\texttt{L}, \texttt{H}\}$ with $\sqsubseteq = \{(\texttt{L}, \texttt{L}), (\texttt{L}, \texttt{H}), (\texttt{H}, \texttt{H})\}$, $\texttt{L}$ for "low" confidentiality, $\texttt{H}$ for "high". We let H, M, L denote $\texttt{H}^{\texttt{H}}$, $\texttt{H}^{\texttt{L}}$ and $\texttt{L}^{\texttt{L}}$, resp..

Figure 3 illustrates the flow of information in the case of the two-level lattice. Input on M is depicted in-between the high and low input. The presence of such an input is low (this dependency on low is illustrated by the dashed line) while the content is high (this dependency on high is illustrated by the solid line). Similarly, output on M is depicted in-between the high and low output. Its presence is observable at low level (cf. the dashed arrow), and its value is observable at high level (cf. solid arrow).

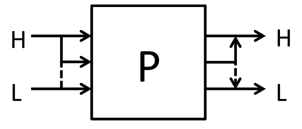

Figure 3: Original execution with fine-grained security

The security labels express *who* can observe *what*. An observer is associated a security level ℓ. An ℓ-observer is capable of observing the presence (resp. content) of a message on c iff $\delta(c) \sqsubseteq \ell$ (resp. $\gamma(c) \sqsubseteq \ell$). $\bar{a}\restriction_\ell$ removes ℓ-unobservable parts of actions in $\bar{a}$. For $\bar{a} = \bullet.\mathsf{L}?0.\mathsf{H}!1.\mathsf{M}?\star.\mathsf{M}!2.\mathsf{H}?\star.\bullet$, $\bar{a}\restriction_{\texttt{L}} = \bullet.\mathsf{L}?0.\bullet.\mathsf{M}?\mathtt{d}.\mathsf{M}!\star.\bullet.\bullet$. $\mathsf{H}!1$ got replaced with $\bullet$ since communication on H is unobservable to a $\texttt{L}$-observer (thus looks like a $\bullet$). $\mathsf{M}!2$ got replaced with $\mathsf{M}!\mathtt{d}$ (for a fixed $\mathtt{d} \in \mathbb{V}$), since a $\texttt{L}$-observer only observes presence of messages on M (all $a \in \{\mathsf{M}!v \mid v \in \mathbb{V}\}$ look the same).

Timing and progress: Eventually we enforce a property stating that variations in unobservable inputs to a system do not cause an ℓ-observable *difference* in the traces the system can perform. How we define trace equivalence defines the class of attackers such a property guarantees security against. We consider two classes of attackers: timing- and progress-sensitive ones respectively. To both, a blank input is unobservable since no message is passed. $\bar{a}\upharpoonright_\star$ replaces all $c?\star$ with $\bullet$. With $\bar{a}$ defined as above, $\bar{a}\upharpoonright_\star = \bullet.\mathsf{L}?0.\mathsf{H}!1.\bullet.\mathsf{M}!2.\bullet.\bullet$.

A *timing-sensitive* (e.g. [Aga00], [DP10]) attacker measures time between observables in a trace. $\bar{a}\upharpoonright_{\vec{\bullet}}$ removes trailing $\bullet$ from $\bar{a}$. E.g. $\bar{a}\upharpoonright_{\vec{\bullet}} = \bullet.\mathsf{L}?0.\mathsf{H}!1.\mathsf{M}?\star.\mathsf{M}!2.\mathsf{H}?\star$. Define *timing-sensitive ℓ-equivalence* $\simeq_\ell$ as $=_{\star,\ell,\vec{\bullet}}$. Consider

```
in H h ; out L 0
```

Since this program can perform traces $\bar{a}_1 = \mathsf{H}?\star.\mathsf{H}?1.\mathsf{L}!0$ and $\bar{a}_2 = \mathsf{H}?1.\mathsf{L}!0$, this program is not secure against a timing-sensitive $\mathtt{L}$-observer since $\bar{a}_1\upharpoonright_{\star,\mathtt{L},\vec{\bullet}} = \bullet.\bullet.\mathsf{L}!0 \neq \bullet.\mathsf{L}!0 = \bar{a}_2\upharpoonright_{\star,\mathtt{L},\vec{\bullet}}$ (L!0 is produced faster in the latter trace).

A *progress-sensitive* (e.g. [OCC06], [AS09], [RS11]) attacker observes whether there are more observables forthcoming in a trace. $\bar{a}\upharpoonright_\bullet$ removes all $\bullet$ from $\bar{a}$. With $\bar{a}$ as above, $\bar{a}\upharpoonright_\bullet = \mathsf{L}?0.\mathsf{H}!1.\mathsf{M}?\star.\mathsf{M}!2.\mathsf{H}?\star$. We define *progress-sensitive ℓ-equivalence* $\approx_\ell$ as $=_{\star,\ell,\bullet}$. $\bar{a}_1$ and $\bar{a}_2$ are not evidence that the above program is insecure against progress-sensitive ℓ-observers since $\bar{a}_1\upharpoonright_{\star,\mathtt{L},\bullet} = \mathsf{L}!0 = \bar{a}_2\upharpoonright_{\star,\mathtt{L},\bullet}$ (in both traces, L!0 eventually appears). Note that $(\simeq_\ell) \subsetneq (\approx_\ell)$. Thus, a progress-sensitive (timing-insensitive) attacker is strictly weaker than a timing-sensitive one.

D. Environments

The inputs to our systems come from the environment in which our systems run. Clark and Hunt [CH08] have demonstrated that when performing security analysis of programs, an environment does not need to be adaptive in any way to provoke a particular (leaking) behavior from a deterministic program. It is therefore sufficient *for our purposes* to consider environments represented as a *stream* (infinite list) of inputs for each input channel. So, an environment I is a mapping from input channels to the stream of inputs the environment provides on that channel. Since streams can contain blanks, our framework considers attacks powered by delayed input.

Input streams restrict which traces are possible; $\bar{a}$ is *consistent* with I, written $I \models \bar{a}$, iff for all c, with $I_c = I(c)$,

$$\forall \bar{a}' \leq \bar{a} \,.\, \exists \bar{i}' \leq I_c \,.\, \big(\quad |\bar{a}'| = |\bar{i}'| \ \wedge\ \bar{a}' \leq_{?,c,\star,\bullet} \bar{i}' \\ \wedge\ (\bar{a}' = \_.c?\star \implies \bar{i}' = \_.c?\star) \ \big).$$

Read this as "all i in $\bar{a}$ came from I, and $\star$ is read only when I had no value ready to be read" (_ is a wildcard). Running s under I constrains the traces which s can perform; s *performs $\bar{a}$ under I*, written $I \models s \xrightarrow{\bar{a}}$, iff $s \xrightarrow{\bar{a}}$ and $I \models \bar{a}$.

Consistency implies that I queues arriving input. Consider

```
in c x ; while |x| { x = |x| - 1 } ; in c y
```

Let s be the $\mathrm{LTS}_{\mathrm{IO}}$ state of this program and let $I_1, \ldots, I_5$ be given such that $I_{1c} = (c?\star)^\infty$, $I_{2c} = c?0.c?0.(c?\star)^\infty$, $I_{3c} = c?2.c?0.(c?\star)^\infty$, $I_{4c} = c?0.c?\star.c?\star.c?\star.c?0.(c?\star)^\infty$, $I_{5c} = c?2.c?\star.c?\star.c?\star.c?0.(c?\star)^\infty$. Then $I_1 \models s \xrightarrow{(c?\star)^n}$ for all $n \in \mathbb{N}$, $I_2 \models s \xrightarrow{c?0.\bullet.c?0}$, $I_3 \models s \xrightarrow{c?2.\bullet.\bullet.\bullet.\bullet.\bullet.c?0}$, $I_4 \models s \xrightarrow{c?0.\bullet.c?\star.c?\star.c?0}$, $I_5 \models s \xrightarrow{c?2.\bullet.\bullet.\bullet.\bullet.\bullet.c?0}$.

Stream equivalence becomes: $I_1 \simeq_\ell I_2$ (resp. $I_1 \approx_\ell I_2$) iff $\forall c \,.\, \delta(c) \sqsubseteq \ell \implies I_1(c) \simeq_\ell I_2(c)$ (resp. $I_1(c) \approx_\ell I_2(c)$).

Totality: An environment is total if it always provides a system with input whenever the system needs it. In our framework, I is total if $\star$ does not occur in any I_c. Previous work on security for interactive programs [OCC06], [CH08] assume that environments are total. However, as we have demonstrated previously [RHS12], this assumption limits (undesirably) the space of possible attacks on input-blocking interactive programs, since the presence of a message can vary depending on high data. Consider the program in Section III-C. Let $I_{1\mathsf{H}} = \mathsf{H}?0.(c?\star)^\infty$ and $I_{2\mathsf{H}} = I_{1c} = I_{2c} = (c?\star)^\infty$ for all $c \neq \mathsf{H}$. While this program can perform $\mathsf{H}?0.\mathsf{L}!0$ under I_1, the program cannot perform an $\approx_{\mathtt{L}}$-equivalent trace under I_2. Since a program can encode a bit in the presence of a message, these attacks becomes crucial in an interactive setting. To emphasise the gravity of the matter, here are three interactive programs [RHS12], each secure under total environments, which, when run in parallel, leak the input on H, bit by bit, on L, by encoding the received value in the presence of H_0 and H_1 messages.

```
while 1 { in H1 x ; out L 1 ; out H1' 42 }
```

```
while 1 { in H0 x ; out L 0 ; out H0' 42 }
```

```
in H h;
for b in bits(h) {
  if b { out H1 42 ; in H1' x }
  else { out H0 42 ; in H0' x }
}
```

In short, the lack (resp. delay) of input impedes on the progress (resp. timing) behavior of input-blocking interactive systems. To guarantee protection against attacks powered by varied input presence, nontotal environments (e.g. our I) must be considered. This in part motivates our fine-grained security types; since no low observables are allowed to occur after a high input, the only way for an input-blocking system to input a high value before performing low observables is if the presence level of the input is low [RHS12].

IV. Fine-grained secure multi-execution

The opening series of our contributions develops a generalization of SME [DP10] with respect to several dimensions. We lift the assumption on the totality of the input environment (that there is always assumed to be input) and on the cooperative scheduling. Furthermore, we distinguish between security levels of presence and content of messages. In addition, we generalize SME to arbitrary deterministic $\mathrm{LTS}_{\mathrm{IO}}$ and strengthen the guarantees SME provides.

By design, our formalization of SME ensures that the ℓ-observable part of the interaction on channels with ℓ presence depends only on ℓ-observable parts of input on channels with $\sqsubseteq \ell$ presence, thus enforcing a noninterference policy.

