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 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 offered 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 [MZ01], FlowCaml [Sim03], and SparkAda Examiner [Bar03] that enhance compilers for Java, Caml, and Ada, respectively. Recently, dynamic monitoring techniques have received increased attention (cf. [LBJ06], [Le07], [SST07], [SR09], [AF10], [HH12]), 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], [BDMP11], [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 \( \ell \) may flow to an output sink at level \( \ell' \) only if \( \ell \subseteq \ell' \) [Den76].

For simplicity, we will often use the two-level lattice with a secret (high) level and a public (low) level of confiden-
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 various 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 website).
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 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].
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, RST11, RHST].

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 \(s \xrightarrow{c \cdot v} s'\) 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.

\[
a ::= i | o \quad i ::= c!v \quad o ::= c?v | \bullet
\]

Here, \(c!v\) (resp. \(c?v\)) denotes a message received (resp. sent) on channel c carrying value v, and \(\bullet\) denotes a noninteraction. \(c\) and \(v\) range over the (nonempty) sets \(C\) and \(V\) 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 [RHST].

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

**Definition III.2.** Let \(\lambda = (S, L, \rightarrow)\) be an LTS\(_{IO}\).

1. \(\lambda\) is input-neutral iff
   \[
   \forall s \in S, c \cdot (\exists v \cdot s \xrightarrow{c?v}) \implies (\forall v \cdot s \xrightarrow{c?v}).
   \]
2. \(\lambda\) is input-blocking iff
   \[
   \forall \{s, s'\} \subseteq S, c \cdot s \xrightarrow{c} s' \implies s' = s.
   \]
3. \(\lambda\) is deterministic iff
   a) \(\forall s \in S, a_1, a_2 \cdot s \xrightarrow{a_1} \wedge s \xrightarrow{a_2} \wedge a_1 \neq a_2 \implies \exists c, e_1, e_2 \cdot c!e_1 \wedge c!e_2 \implies c = c_1 \wedge c = c_2,\) and
   b) \(\forall \{s, s_1, s_2\} \subseteq S, a \cdot s \xrightarrow{a} s_1 \wedge s \xrightarrow{a} s_2 \implies s_1 = s_2.\)

Pt. 1 states that if s is ready to perform input, s be receptive to any v. Pt. 2 states that if input is a blocking operation. Pt. 3a says if \(s \xrightarrow{a}\), then \(s \xrightarrow{a} \) iff \(a \in \{a?v | v \in V\}\), and implicitly, if \(s \xrightarrow{\cdot}\), then \(s \xrightarrow{\cdot} \) iff \(a = a\). Pt. 3b says s has no internal nondeterminism. Unless stated otherwise, any s we consider in this paper is from a \(\lambda\) 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 \(a = a_1 \ldots a_n\). Let \(\bar{a}_i\), \(\bar{a}_i|1\) and \(\bar{a}_i|c\) denote the projection of \(\bar{a}\) to its input-, output- and c-messages, respectively. E.g., if \(\bar{a} = c!1.c!2.c?!c!4\), then \(\bar{a}_1 = c!1.c?!c!4\), \(\bar{a}_i|1 = c!1\), and \(\bar{a}_i|c\) = c!1.c!4. \(\bar{a}_i\) extends to \(\bar{a}_i|c\) for \(C \subseteq C\) in the obvious way. We write \(\bar{a}|_{x_1, \ldots, x_n}\) as short for \(\bar{a}|_{x_1, \ldots, i_n}\); we refer to each \(x_j\) as a projection predicate. With \(\bar{a}\) defined as above, \(\bar{a}|_{c.?c} = \bar{a}|_{c.?} = c?1\).

We write \(\bar{x} \leq \bar{y}\) when, for some \(\bar{x}'\), \(\bar{y}'\), \(\bar{x}' = \bar{y}'\). For a relation \(R\), \(\bar{a} R_{x_1, \ldots, x_n} \bar{a}'\) is short for \((\bar{a}|_{x_1, \ldots, x_n}) R (\bar{a}'|_{x_1, \ldots, x_n})\). 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 \((L, \sqsubseteq, L)\), with \(\sqsubseteq\) ranged by \(\ell\), 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 \(L_{\mathbb{Z}} = \{L, H\}\) with \(L = \{L, L\}, (L, H), (H, H)\}, \(L\) for "low" confidentiality, \(H\) for "high". We let \(H\), \(M\), \(L\) denote \(H^\ell\), \(L^\ell\) and \(L^\ell\), 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).