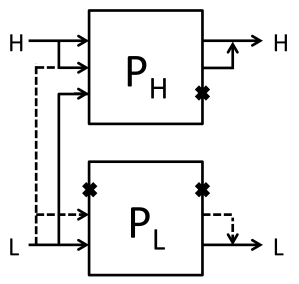

Figure 4: SME with fine-grained security

Figure 4 illustrates the intuition in our handling of the channels with fine-grained security levels for the two-level lattice. In addition to propagating low input to the high run (as in Figure 2), we propagate to the high run the fact that an M message has arrived to the low run. This allows consistent processing of the message. At the output, the presence of an M message is observable at the low level (cf. dashed output arrow). On the other hand, the value of an M output is produced by the high run (cf. solid output arrow).

Our SME of s runs, concurrently, a copy of s for each level in the security lattice. The SME of s can input on c *iff* the $\delta(c)$-run can input on c. An ℓ-run which can consume a c-input with $\delta(c) \not\sqsubseteq \ell$ is fed a (constant, pre-determined, input-independent) default value, denoted $\mathtt{d}$, by SME. An ℓ-run which can consume its nth c-input with $\delta(c) \sqsubset \ell$ gets a copy of the nth input consumed by the $\delta(c)$-run, unless $\delta(c)$ is yet to consume n c-inputs, in which case the ℓ-run blocks until the $\delta(c)$-run has done so. However, if $\gamma(c) \not\sqsubseteq \ell$, then the ℓ-run is fed $\mathtt{d}$ instead of the value in the nth c-input. The SME of s can output on c *iff* the $\delta(c)$-run can output on c. A c-output produced by a ℓ-run for which $\delta(c) \neq \ell$ is discarded by SME (as opposed to being sent to the environment). When an ℓ-run produces its nth c-output with $\ell = \delta(c)$, this output is sent straight to the environment, except when $\delta(c) \neq \gamma(c)$; in that case SME first checks whether the $\gamma(c)$-run has already produced its nth c-output. If so, then the value of the nth c-output produced by the SME of s becomes the value of the nth c-output produced by the $\gamma(c)$-run. Otherwise, the value becomes $\mathtt{d}$.

We now formalize SME for arbitrary s satisfying the assumptions in Definition III.2. Concurrent executions of s are scheduled by a *scheduler*.

Definition IV.1. A *scheduler* σ is a LTS with labels ranged by ℓ. σ is *deterministic* iff $\forall \ell, \ell' \,\boldsymbol{.}\, \sigma \xrightarrow{\ell} \wedge \sigma \xrightarrow{\ell'} \implies \ell = \ell'$. σ is *fair* iff $\forall \bar{\ell} \,\boldsymbol{.}\, \sigma \xrightarrow{\bar{\ell}} \implies \forall \ell \,\boldsymbol{.}\, \exists \bar{\ell}' \,\boldsymbol{.}\, \sigma \xrightarrow{\bar{\ell}.\bar{\ell}'.\ell}$.

Unless stated otherwise, σ is deterministic and fair. An example of deterministic and fair schedulers is the *round-robin schedulers*. For instance, for $\mathcal{L}_{\mathtt{LH}}$, a scheduler which infinitely repeats $\mathtt{H.L}$ or $\mathtt{L.H}$ is a deterministic fair scheduler.

The semantics of SME is given in Figure 5. A SME state is a triple $(\bar{a}, \sigma, S)$, where $\bar{a}$ is the list of actions which the SME has performed so far, σ the state of the scheduler, and S contains the state of the ℓ-runs. S maps each security level ℓ to a pair $(\bar{a}_\ell, s_\ell)$, where $\bar{a}_\ell$ is the list of actions which the ℓ-run has performed so far, and s_ℓ is the current state of the ℓ-run. For a given σ, the SME of s, $\mathsf{SME}(\sigma, s)$, is defined as $\mathsf{SME}(\sigma, s) = (\epsilon, \sigma, \lambda\ell \to (\epsilon, s)))$. The derivation of any SME state transition begins with the rule

$$\frac{\bar{a} \models (\sigma, S) \xrightarrow{a} (\sigma', S')}{(\bar{a}, \sigma, S) \xrightarrow{a} (\bar{a}.a, \sigma', S')}\text{log}$$

The purpose of (log) is to keep track of the interaction $\bar{a}$ which $\mathsf{SME}(\sigma, s)$ has had with the environment. Note that

$$\mathsf{SME}(\sigma, s) \xrightarrow{\bar{a}} (\bar{a}', \sigma', s') \iff \bar{a} = \bar{a}',$$

so we sometimes omit the trace label on a SME transition. We put $\bar{a}$ on the left side of "$\models$" in the next layer of the semantics, Figure 5b, to show that this layer only reads $\bar{a}$. The rules at this layer are mainly responsible for, by use of σ, deciding which ℓ-run takes a step next, using the rules in the third layer, Figure 5a. This layer is responsible for hiding from the previous layer all the different ways which an ℓ-run can take a step without interacting with the environment ((dead), (silence), (old-o), (old-i)), and signaling to the previous layer when the ℓ-run requires I/O with the environment to proceed ((new-o), (new-i)). Each rule at this level appends to the ℓ-run trace the action the ℓ-run performed during the step (not necessarily the same action as the one performed by the SME state). We equate a terminated ℓ-run with an infinitely silent one, as indicated by (dead) (not making terminated runs unschedulable excludes several timing attacks described by Kashyap et al. [KWH11]). SME stores output on channels with presence $\neq \ell$ without forwarding it to the environment, as per (old-o). (old-i) covers multiple scenarios for not inputting from the environment on c. When $\delta(c) \not\sqsubseteq \ell$, input $\mathtt{d}$. When $\delta(c) \sqsubseteq \ell$, this rule is only applicable if the ℓ-run has not already read all the c-inputs which the $\delta(c)$-run has read. When $\gamma(c) \sqsubseteq \ell$, input the same value received from the environment when the $\delta(c)$-run performed the corresp. input action. Otherwise, input $\mathtt{d}$ instead. (new-i) indicates that the ℓ-run requires input to proceed, and for the value v received from the previous layer, indicated on the transition label, instead feeds $\mathtt{d}$ to the ℓ-run iff $v \neq \star \wedge \gamma(c) \not\sqsubseteq \ell$. The previous layer has two rules for this scenario. When $\delta(c) \sqsubset \ell$, then the ℓ-run blocks until the $\delta(c)$-run reads on c, by (block-i). When $\delta(c)$ reads on c, the input is fed to every $\sqsupseteq \delta(c)$-run blocking on c, by (i). (new-o) notifies the previous layer that the ℓ-run has a fresh output for the environment. Rule (o) in the previous layer handles this scenario, checking if the $\gamma(c)$-run has already provided content for this output, and if so, replaces the value in the output with the value in the corresp. $\gamma(c)$-run c-output.

When $\gamma(c) \neq \delta(c)$, the output value is replaced by $\mathtt{d}$ iff $\gamma(c)$ has not yet produced the corresponding value. Having the $\delta(c)$-run instead wait for the $\gamma(c)$-run to reach the corresp. c-output, or giving responsibility of producing the c-output to the $\gamma(c)$-run, can introduce a leak:

```
in M h; l := 0; while l != h {l := l+1}; out M h
```

Here the time it takes for the $\mathtt{H}$-run to produce the M-output, and whether $\mathtt{H}$ produces the output at all, depends on $\mathtt{h}$.

$$\frac{s \not\rightarrow}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{\bullet} (\bar{a}_\ell.\bullet, s)}\text{dead} \qquad \frac{s \xrightarrow{c?v} s' \quad \bar{a}_\ell.c?v \leq_{\star,\ell,\bullet,?,c} \bar{a} \quad \textsf{\textbf{if}}\ \gamma(c) \not\sqsubseteq \ell\ \textsf{\textbf{then}}\ v = \mathtt{d}}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{\bullet} (\bar{a}_\ell.c?v, s')}\text{old-i}$$

$$\frac{s \xrightarrow{\bullet} s'}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{\bullet} (\bar{a}_\ell.\bullet, s')}\text{silence} \qquad \frac{s \xrightarrow{c!v_\ell} s' \quad \delta(c) = \ell \quad \textsf{\textbf{if}}\ \gamma(c) = \ell\ \textsf{\textbf{then}}\ v = v_\ell\ \textsf{\textbf{else}}\ v = \mathtt{d}}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{c!v} (\bar{a}_\ell.c!v_\ell, s')}\text{new-o}$$

$$\frac{s \xrightarrow{c!v} s' \quad \delta(c) \neq \ell}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{\bullet} (\bar{a}_\ell.c!v, s')}\text{old-o} \qquad \frac{s \xrightarrow{c?v_\ell} s' \quad \bar{a}_\ell =_{\star,\ell,\bullet,?,c} \bar{a} \quad \textsf{\textbf{if}}\ v = \star \vee \gamma(c) \sqsubseteq \ell\ \textsf{\textbf{then}}\ v_\ell = v\ \textsf{\textbf{else}}\ v_\ell = \mathtt{d}}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{c?v} (\bar{a}_\ell.c?v_\ell, s')}\text{new-i}$$

(a) SME ℓ-stepper

$$\frac{\sigma \xrightarrow{\ell} \sigma' \quad (\bar{a},\ell) \models S(\ell) \xrightarrow{\bullet} (\bar{a}_\ell, s)}{\bar{a} \models (\sigma, S) \xrightarrow{\bullet} (\sigma', S[\ell \mapsto (\bar{a}_\ell, s)])}\bullet \qquad \frac{\sigma \xrightarrow{\ell} \sigma' \quad (\bar{a},\ell) \models S(\ell) \xrightarrow{c?\star} (\bar{a}_\ell, s) \quad \delta(c) \sqsubset \ell}{\bar{a} \models (\sigma, S) \xrightarrow{\bullet} (\sigma', S[\ell \mapsto (\bar{a}_\ell, s)])}\text{block-i}$$

$$\frac{\sigma \xrightarrow{\ell} \sigma' \quad (\bar{a},\ell) \models S(\ell) \xrightarrow{c?v} \quad \delta(c) = \ell \quad S'(\ell') = \textsf{\textbf{if}}\ \ell \sqsubseteq \ell' \wedge (\bar{a},\ell') \models S(\ell') \xrightarrow{c?v} (\bar{a}', s')\ \textsf{\textbf{then}}\ (\bar{a}', s')\ \textsf{\textbf{else}}\ S(\ell')}{\bar{a} \models (\sigma, S) \xrightarrow{c?v} (\sigma', S')}\text{i}$$

$$\frac{\sigma \xrightarrow{\ell} \sigma' \quad (\bar{a},\ell) \models S(\ell) \xrightarrow{c!v_\ell} (\bar{a}_\ell, s) \quad S(\gamma(c)) = (\bar{a}_\gamma, \_) \quad \textsf{\textbf{if}}\ \exists \bar{a}', v_\gamma \,.\, \bar{a}.c!v_\ell =_{\star,\ell,\bullet,!,c} \bar{a}'.c!v_\gamma \leq_{!,c} \bar{a}_\gamma\ \textsf{\textbf{then}}\ v = v_\gamma\ \textsf{\textbf{else}}\ v = v_\ell}{\bar{a} \models (\sigma, S) \xrightarrow{c!v} (\sigma', S[\ell \mapsto (\bar{a}_\ell, s)])}\text{o}$$

(b) SME ℓ-chooser

Figure 5: Semantics of SME

While Figure 5b says that SME controls each step of each ℓ-run, in practice the responsibility of SME can be distributed to the ℓ-runs as follows. Each ℓ-run is made responsible for environment I/O on all $c \in \delta^{-1}(\ell)$, since $\mathsf{SME}(s)$ performs I/O iff the ℓ-run of s performs it. Each ℓ-run makes input on each $c \in \delta^{-1}(\ell)$ and output on each $c \in \gamma^{-1}(\ell)$ available in a shared resource (e.g. memory) s.t. $\sqsupset$ ℓ-runs can obtain a copy of the input when they need it, and an $\delta(c)$-run can obtain the actual value to output on c. Each ℓ-run processes (after sharing, when $\ell = \delta(c)$) $\mathtt{d}$ in place of the inputted value when $\delta(c) \sqsubseteq \ell \sqsubset \gamma(c)$, and outputs $\mathtt{d}$ when the $\gamma(c)$-run is yet to share the value to put into the output when $\delta(c) \sqsubset \gamma(c) = \ell$. This approach is taken in a SME benchmark by Devriese and Piessens [DP10]. Forcing ℓ-runs to diverge and recording full traces can be avoided [KWH11], [DP10]. This approach is sound as long as the ℓ-run threads cannot influence the scheduler.

While s is input-blocking, $\mathsf{SME}(s)$ is *not*; varied presence of input on $c \in \gamma^{-1}(\ell)$ cannot impede on progress or timing of ℓ'-runs where $\ell \not\sqsubseteq \ell'$. This effect is achieved by $c?\star$ actions; if $\mathsf{SME}(s)$ is in a state where an ℓ-run wants input on c, and I does not have one ready (yet), the ℓ-run can do a $c?\star$-action, allowing $\mathsf{SME}(s)$ to pass control to another ℓ'-run. In contrast, the *formalization* (as opposed to the benchmark implementation) of SME by Devriese and Piessens [DP10] is input blocking; if an ℓ-run is scheduled before a ℓ'-run with $\ell \not\sqsubseteq \ell'$, the nonpresence of input on $c \in \delta^{-1}(\ell)$ can interfere with the ℓ'-run. This hinders sound scheduling of runs for arbitrary nonlinear lattices.

A. Soundness

SME enforces the following property: Under observably equivalent environments, the respective sets of traces produced under any of them are observably equivalent.

Definition IV.2. s is timing-sensitive, progress sensitive noninterfering ($s \in \text{TSNI}$) iff $\forall \ell, I_1, I_2 \,.\, I_1 \simeq_\ell I_2 \implies$

$$\forall \bar{a}_1 \,.\, I_1 \models s \xrightarrow{\bar{a}_1} \implies$$
$$\exists \bar{a}_2 \,.\, I_2 \models s \xrightarrow{\bar{a}_2} \wedge\, \bar{a}_1 \simeq_\ell \bar{a}_2.$$

Theorem IV.3. $\forall \sigma, s \,.\, \mathsf{SME}(\sigma, s) \in \text{TSNI}$.

In contrast to Devriese and Piessens [DP10], who prove soundness for a cooperative scheduler for linear lattices, and to Kashyap et al. [KWH11], who prove soundness for two round-robin schedulers (the "Multiplex-2" and "Lattice-based" approaches), we prove a more general result: soundness for arbitrary deterministic and fair schedulers. While Devriese and Piessens claim their scheduler, called $\mathtt{select}_{\text{lowprio}}$, which executes the ℓ-runs to completion in increasing order by $\sqsubseteq$, works for any linearized lattice, Kashyap et al. have shown that $\mathtt{select}_{\text{lowprio}}$ introduces a timing dependency between ℓ-runs at incomparable levels in nonlinear lattices. For instance, with $\mathcal{L} = \mathcal{L}_{\mathtt{AB}} \overset{\text{def}}{=} \{\mathtt{H}, \mathtt{A}, \mathtt{B}, \mathtt{L}\}$ and $\sqsubseteq$ being the reflexive transitive closure of $\{(\mathtt{L}, \mathtt{A}), (\mathtt{L}, \mathtt{B}), (\mathtt{A}, \mathtt{H}), (\mathtt{B}, \mathtt{H})\}$, with linearization $\mathtt{L} \sqsubseteq \mathtt{A} \sqsubseteq \mathtt{B} \sqsubseteq \mathtt{H}$ and $\mathtt{d} = 0$, the time it takes for $\mathtt{B}^{\mathtt{B}}!1$ to occur is, under $\mathtt{select}_{\text{lowprio}}$, a function of the input on $\mathtt{A}^{\mathtt{A}}$.

```
in A^A a; while a != 0 { a := |a| - 1 }; out B^B 1
```

In the presence of nontotal environments, the situation is even worse; here the presence of input on $\mathtt{A}^{\mathtt{A}}$ leaks to $\mathtt{B}^{\mathtt{B}}$.

```
in A^A a; out B^B 1
```

While swapping $\mathtt{A}$ and $\mathtt{B}$ in the linearization resolves the issue in this program, the following program has no linearization of $\mathcal{L}$ for which $\mathtt{select}_{\text{lowprio}}$ schedules soundly.

```
in A^A a ; out B^B 1 ; in B^B b ; out A^A 1
```

We show in Appendix A that the assumption of termination is problematic when program input arrives arbitrarily in time.

The proof of Theorem IV.3 is a corollary of the following lemma, which can be seen by removing the last two elements in the conjunction in the conclusion of the lemma, and comparing the result with Definition IV.2. The details of this, and all other proofs, are in the full version of this paper [RS13]. We write $S_1 =_\ell S_2$ iff $\forall \ell' \sqsubseteq \ell \,.\, S_1(\ell') = S_2(\ell')$.

Lemma IV.4. $\forall s, \sigma, \ell, I_1, I_2 \,.\, I_1 =_\ell I_2 \implies$
$\forall \bar{a}_1, \sigma_1, S_1 \,.\, I_1 \models \mathsf{SME}(\sigma, s) \to (\bar{a}_1, \sigma_1, S_1) \implies$
$\exists \bar{a}_2, \sigma_2, S_2 \,.\, I_2 \models \mathsf{SME}(\sigma, s) \to (\bar{a}_2, \sigma_2, S_2) \;\wedge$
$\bar{a}_1 =_\ell \bar{a}_2 \;\wedge\; S_1 =_\ell S_2 \;\wedge \sigma_1 = \sigma_2$

Such a strong correspondence is achievable since, for each $\ell' \sqsubseteq \ell$, the ℓ'-run of s, in $I_1 \models \mathsf{SME}(\sigma, s)$ and $I_2 \models \mathsf{SME}(\sigma, s)$, behaves as $I\lceil_{\ell'} \models s$, where

$$(I\lceil_{\ell'})(c) = \begin{cases} (c?\mathtt{d})^\infty & \text{, if } \delta(c) \not\sqsubseteq \ell' \\ I(c)\lceil_\ell & \text{, otherwise.} \end{cases}$$

(While for $\delta(c) \sqsubseteq \ell'$, the number of $c?\star$ preceding a $c?v$ of an ℓ'-run in $I_j \models \mathsf{SME}(\sigma, s)$ compared to $I\lceil_{\ell'} \models s$ can differ due to σ, this number is the same in $I_1 \models \mathsf{SME}(\sigma, s)$ compared to $I_2 \models \mathsf{SME}(\sigma, s)$. Since s is input-blocking, all three are in the same state at the time of the non-blank read, and consume the same input, by $I\lceil_{\ell'}$). Thus, since $I_1 \models \mathsf{SME}(\sigma, s)$ and $I_2 \models \mathsf{SME}(\sigma, s)$ are both run under the same σ, after any number of transitions, the ℓ'-runs will in both runs have performed the same number of actions, consumed the same inputs, produced the same outputs, and be in the same state.

B. Transparency

We show that SME does not adversely modify the I/O behavior of a program for which changes in ℓ-unobservable input does not affect the ℓ-observable parts of the I/O behavior of the program. We obtain this class of programs by weakening TSNI to a timing-insensitive variant by replacing, in Definition IV.5, '$\simeq_\ell$' with '$\approx_\ell$'.

Definition IV.5. s is timing-insensitive, progress sensitive noninterfering ($s \in$ PSNI) iff $\forall \ell, I_1, I_2 \,.\, I_1 \approx_\ell I_2 \implies$
$\forall \bar{a}_1 \,.\, I_1 \models s \xrightarrow{\bar{a}_1} \implies$
$\exists \bar{a}_2 \,.\, I_2 \models s \xrightarrow{\bar{a}_2} \wedge \bar{a}_1 \approx_\ell \bar{a}_2$.

Let $s \in$ PSNI, ℓ, I and σ be arbitrary. In s and $\mathsf{SME}(\sigma, s)$, the interaction on ℓ-presence channels is ℓ-equivalent.

Theorem IV.6. $\forall s \in \text{PSNI}, I, \bar{a}.$

a) $I \models s \xrightarrow{\bar{a}} \implies$
$\exists \bar{a}' \,.\, I \models \mathsf{SME}(\sigma, s) \xrightarrow{\bar{a}'} \wedge \forall \ell \,.\, \bar{a} \leq_{\star, \ell, \delta^{-1}(\ell), \bullet} \bar{a}'$

b) $\forall \sigma \,.\, I \models \mathsf{SME}(\sigma, s) \xrightarrow{\bar{a}} \implies$
$\exists \bar{a}' \,.\, I \models s \xrightarrow{\bar{a}'} \wedge \forall \ell \,.\, \bar{a} \leq_{\star, \ell, \delta^{-1}(\ell), \bullet} \bar{a}'$

When all ℓ-presence outputs also have ℓ-content, the ℓ-presence interaction in s and $\mathsf{SME}(s)$ is the same. This is an improvement on Theorem 2 in [DP10] which establishes the *a)*-part of Theorem IV.6 for the interaction on each channel (as opposed to, for each ℓ, the interaction on all channels with presence level ℓ), and only for terminating runs of termination-sensitive s. Furthermore, under $\mathcal{L}_{\mathtt{AB}}$, $\mathtt{select}_{\text{lowprio}}$ yields a nontransparent run for the following program, as no $\mathtt{B}^\mathtt{B}!1$ occurs if no input on $\mathtt{A}^\mathtt{A}$ arrives.

```
out B^B 1; in A^A a
```