The security labels express what we can observe what. An observer is associated a security level \(\ell\). An \(\ell\)-observer is capable of observing the presence (resp. content) of a message on \(c\) if \(\delta(c) \sqsubseteq \ell\) (resp. \(\gamma(c) \subseteq \ell\)). \(\bar{a}\) removes \(\ell\)-unobservable parts of actions in \(\bar{a}\). For \(\bar{a} = \bullet.L\sqsubseteq.L.H1.M?\sqsubseteq.M!2.H?\bullet\), \(\bar{a}|_L = \bullet.L\sqsubseteq.M?\sqsubseteq.M!2.H?\bullet\). H1 got replaced with \(\bullet\) since communication on H is unobservable to a L-observer (thus looks like a \(\bullet\)). M!2 got replaced with M\!d (for a fixed \(d \in V\)), since a L-observer only observes presence of messages on \(M\) (all \(a \in \{M!\} \sqsubseteq \ell\}) look the same.

Timing and progress: Eventually we enforce a property stating that variations in unobservable inputs to a system do not cause an \(\ell\)-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. \( \bar{a} \) replaces all \( c? \) with \( \ast \). With \( \ast \) defined as above, \( \bar{a} \) removes trailing \( c \) from \( \bar{a} \). E.g. \( \bar{a} \bar{a} = \ast.L0.H!1.\ast.M12.H\ast.\ast \).

A timing-sensitive attacker measures time between observables in a trace. \( \bar{a} \) removes trailing \( c \) from \( \bar{a} \). A progress-sensitive attacker observes whether there are more observables forth-coming in a trace. \( \bar{a} \) replaces all \( c \) from \( \bar{a} \). With \( \ast \) defined as above, \( \bar{a} \) removes trailing \( c \) from \( \bar{a} \). A progress-sensitive attacker observes whether there are more observables forth-coming in a trace. \( \bar{a} \) replaces all \( c \) from \( \bar{a} \). A progress-sensitive attacker observes whether there are more observables forth-coming in a trace. \( \bar{a} \) replaces all \( c \) from \( \bar{a} \).

Define timing-sensitive \( \ell \)-equivalence as \( \approx_{\ell} \). Consider

\[
\begin{align*}
\text{in H } & ; \text{ out L 0} \\
\end{align*}
\]

Since this program can perform traces \( \bar{a}_1 = L0.H!1.L0 \) and \( \bar{a}_2 = H?0.L0 \), this program is not secure against a timing-sensitive \( \ell \)-observer since \( \bar{a}_1 \not\approx \bar{a}_2 \). We define progress-sensitive \( \ell \)-equivalence as \( \approx_{\ell} \). Consider

\[
\begin{align*}
\text{in H } & ; \text{ out L 0} \\
\end{align*}
\]

A progress-sensitive attacker observes whether there are more observables forth-coming in a trace. \( \bar{a} \) replaces all \( c \) from \( \bar{a} \). With \( \ast \) defined as above, \( \bar{a} \) removes trailing \( c \) from \( \bar{a} \). A progress-sensitive attacker observes whether there are more observables forth-coming in a trace. \( \bar{a} \) replaces all \( c \) from \( \bar{a} \). A progress-sensitive attacker observes whether there are more observables forth-coming in a trace. \( \bar{a} \) replaces all \( c \) from \( \bar{a} \).

D. Environments

The inputs to our systems come from the environment in which our systems run. Clark and Hunt 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 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 \( I_c = I(c) \),

\[
\forall \bar{a}' \leq \bar{a} \exists \bar{a}' \leq I_c (\bar{a}' = \bar{a}' \land \bar{a}' \leq I_c (\bar{a}' \land \bar{a}' \leq I_c (\bar{a}' \land \bar{a}' \leq I_c (\bar{a}')).
\]

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

Consistency implies that \( I \) queues arriving input. Consider

\[
\begin{align*}
in \ast \times ; \text{ while } |c| \{ x = |c| - 1 \} ; \text{ in c y} \\
\end{align*}
\]

Let \( s \) be the LTS\(_{\bar{a}}\) state of this program and let \( I_1, \ldots, I_5 \) be given such that \( I_1c = (c?c?c?c?c?c?)^\infty \), \( I_2c = c?0.c?0.(c?c?)^\infty \), \( I_3c = (c?c?0.(c?c?)^\infty \), \( I_4c = c?0.c?c?0.(c?c?)^\infty \), \( I_5c = (c?c?c?c?c?c?c?c?c?c?c?c?c?c?c?c?)^\infty \).