However, when s outputs on c with $\gamma(c) \sqsupset \delta(c)$, $\mathsf{SME}(\sigma, s)$ might replace the value in its corresponding output with $\mathtt{d}$. Thus, the timing behavior of $s \in$ PSNI can impede the ability of a σ to soundly schedule runs in $\mathsf{SME}(\sigma, s)$ such that the $\gamma(c)$-run reaches the output before the $\delta(c)$-run does irrespective of previously inputted values. In TSNI programs, however, all ℓ-runs for which $\delta(c) \sqsubseteq \ell$ will reach an output on c after the same number of reduction steps. This includes the $\gamma(c)$-run, since $\delta(c) \sqsubseteq \gamma(c)$. Thus, if we ensure that any ℓ-run never "outruns" its parent-runs, we can ensure that the content-provider of an output reaches the output before its presence-provider does. It is, however, not sufficient to require, for instance, that at any given point, $\mathtt{H}$ has been scheduled more often than $\mathtt{L}$, as the $\mathtt{H}$-run can waste its turns blocking on channels with presence level $\sqsubseteq \mathtt{H}$. The following predicate, when invoked as $\phi(\ell_\mathtt{H}, \ell_\mathtt{L}, 0, \bar{\ell})$, yields 1 when $\ell_\mathtt{H}$, a parent of $\ell_\mathtt{L}$, has been scheduled in this manner in $\bar{\ell}$, and 0 otherwise.

$$\begin{array}{lll} \phi(\_\,,\_\,,\_\,,\epsilon\,) & & = 1 \\ \phi(\ell_\mathtt{H}, \ell_\mathtt{L}, b_\text{Hseen}, \ell.\bar{\ell}) & \mid \ell = \ell_\mathtt{H} & = \phi(\ell_\mathtt{H}, \ell_\mathtt{L}, 1, \bar{\ell}) \\ & \mid \ell = \ell_\mathtt{L} \wedge b_\text{Hseen} & = \phi(\ell_\mathtt{H}, \ell_\mathtt{L}, 0, \bar{\ell}) \\ & \mid \ell = \ell_\mathtt{L} \wedge \neg b_\text{Hseen} & = 0 \\ & \mid \text{otherwise} & = \phi(\ell_\mathtt{H}, \ell_\mathtt{L}, b_\text{Hseen}, \bar{\ell}) \end{array}$$

Definition IV.7. σ is a *high-lead* scheduler ($\sigma \in$ *highlead*) if $\forall \bar{\ell} \,.\, \sigma \xrightarrow{\bar{\ell}} \implies \forall \ell_\mathtt{L}, \ell_\mathtt{H} \,.\, \ell_\mathtt{L} \sqsubset \ell_\mathtt{H} \implies \phi(\ell_\mathtt{H}, \ell_\mathtt{L}, 0, \bar{\ell})$.

An example of a high-lead scheduler is the round-robin scheduler which schedules levels in (increasing) order of maximal descendancy from the top element in the security lattice (ties broken arbitrarily). For instance, for $\mathcal{L} = \{\mathtt{H}, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{L}\}$ and $\sqsubseteq$ being the reflexive transitive closure of $\{(\mathtt{L}, \mathtt{C}), (\mathtt{L}, \mathtt{A}), (\mathtt{C}, \mathtt{B}), (\mathtt{A}, \mathtt{H}), (\mathtt{B}, \mathtt{H})\}$, σ which infinitely repeats $\mathtt{H.B.A.C.L}$ or $\mathtt{H.A.B.C.L}$ is a high-lead scheduler. It is for these schedulers that, in $s \in$ TSNI and $\mathsf{SME}(\sigma, s)$, the interaction on ℓ-presence channels is the same, for all ℓ.

Theorem IV.8. $\forall s \in \text{TSNI}, \sigma \in \textit{highlead}, I, \bar{a}.$

a) $I \models s \xrightarrow{\bar{a}} \implies$
$\exists \bar{a}' \,.\, I \models \mathsf{SME}(\sigma, s) \xrightarrow{\bar{a}'} \wedge \forall \ell \,.\, \bar{a} \leq_{\star, \delta^{-1}(\ell), \bullet} \bar{a}'$

b) $I \models \mathsf{SME}(\sigma, s) \xrightarrow{\bar{a}} \implies$
$\exists \bar{a}' \,.\, I \models s \xrightarrow{\bar{a}'} \wedge \forall \ell \,.\, \bar{a} \leq_{\star, \delta^{-1}(\ell), \bullet} \bar{a}'$

Thus, if SME puts a $\mathtt{d}$ in an M output when run using $\sigma \in$ *highlead*, then this must have been done to prevent a (timing or progress) leak – a desired effect.

V. Declassification

The challenge for declassification in SME is limited communication of information from the high to the low run. This is non-trivial as SME is originally designed to prevent any such leaks. This section demonstrates how information can be intentionally released in SME without violating the guarantees SME was originally designed to provide.

It turns out that the communication model from Section IV is an excellent fit for secure communication between the runs at different levels. A key desired property is to prevent the *occurrence* of declassification events from leaking information about the context while allowing intended release of the *value* to be declassified. This is a convenient match with our model that distinguishes the security levels of presence and content. The core idea is depicted in Figure 6. Declassification essentially corresponds to routing an M output from the high run into the low run.

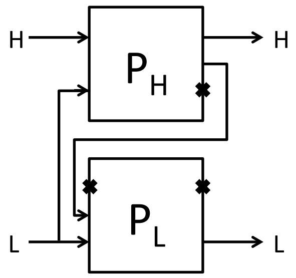

Figure 6: SME with declassification

A release policy ρ is a subset of $\sqsupset$. It indicates which information releases are allowed. When $\rho = \emptyset$, then ρ indicates a classical no-downward-flows policy, that is, one with no information release. When $(\ell, \ell') \in \rho$, then ρ permits the downward flow from ℓ to ℓ'. We write $\ell_0 \, \rho \, \ell$ iff

$$\exists \ell_0, \ldots, \ell_n \centerdot \ell_n \sqsubseteq \ell \wedge \forall 0 \leq i < n \centerdot (\ell_i, \ell_{i+1}) \in (\sqsubseteq \cup \, \rho).$$

Definition V.1. I_1 and I_2 are ρ-ℓ-equivalent ($I_1 \simeq_\ell^\rho I_2$) iff
$I_1 \simeq_\ell I_2 \wedge \forall \ell' \centerdot \ell' \, \rho \, \ell \implies I_1 \simeq_{\ell'} I_2$.

Definition V.2. s is ρ-releasing TSNI ($s \in \mathrm{TSNI}_\rho$) iff
$\forall \ell, I_1, I_2 \centerdot I_1 \simeq_\ell^\rho I_2 \implies$
$\forall \bar{a}_1 \centerdot I_1 \models s \xrightarrow{\bar{a}_1} \implies$
$\exists \bar{a}_2 \centerdot I_2 \models s \xrightarrow{\bar{a}_2} \wedge \bar{a}_1 \simeq_\ell \bar{a}_2$.

This noninterference policy only states *what* can be released, without constraining *where* during control flow information release is permitted. A common construct for aiding programmers in specifying information release is `declassify`(e, ℓ), which declassifies the value of expression e (in the state in which this command is executed) to level ℓ. For instance, only `b` is intended to leak to `L` in the below program p_{d1} under lattice $\mathcal{L}_{\texttt{AB}}$ with $\rho = \{(\texttt{B}, \texttt{L})\}$, but it turns out `a` leaks as well; a TSNI$_\rho$-insecurity.

```
in A^L a; in B^L b ;                         // p_d1
if a { l := declassify(b, L) }
out L l
```

The only interface SME has to its ℓ-runs are action labels. To transfer the right value from the `H`-run to be declassified by the `L`-run, we need to enable the environment (in our case SME) to obtain the value which an s wishes to declassify from an action label, and (optionally) subsequently replace it. To achieve this, we introduce *release channels*. Let $\mathbb{R} \subseteq \mathbb{C}$ be the release channels, ranged by r. Let $\mathbf{C} = \mathbb{C} \setminus \mathbb{R}$ be the communication channels, ranged by $\mathbf{c}$. Let $\varrho : \mathcal{L}^2 \to \mathbb{R}$ be bijective. ϱ associates a release channel $\varrho(\ell, \ell')$ with each kind of information release (ℓ, ℓ') (not all of which are permitted by ρ). The following inference rule illustrates how the semantics of declassification, in a simple imperative language with I/O, can be given such that the environment can (optionally) have a program declassify a completely different value (through "`in` r") than the value the program announced it would declassify (through "`out` r v"). This declassify construct is more fine-grained than the standard one as it specifies both the *from* level ℓ and the *to* level ℓ' of the declassification operation.

$$\frac{m \models e = v \quad \varrho(\ell, \ell') = r \quad (m, \texttt{out}\ r\ v) \xrightarrow{r!v} (m', \texttt{skip})}{(m, x := \texttt{declassify}(e, \ell \to \ell')) \xrightarrow{r!v} (m', x := \texttt{in}\ r)}$$

To restore the typical semantics of declassification (which does not make declassification an effect) in an s, we simply place s in a wrapper which makes communication on release channels a feedback, as per $\mathsf{F}(s)$, given below. $\mathsf{F}(s)$ only interacts with its environment on communication channels.

$$\frac{s \xrightarrow{r!v} s'}{\mathsf{F}(s) \xrightarrow{\bullet} \mathsf{F}(r?v, s')} \quad \frac{s \xrightarrow{r?v} s'}{\mathsf{F}(r?v, s) \xrightarrow{\bullet} \mathsf{F}(s')} \quad \frac{s \xrightarrow{a} s' \quad \nexists r, v \centerdot a = r!v}{\mathsf{F}(s) \xrightarrow{a} \mathsf{F}(s')}$$

This wrapper only has the desired effect if s communicates on release channels in the expected manner, by always first performing an output on r, and subsequently performing an input on r. We define the class of such s now.

Definition V.3. An LTS$_{\mathrm{IO}}$ $\lambda = (S, L, \to)$ is an LTS$_{\mathrm{IO}}$ with release (LTS$_{\mathrm{IO}}^{\mathbb{R}}$) iff

$$\forall s \in S, r, v, \bar{a} \centerdot \; (s \xrightarrow{\bar{a}.r!v} \implies \exists v' \centerdot s \xrightarrow{\bar{a}.r!v.r?v'})$$
$$\wedge (s \xrightarrow{\bar{a}.r?v} \implies \exists v', \bar{a}' \centerdot \bar{a} = \bar{a}'.r!v')$$

λ is an LTS$_{\mathrm{IO}}$ without release (LTS$_{\mathrm{IO}}^{\mathbf{C}}$) iff
$\forall s \in S \centerdot \nexists r, \bar{a}, v \centerdot s \xrightarrow{\bar{a}.r!v} \vee s \xrightarrow{\bar{a}.r?v}$.
We write $s \in$ LTS$_{\mathrm{IO}}^{\mathbb{R}}$ (resp. $s \in$ LTS$_{\mathrm{IO}}^{\mathbb{R}}$) iff s is a state in some LTS$_{\mathrm{IO}}^{\mathbb{R}}$ (resp. LTS$_{\mathrm{IO}}^{\mathbb{R}}$).

We consider only LTS$_{\mathrm{IO}}^{\mathbb{R}}$ and LTS$_{\mathrm{IO}}^{\mathbf{C}}$ s for the remainder of this section; an s which is neither interacts on some r in an undesired way. We require that $\forall r \centerdot \delta(r) = \gamma(r) \wedge \exists \ell' \centerdot \varrho(\gamma(r), \ell') = r$. Read this as "$r$ releases information from $\gamma(r)$ to ℓ'." The information release which r is responsible for is permitted by the release policy, iff, $(\gamma(r), \ell') \in \rho$.

The semantics of SME extended with release channels (SME$_{\mathbb{R}}$) is given in Figure 7. It is parameterized by the release policy ρ. By (r-not), any release action which is not permitted by ρ is a feedback. By (r-d), when a target ℓ-run of a release reaches a release before the source $\gamma(r)$-run does, `d` is released instead (same justification as for the treatment of M-output in SME). By (r), if a valid release receiver

$$\dfrac{s \xrightarrow{r!v} s'}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{\bullet} (\bar{a}_\ell.r!v, s)}\,\text{r-o} \qquad \dfrac{s \xrightarrow{r?v} s'}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{r?v} (\bar{a}_\ell.r?v, s)}\,\text{r-i}$$

(a) $\mathsf{SME}_{\mathbb{R}}$ ℓ-stepper; add Figure 5a rules w/ occurrences of c replaced with $\mathbf{c}$.

$$\dfrac{\sigma \xrightarrow{\ell} \sigma' \quad (\bar{a},\ell) \models S(\ell) \xrightarrow{r?v} (\bar{a}_\ell.r!v.r?v, s) \quad \neg\mathsf{releaseok}(r,\ell)}{\bar{a} \models (\sigma, S) \xrightarrow{\bullet} (\sigma', S[\ell \mapsto (\bar{a}_\ell.r!v.r?v, s)])}\,\text{r-not} \qquad \mathsf{releaseok}(r,\ell) = (\varrho^{-1}(r) = (\ell', \_) \in \rho \wedge \ell' \,\rho\, \ell \wedge \ell' \not\sqsubseteq \ell).$$

$$\dfrac{\sigma \xrightarrow{\ell} \sigma' \quad (\bar{a},\ell) \models S(\ell) \xrightarrow{r?\mathtt{d}} (\bar{a}_\ell.r?\mathtt{d}, s) \quad \mathsf{releaseok}(r,\ell) \quad S(\gamma(r)) = (\bar{a}_\gamma, \_) \quad |\bar{a}_\gamma{\restriction_{!,r,\bullet}}| < |\bar{a}_\ell{\restriction_{!,r,\bullet}}|}{\bar{a} \models (\sigma, S) \xrightarrow{\bullet} (\sigma', S[\ell \mapsto (\bar{a}_\ell.r?\mathtt{d}, s)])}\,\text{r-d}$$

$$\dfrac{\begin{array}{c}\sigma \xrightarrow{\ell} \sigma' \quad (\bar{a},\ell) \models S(\ell) \xrightarrow{r?v} (\bar{a}_\ell.r!v_\ell.r?v, s) \quad \mathsf{releaseok}(r,\ell) \quad S(\gamma(r)) = (\bar{a}_\gamma.r!v_\gamma.\_\_, \_) \quad |\bar{a}_\gamma{\restriction_{!,r,\bullet}}| = |\bar{a}_\ell{\restriction_{!,r,\bullet}}| \\ \textbf{if } (\ \exists \ell_{\mathtt{d}}, \bar{a}_{\mathtt{d}} \,.\, S(\ell_{\mathtt{d}}) = (\bar{a}_{\mathtt{d}}.r!\_.r?\mathtt{d}.\_\_, \_) \wedge |\bar{a}_\gamma{\restriction_{!,r,\bullet}}| = |\bar{a}_{\mathtt{d}}{\restriction_{!,r,\bullet}}| \wedge \mathsf{releaseok}(r, \ell_{\mathtt{d}})\ ) \textbf{ then } v = \mathtt{d} \textbf{ else } v = v_\gamma\end{array}}{\bar{a} \models (\sigma, S) \xrightarrow{\bullet} (\sigma', S[\ell \mapsto (\bar{a}_\ell.r!v_\ell.r?v, s)])}\,\text{r}$$

(b) $\mathsf{SME}_{\mathbb{R}}$ ℓ-chooser; add Figure 5b rules w/ occurrences of c replaced with $\mathbf{c}$.

Figure 7: Semantics of $\mathsf{SME}_{\mathbb{R}}$

has already received $\mathtt{d}$ for this release, so does the ℓ-run. Otherwise $\gamma(r)$ has already done the release, so the ℓ-run receives the right value.

A. Soundness

$\mathsf{SME}_{\mathbb{R}}$ is sound with regards to any release policy.

Theorem V.4. $\forall s \in \mathrm{LTS}^{\mathbb{R}}_{\mathrm{IO}}, \sigma, \rho \,.\, \mathsf{SME}_{\mathbb{R}}(\sigma, s, \rho) \in \mathrm{TSNI}_\rho$.

The proof of this theorem follows a similar pattern as the proof of Theorem IV.3, utilizing a lemma which is near-identical to Lemma IV.4.

$\mathsf{SME}_{\mathbb{R}}$ does not introduce an information release into a program which does not already release information. The following statements correspond to the *conservativeness* principle of declassification [SS09] that stipulates that the security condition for systems with no information release is equivalent to baseline noninterference.

Theorem V.5. $\forall s \in \mathrm{LTS}^{\mathbb{R}}_{\mathrm{IO}} \,.\, s \in \mathrm{TSNI} \implies \forall \rho, \sigma \,.\, \mathsf{SME}_{\mathbb{R}}(\sigma, s, \rho) \in \mathrm{TSNI}$.

When no information release is permitted, Definitions V.2 and IV.2 coincide ($I_1 \simeq^{\emptyset}_{\ell} I_2$ becomes $I_1 \simeq_\ell I_2$).

Corollary V.6. $\mathrm{TSNI}_\emptyset = \mathrm{TSNI}$.

$\mathsf{SME}_{\mathbb{R}}$ prevents all downward flows in all s which do not announce information release through release actions. This is a corollary of Theorem IV.3, since no step of in a trace from s is derived using a rule from Figure 7.

Corollary V.7. $\forall s \in \mathrm{LTS}^{\mathbf{C}}_{\mathrm{IO}}, \sigma, \rho \,.\, \mathsf{SME}_{\mathbb{R}}(\rho, \sigma, s) \in \mathrm{TSNI}$.

B. Transparency

Information release can impede on transparency, even in secure programs. By Corollary V.7, when s releases information w/o announcing the release on a release channel, the corresponding control in $\mathsf{SME}_{\mathbb{R}}(\rho, s)$ never receives the declassified value. With $\rho = \{(\mathtt{H}, \mathtt{L})\}$, consider this s.

```
in M h; out L h
```

We have $s \in \mathrm{TSNI}_\rho$. However, the $\mathtt{L}$-run in $\mathsf{SME}_{\mathbb{R}}(\rho, s)$ never gets the $\mathtt{H}$-value in the $\mathtt{H}^{\mathtt{L}}$-input, and thus, $\mathsf{SME}_{\mathbb{R}}(\rho, s)$ cannot be transparent. A similar problem arises when the $\mathtt{L}$-run reaches an information release from $\mathtt{H}$ before the $\mathtt{H}$-run does; then the $\mathtt{L}$-run instead receives $\mathtt{d}$. This occurs in the following program p_{d2} if $\mathtt{L}$ is scheduled too often before $\mathtt{H}$.

```
in M h ;                                   // p_d2
l := declassify(h, H->L) ;
out L l
```

It turns out that these are the only inhibitors for transparency. We define the class of programs which only release information through release channels. The definition makes use of a wrapper which binds release channels internally.

$$\dfrac{s \xrightarrow{r!v} s'}{\mathsf{B}(\bar{a}, I, s) \xrightarrow{\bullet} \mathsf{B}(\bar{a}.r!v, I, s')} \qquad \dfrac{s \xrightarrow{r?v} s' \quad I \models \bar{a}.r?v}{\mathsf{B}(\bar{a}, I, s) \xrightarrow{\bullet} \mathsf{B}(\bar{a}.r?v, I, s')}$$

$$\dfrac{s \xrightarrow{a} s' \quad \nexists r, v \,.\, a = r?v}{\mathsf{B}(\bar{a}, I, s) \xrightarrow{a} \mathsf{B}(\bar{a}.a, I, s')} \qquad \mathsf{B}(I, s) = \mathsf{B}(\epsilon, I, s)$$

Definition V.8. s is TSNI modulo release ($s \in \mathrm{TSNI}_{\mathsf{mod}\,\mathbb{R}}$) iff $\forall I \,.\, (\forall r \,.\, \nexists i \,.\, \bar{i}.r?\star \leq I_r) \implies \mathsf{B}(I, s) \in \mathrm{TSNI}$.

Let $\rho(s) = \{\varrho^{-1}(r) \mid \exists \bar{a}, v \,.\, s \xrightarrow{r?v.\bar{a}} \vee s \xrightarrow{r!v.\bar{a}}\}$ be the releases of s. If $\rho(s) \subseteq \rho$, then $\mathsf{SME}_{\mathbb{R}}$ will not prevent any declassifications; this enables transparency.