Stream equivalence becomes: \( I_1 \approx_{\ell} I_2 \) (resp. \( I_1 \approx_{\ell} I_2 \)) iff \( c \in \delta(c) \subseteq \ell \implies I_1(c) \approx_{\ell} I_2(c) \) (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 \( \ast \) does not occur in any \( I_c \). Previous work on security for interactive programs assume that environments are total. However, as we have demonstrated previously, 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 \( I_{1H} = H?0.(c?c?)^\infty \) and \( I_{2H} = I_{1c} = I_{2c} = (c?c?)^\infty \) for all \( c \neq H \). While this program can perform \( H?0.L0 \) under \( I_1 \), the program cannot perform an \( \approx_{\ell} \)-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 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.

\[
\begin{align*}
\text{while } 1 & ( \text{ in } H_1 x ; \text{ out } L 1 ; \text{ out } H_1 42 ) \\
\text{while } 1 & ( \text{ in } H_0 x ; \text{ out } L 0 ; \text{ out } H_0 42 ) \\
\text{in H } & ; \text{ out L 1 ; out H}_1 42 \\
\text{for } b \text{ in bits(h) } \{ \\
\text{ if } b \{ \text{ out } H_1 42 ; \text{ in } H_1 x \} \\
\text{ else } \text{ out } H_0 42 ; \text{ out } H_0 x \}
\end{align*}
\]

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, non-total 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.

IV. FINE-GRAINED SECURE MULTI-EXECUTION

The opening series of our contributions develops a generalization of SME with respect to several dimensions. We lift the assumption on the totality of the input environment (that there is always supposed 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 LTS\(_{\bar{a}}\) and strengthen the guarantees SME provides.

By design, our formalization of SME ensures that the \( \ell \)-observable part of the interaction on channels with \( \ell \) presence depends only on \( \ell \)-observable parts of input on channels with \( \subseteq \ell \) presence, thus enforcing a noninterference policy.
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 δ(c)-run can input on c. An ℓ-run which can consume a c-input with δ(c) ⊑ ℓ is fed a (constant, pre-determined, input-independent) default value, denoted d, by SME. An ℓ-run which can consume its n-th c-input with δ(c) ⊑ ℓ gets a copy of the nth input consumed by the δ(c)-run, unless δ(c) is yet to consume n c-inputs, in which case the ℓ-run blocks until the δ(c)-run has done so. However, if γ(c) ⊏ ℓ, then the ℓ-run is fed d instead of the value in the nth c-input. The SME of s can output on c iff the δ(c)-run can output on c. A c-output produced by a ℓ-run for which δ(c) ⊏ ℓ is discarded by SME (as opposed to being sent to the environment). When an ℓ-run produces its nth c-output with ℓ = δ(c), this output is sent straight to the environment, except when δ(c) ̸= γ(c); in that case SME first checks whether the γ(c)-run has already produced its nth c-output. If so, then the value of the nth c-output produced by SME of s becomes the value of the nth c-output produced by the γ(c)-run. Otherwise, the value becomes 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 ∀ℓ, ℓ′, σ ̸→ ℓ → σ → ℓ = ℓ′. σ is fair iff ∀ℓ, σ ̸→ ℓ → ∀ℓ′, ∃σ ′ → ℓ′ ̸→ ℓ → ℓ′.

Unless stated otherwise, σ is deterministic and fair. An example of deterministic and fair schedulers is the round-robin schedulers. For instance, for L_{H, L}, a scheduler which infinitely repeats H. L or L. H is a deterministic fair scheduler.

The semantics of SME is given in Figure 5. A SME state is a triple (a, σ, S), where 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 (a_ℓ, s_ℓ), where a_ℓ 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, SME(σ, s), is defined as SME(σ, s) = (ε, σ, λℓ → (ε, s))). The derivation of any SME state transition begins with the rule
\[
\bar{a} \models (σ, S) \xrightarrow{a} (σ', S')
\]
\[
\bar{a}, σ, S \xrightarrow{a} (\bar{a}, a, σ', S')
\]
The purpose of log is to keep track of the interaction a which SME(σ, s) has had with the environment. Note that SME(σ, s) ̸→ (a′, σ′, S′) ⇔ a = a′,

so we sometimes omit the trace label on a SME transition. We put a on the left side of "−→" in the next layer of the semantics, Figure 5, to show that this layer only reads 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 ≠ ℓ without forwarding it to the environment, as per (old-o), (old-i) covers multiple scenarios for not inputting from the environment on c. When δ(c) ⊏ ℓ, input d. When δ(c) ⊑ ℓ, this rule is only applicable if the ℓ-run has not already read all the c-inputs which the δ(c)-run has read. When γ(c) ⊏ ℓ, input the same value received from the environment when the δ(c)-run performed the corresponding input action. Otherwise, input d instead. (new-o) 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 d to the ℓ-run iff v ≠ σ ∧ γ(c) ⊏ ℓ. The previous layer has two rules for this scenario. When δ(c) ⊏ ℓ, then the ℓ-run blocks until the δ(c)-run reads on c, by (block-i). When δ(c) reads on c, the input is fed to every δ(c)-run blocking on c, by (l). (new-o) notifies the previous layer that the ℓ-run has a fresh output for the environment. Rule (0) in the previous layer handles this scenario, checking if the γ(c)-run has already provided content for this output, and if so, replaces the value in the output with the value in the corresp. γ(c)-run c-output.

When γ(c) ̸= δ(c), the output value is replaced by d if γ(c) has not yet produced the corresponding value. Having the δ(c)-run instead wait for the γ(c)-run to reach the corresponding c-output, or giving responsibility of producing the c-output to the γ(c)-run, can introduce a leak:

\[
\text{in M h l := 0; while l := h (l := l+1); out M h}
\]

Here the time it takes for the h-run to produce the M-output, and whether h produces the output at all, depends on h.
While Figure 5B says that SME controls each step of each \( \ell \)-run, in practice the responsibility of SME can be distributed to the \( \ell \)-runs as follows. Each \( \ell \)-run is made responsible for environment I/O on all \( c \in \delta^{-1}(\ell) \), since SME(\( s \)) performs I/O iff the \( \ell \)-run of \( s \) performs it. Each \( \ell \)-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. \( \Box \ \ell \)-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 \( \ell \)-run processes (after sharing, when \( \ell = \delta(c) \)) \( d \) in place of the inputted value when \( \delta(c) \subseteq \ell \subseteq \gamma(c) \), and outputs \( d \) when the \( \gamma(c) \)-run is yet to ready the value to put into the output when \( \delta(c) \subseteq \gamma(c) = \ell \). This approach is taken in a SME benchmark by Devriese and Piessens [DP10]. Forcing \( \ell \)-runs to diverge and recording full traces can be avoided [KWH11, DP10]. This approach is sound as long as the \( \ell \)-run threads cannot influence the scheduler.

While \( s \) is input-blocking, SME(\( s \)) is not; varied present of input on \( c \in \gamma^{-1}(\ell) \) cannot impede on progress or timing of \( \ell' \)-runs where \( \ell \not\subseteq \ell' \). This effect is achieved by \( c!\ast \)-actions; if SME(\( s \)) is in a state where an \( \ell \)-run wants input on \( c \), and \( I \) does not have one ready (yet), the \( \ell \)-run can do a \( c!\ast \)-action, allowing SME(\( s \)) to pass control to another \( \ell' \)-run. In contrast, the formalization (as opposed to the benchmark implementation) of SME by Devriese and Piessens [DP10] is input blocking: if an \( \ell \)-run is scheduled before a \( \ell' \)-run with \( \ell \not\subseteq \ell' \), the nonpresence of input on \( c \in \delta^{-1}(\ell) \) can interfere with the \( \ell' \)-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 (\( s \in TSNI \)) iff \( \forall \ell, I_1, I_2, \ell.I_1 \simeq_\ell I_2 \implies \forall \bar{a}_1, I_1 \models s \bar{a}_1 \implies \exists \bar{a}_2, I_2 \models s \bar{a}_2 \land \bar{a}_1 \simeq_\ell \bar{a}_2. \)

### Theorem IV.3.

\( \forall \sigma, s, \sigma(\sigma, s) \in TSNI. \)

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 select\(_{lowprio} \), which executes the \( \ell \)-runs to completion in increasing order by \( \subseteq \), works for any linearized lattice, Kashyap et al. have shown that select\(_{lowprio} \) introduces a timing dependency between \( \ell \)-Runs at incomparable levels in nonlinear lattices. For instance, with \( L = L_{AB} = H, A, B, L \) and \( \subseteq \) being the reflexive transitive closure of \( \{ (L, A), (L, B), (A, H), (B, H) \} \), with linearization \( L \subseteq A \subseteq B \subseteq H \) and \( d = 0 \), the time it takes for \( B^0!1 \) to occur is, under select\(_{lowprio} \), a function of the input on \( A^k \).
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 and the details of this, and all other proofs, are in the full version of this paper [RST13]. We write \( S_1 = S_2 \) iff \( \forall \ell' \subseteq \ell, S_1(\ell') = S_2(\ell') \).