Theorem V.9. $\forall s \in (\mathrm{LTS}^{\mathbb{R}}_{\mathrm{IO}} \cap \mathrm{TSNI}_{\mathsf{mod}\,\mathbb{R}}), \sigma \in \mathit{highlead}, I, \bar{a} \,.$

a) $I \models \mathsf{F}(s) \xrightarrow{\bar{a}} \implies$
$\exists \bar{a}' \,.\, I \models \mathsf{SME}_{\mathbb{R}}(\rho(s), \sigma, s) \xrightarrow{\bar{a}'} \wedge \forall \ell \,.\, \bar{a} \leq_{\star, \delta^{-1}(\ell), \bullet} \bar{a}'$

b) $I \models \mathsf{SME}_{\mathbb{R}}(\rho(s), \sigma, s) \xrightarrow{\bar{a}} \implies$
$\exists \bar{a}' \,.\, I \models \mathsf{F}(s) \xrightarrow{\bar{a}'} \wedge \forall \ell \,.\, \bar{a} \leq_{\star, \delta^{-1}(\ell), \bullet} \bar{a}'$

Program p_{d2} satisfies $\mathrm{TSNI}_{\mathsf{mod}\,\mathbb{R}}$. It is easy to see that if $\sigma \in \mathit{highlead}$ and $\rho = \{(\mathtt{H}, \mathtt{L})\}$, then $\mathsf{SME}_{\mathbb{R}}(\rho, \sigma, s)$ routes the value announced by the $\mathtt{H}$-run (which is the value received on M) to the $\mathtt{L}$-run in the desired way, thus yielding a transparent run. When $\rho = \emptyset$, $\mathsf{SME}_{\mathbb{R}}(\rho, \sigma, s)$ stops the forbidden declassification, retaining soundness. In program p_{d1}, the $\mathsf{SME}_{\mathbb{R}}(\rho, \sigma, s)$ allows the declassification but prevents the implicit flow of a at the same time! This is a fruitful byproduct of the *separation* of computation into ℓ-runs; the B-run never obtains A-information, and thus cannot leak it (not even implicitly). At last, $\mathsf{SME}_{\mathbb{R}}(\rho, \sigma, s)$, with $\rho = \{(\mathtt{H}, \mathtt{L})\}$, allows the announced declassification, but stops the explicit flow, in the following program. This indicates that $\mathsf{SME}_{\mathbb{R}}$ not only enforces *what* is released, but also *where* in the program release takes place. We discuss this further in Section VIII.

```
in M h1 ; in M h2 ;
l1 := declassify(h1, H->L) ;
l2 := h2 ;
out L l1 ; out L l2
```

VI. Full transparency

This section shows how to achieve fully transparency for secure multi-execution by barrier synchronization. Full transparency, in contrast to per-channel or per-level transparency, guarantees that our SME enforcement preserves the I/O behavior of secure programs, including the ordering of I/O messages. Thanks to such a strong property, we are able to deploy SME to detect attacks.

The core idea is pictorially summarized in Figure 8. In contrast to Figure 2, we are not ignoring the low output produced by the high run. Instead, we match it with the low output produced by the low run. If the program is secure, this approach guarantees that there may not be any deviation in this matching. Thus, if there is a deviation, it must be due to the insecurity of the original program. From this deviation, we can construct a counterexample against noninterference.

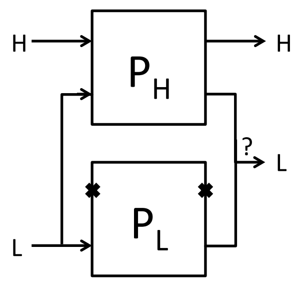

Figure 8: SME by barrier synchronization

We formalize our approach for a two-level lattice in Figure 9. While nothing inhibits the H-run from performing non-L-presence actions (by (H-a) and (●)), a barrier forms when the H-run reaches a L-presence action (by (H-block)). The H-run then only proceeds once the L-run reaches a L-presence action. When the L-run reaches a L-presence action, and the H-run is yet to perform the corresp. action, a barrier forms. The L-run then only proceeds once the H-run has done one of two things. 1) reached an L-observable action before advancing t steps beyond the L-run (by (L-a)), or 2) not (by (L-timeout)). In 1), if the H-run reached a L-observably different action, we note an attack in $\bar{\alpha}_1$. In 2), we note a timeout in $\bar{\alpha}_2$ (which might be an attack). Input streams are constructed from traces using ι, defined as

$$(\iota(\bar{a}))_c = (\bar{a}{\upharpoonright_{\star,?,c,\bullet}}).(c?\star)^{\infty}.$$

Definition VI.1. An *attack* α is a 4-tuple $(\ell, I_1, I_2, \bar{a}_1)$ where $I_1 =_\ell I_2$ and $I_1 \models \bar{a}_1$. α is an *attack on* s iff

1) $I_1 \models s \xrightarrow{\bar{a}_1}$, and
2) $\forall \bar{a}_2 \,\textbf{.}\, I_2 \models s \xrightarrow{\bar{a}_2} \implies \bar{a}_2 \not\approx_\ell \bar{a}_1$.

A. Soundness

$\mathsf{SME}_{\mathbb{T}}$ enforces a timing-insensitive noninterference notion. This is easily seen by observing that the traces produced by $I \models \mathsf{SME}_{\mathbb{T}}(s)$ and $I{\upharpoonright_{\mathtt{L}}} \models s$ are $\approx_{\mathtt{L}}$-equivalent.

Theorem VI.2. $\forall s, \sigma \,\textbf{.}\, \mathsf{SME}_{\mathbb{T}}(\sigma, s) \in \mathrm{PSNI}$.

By allowing the L-run to wait up to t steps for the H-run to match an L-observable, $\mathsf{SME}_{\mathbb{T}}(s)$ introduces a timing leak into s, and thus does not enforce TSNI. We note however that $\mathsf{SME}_{\mathbb{T}}(\sigma, s)$ can be wrapped in a black-box timing leak mitigator to alleviate this weakening of the soundness guarantee [AZM10].

At the point where the H-run deviates from the L-run, the H-run is "frozen" (to avoid leaks), becoming semantically equivalent to a program producing ● infinitely, by (conflict). A more practical approach would be to instead have the H-run behave like it would under $\mathsf{SME}(s)$ henceforth. We hypothesize (but do not prove) that this modification of $\mathsf{SME}_{\mathbb{T}}$ yields a sound enforcement.

B. Transparency

Modulo ●, $\mathsf{SME}_{\mathbb{T}}(s)$ and s produce the same sequence of I/O (up to an attack or timeout in $\mathsf{SME}_{\mathbb{T}}(s)$). In contrast to e.g. Devriese and Piessens [DP10], who swap the order of outputs in the following two programs (linearization $\mathtt{B} \sqsubseteq \mathtt{A}$),

```
out H 1 ; out L 1
```

```
out A^A 1 ; out B^B 1
```

the I/O correspondence is full. Furthermore, this result guarantees transparency even when s is insecure.

Theorem VI.3. $\forall s, \sigma, I, \bar{a} \textbf{.}$

a) $I \models s \xrightarrow{\bar{a}} \implies$
$\nexists \bar{a}' \,\textbf{.}\, I \models \mathsf{SME}_{\mathbb{T}}(\sigma, s) \xrightarrow{\bar{a}'} (\_, \_, \epsilon, \epsilon)$
$\quad \wedge \; |\bar{a}'{\upharpoonright_{\star,\bullet}}| \leq |\bar{a}'{\upharpoonright_{\star,\bullet}}| \; \wedge \; \bar{a}' \not\leq_{\star,\bullet} \bar{a}$

b) $I \models \mathsf{SME}_{\mathbb{T}}(\sigma, s) \xrightarrow{\bar{a}} (\_, \_, \epsilon, \epsilon) \implies$
$\exists \bar{a}' \,\textbf{.}\, I \models s \xrightarrow{\bar{a}'} \wedge \; \bar{a} \approx \bar{a}'$

It is easy to see that if $s \in \mathrm{TSNI}$, then $\mathsf{SME}_{\mathbb{T}}$ never generates attacks, and thus, s and $\mathsf{SME}_{\mathbb{T}}(\sigma, s)$ have the same (that is, $\approx_{\mathtt{L}}$-equivalent) I/O behavior.

C. Attacks

Any match deviation found by $\mathsf{SME}_{\mathbb{T}}(s)$ (before a timeout), forms the basis of a concrete proof that $s \notin \mathrm{PSNI}$.

Theorem VI.4. $\forall s, \sigma, I \textbf{.} I \models \mathsf{SME}_{\mathbb{T}}(\sigma, s) \to (\_, \_, \alpha, \epsilon) \implies$ *α is an* $\mathtt{L}$*-attack on* s.

$$\dfrac{s \xrightarrow{a} s' \qquad \bar{a}_\ell.a \leq_{\star,\ell,\bullet} \bar{a} \\ \textbf{if } a = c?v \textbf{ then if } \gamma(c) \not\sqsubseteq \ell \textbf{ then } v = \mathtt{d} \textbf{ else } v \neq \star}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{\bullet} (\bar{a}_\ell.a, s')}\text{old} \qquad \dfrac{\bar{a}'.a' \leq \bar{a} \quad \delta(a') = \mathtt{L} \quad \bar{a}' \approx_{\mathtt{L}} \bar{a}_{\mathtt{H}} \quad \forall a \,.\, s \xrightarrow{a} \implies a \neq_{\mathtt{L}} a'}{(\bar{a},\mathtt{H}) \models (\bar{a}_{\mathtt{H}}, s) \xrightarrow{\bullet} (\bar{a}_{\mathtt{H}}.\bullet, s')}\text{conflict}$$

$$\dfrac{s \xrightarrow{c!v_\ell} s' \quad \bar{a}_\ell =_{\star,\ell,\bullet} \bar{a} \quad \delta(c) \sqsubseteq \ell \\ \textbf{if } \gamma(c) \sqsubseteq \ell \textbf{ then } v = v_\ell \textbf{ else } v = \mathtt{d}}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{c!v} (\bar{a}_\ell.c!v_\ell, s')}\text{new-o} \qquad \dfrac{s \xrightarrow{c?v_\ell} s' \quad \bar{a}_\ell =_{\star,\ell,\bullet} \bar{a} \quad \delta(c) \sqsubseteq \ell \\ \textbf{if } \gamma(c) \sqsubseteq \ell \vee v = \star \textbf{ then } v_\ell = v \textbf{ else } v_\ell = \mathtt{d}}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{c?v} (\bar{a}_\ell.c?v_\ell, s')}\text{new-i}$$

(a) SME$_{\mathbb{T}}$ ℓ-stepper; add (dead) and (silence) from Figure 5a

$$\dfrac{\sigma \xrightarrow{\mathtt{H}} \sigma' \quad (\bar{a},\mathtt{H}) \models S(\mathtt{H}) \xrightarrow{a} (\bar{a}_{\mathtt{H}}, s_{\mathtt{H}}) \quad \delta(a) = \mathtt{H}}{\bar{a} \models (\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2) \xrightarrow{a} (\sigma', S[\ell \mapsto (\bar{a}_{\mathtt{H}}, s_{\mathtt{H}})], \bar{\alpha}_1, \bar{\alpha}_2)}\mathtt{H}\text{-a}$$

($\bullet$) as in Figure 5b (with $\bar{\alpha}_1$, $\bar{\alpha}_2$ equal in states before & after $\xrightarrow{\bullet}$).