**Lemma IV.4.** \( \forall s, \sigma, \ell, I_1, I_2, \exists I \Rightarrow \)

\[
\forall a_1, \sigma_1, I_1 \models \text{SME}(\sigma, s) \Rightarrow (a_1, \sigma_1, S_1) \Rightarrow \exists a_2, \sigma_2, I_2 \models \text{SME}(\sigma, s) \Rightarrow (a_2, \sigma_2, S_2) \land a_1 = \ell \land \exists a_2 \land \sigma_1 = \sigma_2
\]

Such a strong correspondence is achievable since, for each \( \ell' \subseteq \ell \), the \( \ell' \)-run of \( s \), in \( I_1 \models \text{SME}(\sigma, s) \) and \( I_2 \models \text{SME}(\sigma, s) \), behaves as \( I|_{\ell'} \models s \), where

\[
(I|_{\ell'}(c)) = \begin{cases} (c?)^{\infty} \text{ if } \delta(c) \not\subseteq \ell' \hfill, \\ I(c)|_{\ell} \hfill, \text{ otherwise.} \end{cases}
\]

(While for \( \delta(c) \not\subseteq \ell' \), the number of \( c?\ast \) preceding a \( c?v \) of an \( \ell' \)-run in \( I_1 \models \text{SME}(\sigma, s) \) compared to \( I|_{\ell'} \models s \) can differ due to \( \sigma \), this number is the same in \( I_1 \models \text{SME}(\sigma, s) \) compared to \( I_2 \models \text{SME}(\sigma, s) \). 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 \( I|_{\ell'} \models s \). Thus, since \( I_1 \models \text{SME}(\sigma, s) \) and \( I_2 \models \text{SME}(\sigma, s) \) are both run under the same \( \sigma \), after any number of transitions, the \( \ell' \)-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 \( \ell \)-unobservable input does not affect the \( \ell \)-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, ‘\( \preceq \ell \)’ with ‘\( \preceq \ell' \).

**Definition IV.5.** \( s \) is timing-insensitive, progress sensitive noninterfering (\( s \in \text{PSNI} \)) iff \( \forall \ell, I_1, I_2, I_1 \models \ell \Rightarrow \exists I_2 \models \ell \Rightarrow \exists a_1 \models a \Rightarrow \exists a_2 \models a \land a \models a_1 \models \ell' \)

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

**Theorem IV.6.** \( \forall s \in \text{PSNI}, I, a \).

\[
a \models a \Rightarrow \exists a' \models \text{SME}(\sigma, s) \models \ell \Rightarrow a' \preceq \ell \Rightarrow \end{cases}
\]

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

\[
\text{out } B^a I \text{ in } A \]

However, when \( s \) outputs on \( c \) with \( \gamma(c) \not\subseteq \delta(c) \), SME(\( \sigma, s \)) might replace the value in its corresponding output with \( d \). Thus, the timing behavior of \( s \in \text{PSNI} \) can impede the ability of \( a \) to soundly schedule runs in 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 \( \ell \)-runs for which \( \delta(c) \not\subseteq \ell \) will reach an output on \( c \) after the same number of reduction steps. This includes the \( \gamma(c) \)-run, since \( \delta(c) \not\subseteq \ell \). Thus, if we ensure that any \( \ell \)-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, \( B \) has been scheduled more often than \( A \), as the \( B \)-run can waste its turns blocking on channels with presence level \( \preceq H \). The following predicate, when invoked as \( \phi(\ell_\text{H}, \ell_\text{L}, 0, \ell) \), yields 1 when \( \ell_\text{H} \), a parent of \( \ell_\text{L} \), has been scheduled in this manner in \( \ell_\text{L} \), and 0 otherwise.

\[
\phi(\ell_\text{H}, \ell_\text{L}, b_{\text{biseen}}, \ell) \begin{cases} 1 \hfill, \\ \phi(\ell_\text{H}, \ell_\text{L}, 0, \ell) \hfill, \\ \phi(\ell_\text{H}, \ell_\text{L}, 0, \ell) \hfill, \\ \phi(\ell_\text{H}, \ell_\text{L}, b_{\text{biseen}}, \ell) \hfill, \end{cases}
\]

**Definition IV.7.** \( \sigma \) is a high-head scheduler (\( \sigma \in \text{highhead} \)) if \( \forall \ell, \sigma | \ell \Rightarrow \forall \ell_\text{H}, \ell_\text{L}, \ell_\text{H} \subseteq \ell_\text{L} \Rightarrow \phi(\ell_\text{H}, \ell_\text{L}, 0, \ell) \).