$$\dfrac{\sigma \xrightarrow{\mathtt{H}} \sigma' \quad (\bar{a},\mathtt{H}) \models S(\mathtt{H}) \xrightarrow{a} \quad \delta(a) = \mathtt{L}}{\bar{a} \models (\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2) \xrightarrow{a} (\sigma', S, \bar{\alpha}_1, \bar{\alpha}_2)}\mathtt{H}\text{-block}$$

$$\dfrac{\sigma \xrightarrow{\mathtt{L}} \sigma' \quad (\bar{a},\mathtt{L}) \models S(\mathtt{L}) \xrightarrow{a_{\mathtt{L}}} (\bar{a}_{\mathtt{L}}, \_) \quad \delta(a_{\mathtt{L}}) = \mathtt{L} \quad S(\mathtt{H}) = (\bar{a}_{\mathtt{H}}, \_) \quad |\bar{a}_{\mathtt{L}}| + t \geq |\bar{a}_{\mathtt{H}}| \quad (\bar{a},\mathtt{H}) \models S(\mathtt{H}) \xrightarrow{a_{\mathtt{H}}} \quad \delta(a_{\mathtt{H}}) \neq \mathtt{L}}{\bar{a} \models (\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2) \xrightarrow{\bullet} (\sigma', S, \bar{\alpha}_1, \bar{\alpha}_2)}\text{L-wait}$$

$$\dfrac{\begin{array}{c}\sigma \xrightarrow{\mathtt{L}} \sigma' \quad (\bar{a},\mathtt{L}) \models S(\mathtt{L}) \xrightarrow{a_{\mathtt{L}}} (\bar{a}_{\mathtt{L}}, s_{\mathtt{L}}) \quad \delta(a_{\mathtt{L}}) = \mathtt{L} \quad S(\mathtt{H}) = (\bar{a}_{\mathtt{H}}, \_) \quad |\bar{a}_{\mathtt{L}}| + t \geq |\bar{a}_{\mathtt{H}}| \quad (\bar{a},\mathtt{H}) \models S(\mathtt{H}) \xrightarrow{a_{\mathtt{H}}} \quad \delta(a_{\mathtt{H}}) = \mathtt{L} \\ \begin{array}{lll}\textbf{if} & (\bar{a},\mathtt{H}) \models S(\mathtt{H}) \xrightarrow{a_{\mathtt{L}}} & \textbf{then } a = a_{\mathtt{L}} \wedge \bar{\alpha}'_1 = \bar{\alpha} \\ \textbf{else if} & a_{\mathtt{L}} =_{\mathtt{L}} a_{\mathtt{H}} & \textbf{then } a = a_{\mathtt{H}} \wedge \bar{\alpha}'_1 = \bar{\alpha} \\ \textbf{else} & & \phantom{\textbf{then }} a = a_{\mathtt{L}} \wedge \bar{\alpha}'_1 = \bar{\alpha}_1.(\mathtt{L}, \iota(\bar{a}.i_{\mathtt{L}}), \iota(\bar{a}_{\mathtt{L}}), \bar{a}_{\mathtt{L}})\end{array}\end{array}}{\bar{a} \models (\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2) \xrightarrow{a} (\sigma', S[\mathtt{L} \mapsto (\bar{a}_{\mathtt{L}}, s_{\mathtt{L}})], \bar{\alpha}'_1, \bar{\alpha}_2)}\text{L-a}$$

$$\dfrac{\sigma \xrightarrow{\mathtt{L}} \sigma' \quad (\bar{a},\mathtt{L}) \models S(\mathtt{L}) \xrightarrow{a_{\mathtt{L}}} (\bar{a}_{\mathtt{L}}, s_{\mathtt{L}}) \quad \delta(a_{\mathtt{L}}) = \mathtt{L} \quad S(\mathtt{H}) = (\bar{a}_{\mathtt{H}}, \_) \quad |\bar{a}_{\mathtt{L}}| + t < |\bar{a}_{\mathtt{H}}|}{\bar{a} \models (\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2) \xrightarrow{a_{\mathtt{L}}} (\sigma', S[\mathtt{L} \mapsto (\bar{a}_{\mathtt{L}}, s_{\mathtt{L}})], \bar{\alpha}_1, \bar{\alpha}_2.(\mathtt{L}, \iota(\bar{a}.a_{\mathtt{L}}), \iota(\bar{a}_{\mathtt{L}}), \bar{a}_{\mathtt{L}}))}\text{L-timeout}$$

(b) SME$_{\mathbb{T}}$ ℓ-chooser

Figure 9: Semantics of SME$_{\mathbb{T}}$

If a timeout ($\bar{\alpha}_2$) is discovered before the discrepancy ($\bar{\alpha}_1$), then the mismatch might be consequence of a timeout, which is not necessarily the basis of a leak in s.

We end this section with two PSNI-insecure programs, and explain attacks which SME$_{\mathbb{T}}$ finds on them. Consider

```
in M h ; out L h
```

With $\mathtt{d} = 0$, $t = 100$ and initial input 1, SME$_{\mathbb{T}}(s)$ generates an attack $(\mathtt{L}, I_1, I_2, \mathsf{M}?\mathtt{d}.\mathsf{L}!0)$ on s in $\bar{\alpha}_1$, where $I_1(\mathsf{M}) = \mathsf{M}?\mathtt{d}.(\mathsf{M}?\star)^\infty$, $I_2(\mathsf{M}) = \mathsf{M}?1.(\mathsf{M}?\star)^\infty$. Now consider

```
in M h ; while h != 0 { h := h - 1 } ; out L 0
```

With $\mathtt{d} = 0$, $t = 100$ and initial input -1, SME$_{\mathbb{T}}(s)$ generates an attack $(\mathtt{L}, I_1, I_2, \mathsf{M}?\mathtt{d}.\bullet.\mathsf{L}!0)$ on s in $\bar{\alpha}_2$, where $I_1(\mathsf{M}) = \mathsf{M}?\mathtt{d}.(\mathsf{M}?\star)^\infty$, $I_2(\mathsf{M}) = \mathsf{M}? - 1.(\mathsf{M}?\star)^\infty$. With initial input 5, no attack is generated. With initial input 500, however, an attack is generated which is not an attack on s (it just took "too long" for the $\mathtt{H}$-run to match L!0).

VII. Related work

Referring the reader for general overviews on language-based information-flow security [SM03], on dynamic information-flow control [Gue07], and on declassification [SS09], we focus on related work on multi-execution.

Li and Zdancewic [LZ05] observe that "a noninterfering program $f(h,l)$ can usually be factored to a 'high security' part $f_H(h,l)$ and a 'low security part' $f_L(l)$ that does not use any of the high-level inputs h. As a result, noninterference can be proved by transforming the program into a special form that does not depend on the high-level input." They propose *relaxed noninterference* that allows information release through a set of prescribed syntactic expressions. This focus is on enforcing relaxed noninterference statically, by a security type system.

Russo et al. [RHNS07] sketch the idea of running multiple runs of a program, where each run corresponds to the computation of information at a security level. They discuss that by running the public computation ahead of the secret run, certain classes of timing attacks can be prevented.

Capizzi et al. [CLVS08] consider enforcement of secure information flow in the setting of an operating system. The enforcement is based on *shadow executions* as operating system processes for different security levels. They report on an implementation and an experimental study with benchmarks.

As discussed earlier, Devriese and Piessens [DP10] develop a general treatment of secure multi-execution at the application level and establish soundness and precision under the assumption of total environments (there is always new input), linear lattices and low priority scheduling.

Bielova et al. [BDMP11a] investigate multi-execution in a reactive setting. Their model multi-executes Featherweight Firefox [BP10], a formalization of a web browser as a reactive system. The environments are not necessarily total, but the security guarantee is weaker (than Devriese and Piessens' [DP10]): termination-insensitive noninterference. The I/O model targets the browser setting, with handlers under cooperative scheduling. The full version [BDMP11b] contains an informal discussion of what the authors call *sub-input-event* security policies, which corresponds to more flexible policies on input events (flexible policies on output events are not considered). These policies are defined by projections that describe how much is visible at each level. This mechanism is however not formalized. A formalization would require reasoning about policy consistency: for example, projections for less restrictive levels should not reveal more than projections for more restrictive levels.

Kashyap et al. [KWH11] show that the low-priority scheduling might exhibit timing leaks for non-linear security lattices, and present several sound schedulers. We show (Appendix A) that in the presence of handlers, it is not necessary for the lattice to be non-linear to produce attacks on the low-priority scheduler. Timing leaks can freely occur in linear lattices, including the simple *low-high* lattice.

Jaskelioff and Russo [JR11] implement a monadic library for secure multi-execution in Haskell. Austin and Flanagan [AF12] introduce *faceted values* to simulate secure multi-execution by execution on enriched values. Faceted values can be projected to the different security levels. The projection theorem assures that a computation over faceted values faithfully simulates non-faceted computations. They show that faceted values guarantee termination-insensitive noninterference. Faceted values provide a viable alternative for an efficient implementation of our technique. Austin and Flanagan also show how to relax noninterference by facet declassification, based on *robust declassification* [ZM01], [MSZ06]. Robust declassification operates on both confidentiality and integrity labels, requiring both data to be declassified and code that does the declassification to be trusted. This leads to the introduction of integrity labels to model trust and integrity checks that the declassification operation is not influenced by untrusted data. This corresponds to the *who* dimension of declassification [SS09]. Compared to this approach, our declassification focuses on the *what* dimension, specifying what source and sinks are affected We are able to capitalize on what secure multi-execution is best at: built-in security against implicit flows. No matter where in the code declassification occurs, it will not leak information about the context. There is no need to track the integrity of the code in our model.

Barthe et al. [BCD$^+$12] present a "whitebox" approach to secure multi-execution. They devise a transformation that guarantees noninterference via secure multi-execution for programs in a language with communication and dynamic code evaluation primitives

De Groef et al. [GDNP12] implement secure multi-execution as an extension of the Firefox browser and report on experiments with browsing the web. Compared to the work by Bielova et al. [BDMP11a], they multi-execute the actual scripts in web pages rather than the entire browser. The main focus of the experiments is to confirm that the enforcement does not modify the behavior of secure pages in the presence of simple policies.

Compared to the work above, this paper enriches secure multi-execution with the following features: (i) channels with distinct presence and content security levels, (ii) the *what* dimension of declassification for secure multi-execution, (iii) full transparency results that preserve the order of messages, and (iv) show how secure multi-execution can be used to detect attacks. To the best of our knowledge, none of these features have been previously explored in the context of secure multi-execution.

This paper stands on the ground laid by our previous work [RHS12] on the foundations of security for interactive programs. This earlier work presents a general framework for environments as strategies, lifts the assumptions of the total environment, and distinguishes between the security level of message presence and content in the general setting. While the previous work provides an excellent starting point for the present paper, it does not treat secure multi-execution.

Unno et al. [UKY06] focus on the problem of finding counterexamples against noninterference: pairs of input states that agree on the public parts and lead to paths that disagree on public outputs. Their technique combines type-based analysis and model checking to explore execution paths for programs that may cause insecure information flow. They show that this method is more efficient than model checking alone. In comparison, our attack-detection technique does not require program analysis and allows reasoning about the security of individual runs.

In an independent effort, Zanarini et al. [ZJR13] apply secure multi-execution for program monitoring in a reactive setting modeled by interaction trees. This work relates to our attack detection results, although it focuses on a more relaxed, progress-insensitive, security condition. Given a program, the goal is to construct a scheduler for secure multi-execution that mimics the execution of the original program. Whenever a deviation is detected, the execution is blocked to avoid leakage. This approach enforces progress-insensitive noninterference.

VIII. Conclusion

Secure multi-execution emerges as a promising technique for enforcing secure information flow. We have overviewed the pros and cons of secure multi-executions and identified most pressing challenges with it. This paper pushes the boundary of what can be achieved with secure multi-execution. First, we lift the assumption from the original secure multi-execution work on the totality of the input environment (that there is always assumed to be input) and on cooperative scheduling. Second, we generalize secure multi-execution to distinguish between security levels of presence and content of messages. Third, we introduce a declassification model for secure multi-execution that allows expressing what information can be released. Fourth, we establish a full transparency result by barrier synchronization of the runs at different security levels. Full transparency guarantees that secure multi-execution preserves the original order of messages in secure programs. We demonstrate that full transparency is a key enabler for discovering attacks with secure multi-execution.

Representing reactive systems in our setting is an interesting topic of future work. In a reactive setting, an incoming input event determines which handler may be triggered. We can model this by tagging input values with a channel id. The program can then pattern-match on the tag to dispatch the value to the handler associated with the channel.

Although our formal results on declassification focus on *what* information is released, the mechanism also supports *where* information is released. Indeed, our declassification mechanism restricts release to program points with declassification annotations. Therefore, we expect the mechanism to enforce a variant of intransitive noninterference [Rus92]. An investigation of formal guarantees for the *where* dimension of declassification is a worthwhile topic for future work.

Future work also includes implementation and case studies. We plan to experiment with modifying the Firefox browser to accommodate fine-grained, declassification-aware, and transparent secure multi-execution. The modification will allow us to multi-execute JavaScript code in an environment with preemptive scheduling of the runs at different levels.

Acknowledgments: Thanks are due to Frank Piessens for generous feedback, and to Daniel Hedin, and David Sands for the useful discussions. This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR.

References

[AF09] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

[AF10] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2010.

[AF12] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 165–178, 2012.

[Aga00] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, January 2000.

[AS09] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[AZM10] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *CCS*, 2010.

[Bar03] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[BCD$^+$12] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. Secure multi-execution through static program transformation. In *Formal Techniques for Distributed Systems (FMOODS/FORTE 2012)*, volume 7273 of *LNCS*, pages 186–202, June 2012.

[BDMP11a] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proceedings of the 5th International Conference on Network and System Security (NSS)*, 2011.

[BDMP11b] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for the browser: extended version. Technical Report CW602, CS Dept., K.U.Leuven, February 2011.

[BP10] A. Bohannon and B. C. Pierce. Featherweight Firefox: Formalizing the Core of a Web Browser. In *Proc. of the USENIX Conference on Web Application Development*, 2010.

[BPS$^+$09] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *ACM Conference on Computer and Communications Security*, pages 79–90, November 2009.

[CH08] D. Clark and S. Hunt. Noninterference for deterministic interactive programs. In *Workshop on Formal Aspects in Security and Trust (FAST'08)*, October 2008.

[CLVS08] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In *Annual Computer Security Applications Conference (ACSAC)*, pages 322–331, 2008.

[Den76] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

[DP10] D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *Proc. IEEE Symp. on Security and Privacy*, May 2010.

[GDNP12] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security*, October 2012.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[Gue07] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[HS12] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *Proc. IEEE Computer Security Foundations Symposium*, pages 3–18, 2012.

[JR11] M. Jaskelioff and A. Russo. Secure multi-execution in haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 7162 of *LNCS*, pages 170–178. Springer-Verlag, June 2011.

[KWH11] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. IEEE Symp. on Security and Privacy*, 2011.

[LBJS06] G. Le Guernic, Anindya Banerjee, Thomas Jensen, and David Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)*, volume 4435 of *LNCS*. Springer-Verlag, 2006.

[Le 07] G. Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proc. IEEE Computer Security Foundations Symposium*, pages 218–232, July 2007.

[LZ05] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 158–170, January 2005.

[MSZ06] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Computer Security*, 14(2):157–196, May 2006.

[MZZ$^+$01] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[NIK$^+$12] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *ACM Conference on Computer and Communications Security*, pages 736–747, October 2012.

[OCC06] K. O'Neill, M. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 190–201, July 2006.

[RHNS07] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Asian Computing Science Conference (ASIAN'06)*, LNCS. Springer-Verlag, 2007.

[RHS12] W. Rafnsson, , D. Hedin, and A. Sabelfeld. Securing interactive programs. In *Proc. IEEE Computer Security Foundations Symposium*, June 2012.

[RS09] A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[RS10] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.

[RS11] W. Rafnsson and A. Sabelfeld. Limiting information leakage in event-based communication. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2011.

[RS13] W. Rafnsson and A. Sabelfeld. Secure multi-execution: fine-grained, declassification-aware, and transparent: Extended version. Located at http://www.cse.chalmers.se/~rafnsson/2013csf.pdf, 2013.

[Rus92] J. M. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, 1992.

[SD12] N. Singer and C. Duhigg. Tracking Voters' Clicks Online to Try to Sway Them. http://www.nytimes.com/2012/10/28/us/politics/tracking-clicks-online-to-try-to-sway-voters.html, October 2012.

[Sim03] V. Simonet. The Flow Caml system. Software release. Located at http://cristal.inria.fr/~simonet/soft/flowcaml, July 2003.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[SR09] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[SS09] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 17(5):517–548, January 2009.

[SST07] P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proc. IEEE Computer Security Foundations Symposium*, pages 203–217, July 2007.

[UKY06] H. Unno, N. Kobayashi, and A. Yonezawa. Combining type-based analysis and model checking for finding counterexamples against non-interference. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 17–26, 2006.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[ZJR13] D. Zanarini, M. Jaskelioff, and A. Russo. Precise enforcement of confidentiality for reactive system, 2013.

[ZM01] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.

Appendix

A. FlowFox leak

The leak exploits the fact that FlowFox [GDNP12] multi-executes JavaScript with the low-priority scheduler on a per-event basis. Low priority implies that the low run is executed first and without preemption by the high run. The low-priority scheduler first applies to the main code. If the main code sets event handlers, they are processed after the multi-execution of the main code. Low handlers are multi-executed. High handlers are only run once, at the high level.

Note that the problem with low-priority scheduling is fundamental because it is not possible to extend the low-priority discipline over multiple events—simply because it is not possible to run the low handlers that have not yet been triggered.

The security theorem in the abstract setting of secure multi-execution [DP10] takes advantage of the low-priority scheduler and establishes timing-sensitive security. This is intuitive because the last access of the low data occurs before any high data is accessed. This implies that whenever the timing behavior is affected by secrets, there is no possibility for the attacker to inspect the difference.

We show that the situation is different in the presence of handlers. All we need to do is to set a low handler to execute after the high run for the main code has finished. Then the low handler can inspect the computation time taken by the high run. For a simple experiment, we consider the default example policy from the FlowFox distribution[1] in Listing 1.

Listing 1: FlowFox policy

```
1  /** Example policy file for FlowFox.
2
3      Detailed project information and contact
           address can be found on:
4      https://www.distrinet.cs.kuleuven.be/software/
           FlowFox/
5
6      HOWTO: modify the policy rules at the end this
            file
7  **/
8
9  ... non-customizable part of the policy skipped...
10
11 /************************************************/
12 /*********************** E D I T    M E *******/
13 /************************************************/
14
15 /* Example label conditional function */
16 var cross_origin = function ([url]) { return (url.
       indexOf("same-origin") == -1); };
17
18 /* Examples */
19 SME.Label("nsIDOMHTMLDocument_GetCookie").as(SME.
       Labels.HIGH).default("eat=this");
20 SME.Label("nsIDOMHTMLImageElement_SetSrc").if(
       cross_origin).as(SME.Labels.LOW).else(SME.
       Labels.HIGH);
21 SME.Label("nsIDOMHTMLScriptElement_SetSrc").as(SME
       .Labels.LOW).if(cross_origin).else(SME.Labels.
       HIGH);
```

The listing omits the non-customizable part of the policy, focusing on the sources and sinks. This policy defines same-origin domain as `HIGH` and cross-origin domains as `LOW` (line 16). In order to protect cookies, secret source are defined by labeling `document.cookie` as `HIGH` (line 19). Lines 20 and 21 define the sinks that correspond to setting the source attributes of image and script HTML elements. These are labeled as `HIGH` for the same origin and `LOW` for the other origins. The intention is to prevent attacks that leak information about the cookie to third-party web sites (any sites other than the site of the web page origin).

Nevertheless, the code in the web page in Listing 2 leaks one bit of information about the cookie to the third-party web site `attacker.com`.

Listing 2: One–bit timing leak

```
1  <html>
2  <script>
3  var c = new Date();
4  var m = c.getTime();
5  setTimeout(function() {leak();},1);
6  document.cookie="1";
7  //document.cookie="0";
8  var h=parseInt(document.cookie,10);
9  if (h > 0) {
10     var t = 0; while( t < 10000000) {t++;}
11 }
12
13 function leak() {
14     var d = new Date();
15     var n = d.getTime();
16     var x = n-m;
17     var s = new Image();
18     s.src = "http://attacker.com?v=" +
           encodeURIComponent(x);
19 }
20 </script>
21 <head></head>
22 <body>One-bit timing leak</body>
23 </html>
```

Function `getTime()` of the `Date` object returns the number of milliseconds since the midnight of January 1, 1970. First, information about the cookie flows via `document.cookie` into variable `h` (line 8). Depending on the value of `h`, the program might take longer time to execute (line 10). As foreshadowed below, all we need to do is to get a time stamp at the beginning of execution (line 4) and after the high run has finished. The difference in time reveals whether `h` was zero. In order to bypass FlowFox's multi-execution, we simply create a low handler (line 5) to perform the final time measurement (line 15). Running this page in FlowFox results in a request for an image with URL `http://attacker.com?v=496` (repetitive runs show slight fluctuation around the value of 496). Running the code with line 7 uncommented and line 6 commented out, results in a request for an image with URL `http://attacker.com?v=6` (repetitive runs fluctuate insignificantly around the value of 6). Hence, we can reliably leak one bit of secret information about the cookies. Clearly, the leak can be easily magnified to leak the entire cookie by walking through it bit-by-bit in a simple loop and sending the results for each bit to the attacker.

Note that changing the policy for `getTime()` to return `HIGH` result does not close the timing leak. The leak can be still achieved exploiting the difference in the *internal timing* behavior by a combination of low handlers [RS09].

While the leak outlined above is achieved by issuing a timeout event, other events (such as user-generated events and `XMLHttpRequest`) can be used to achieve the same effect.

The low-priority scheduler is both at the heart of the soundness results by Devriese and Piessens [DP10] and at the heart of FlowFox [GDNP12]. The experiment points to a fundamental problem with low-priority scheduling. The leak demonstrates that the low-priority scheduler breaks timing-sensitive security and motivates the need for (fair) interleaving of the runs at different levels, as pursued in this paper.

[1]https://distrinet.cs.kuleuven.be/software/FlowFox/