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

**Theorem IV.8.** \( \forall s \in \text{TSNI}, \sigma \in \text{highhead}, I, a \).

\[
a \models a \Rightarrow \exists a' \models \text{SME}(\sigma, s) \models \ell \Rightarrow a' \preceq \ell \Rightarrow \end{cases}
\]

Thus, if SME puts a \( d \) in an \( M \) output when run using \( \sigma \in \text{highhead} \), 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 to be declassified. This is done via a convenient match with our model that distinguishes the context while allowing in-

Figure 6: SME with declassification

\[ P \quad \text{H} \quad L \quad P_H \quad H \quad L \quad P_L \quad L \]

\[ H \quad L \quad H \quad L \]

\[ \text{Definition V.1.} \ I_1 \text{ and } I_2 \text{ are } \rho-\ell\text{-equivalent } (I_1 \simeq^\rho_{\ell} I_2) \text{ iff } \]

\[ I_1 \simeq_{\ell} I_2 \land \forall \ell' \cdot \rho \ell \implies I_1 \simeq_{\ell'} I_2. \]

\[ \text{Definition V.2.} \ s \text{ is } \rho\text{-releasing TSNI } (s \in \text{TSNI}_\rho) \text{ iff } \]

\[ \forall \ell_1, I_1, I_2 \simeq^\rho_{\ell_1} I_2 \implies \]

\[ \forall a_1 \cdot I_1 = s \Rightarrow a_1 \Rightarrow \exists a_2 \cdot I_2 = s \Rightarrow a_2 \wedge \exists a_1 \simeq_{\ell_1} a_2. \]

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 \text{declassify}(e, \ell), which declassifies the expression \( e \) (in the state in which this command is executed) to level \( \ell \). For instance, only \( b \) is intended to leak to \( L \) in the below program \( p_{d1} \) under lattice \( L_{ab} \) with \( p = \{ (B, L) \} \), but it turns out \( a \) leaks as well; a TSNI-if-insecurity.

\[ \text{in } k \quad \overleftarrow{a}; \text{ in } b \quad \overrightarrow{b}; \]

\[ \text{if } a \wedge (l := \text{declassify}(b, L)) \]

\[ \text{out } L \]

The only interface SME has to its \( \ell \)-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 \( R \subseteq C \) be the release channels, ranged by \( r \). Let \( \varphi : L^2 \rightarrow R \) be a bijection. \( \varphi \) associates a release channel \( \varphi(\ell, \ell') \) with each kind of information release \( (\ell, \ell') \) (not all of which are permitted by \( \rho \)). 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 \( \ell \) and the to level \( \ell' \) of the declassification operation.

\[ m := v; \varphi(\ell, \ell') = r \]

\[ m, x := \text{declassify}(e, \ell \rightarrow \ell')) \rightarrow (m', \text{skip}) \]

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 \( F(s) \), given below. \( F(s) \) only interacts with its environment on release channels.

\[ \text{s} \xrightarrow{r,v} s' \]

\[ F(s) \xrightarrow{r,v} F(s) \]

\[ F(s) \xrightarrow{r,v} F(s') \]

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.

\[ \text{Definition V.3.} \text{ An LTSIO } \lambda = (S, L, \rightarrow) \text{ is an LTSIO with release } (\text{LTSIO}_R) \text{ iff } \]

\[ \forall s \in S, r, v, a \cdot (s \overrightarrow{r,v,r'v'}) \implies \exists s' \cdot s \overrightarrow{r,v,r'v'} \]

\[ \lambda \text{ is an LTSIO without release } (\text{LTSIO}) \text{ iff } \]

\[ \forall s \in S \cdot (s \overrightarrow{r,v,a}) \Rightarrow \exists s' \cdot a = \overrightarrow{r,v,a} \]

We write \( s \in \text{LTSIO}_R \) (resp. \( s \in \text{LTSIO} \)) if \( s \) is a state in some \( \text{LTSIO}_R \) (resp. \( \text{LTSIO} \)).

We consider only \( \text{LTSIO}_R \) and \( \text{LTSIO}_C \) \( s \) for the remainder of this section; an \( s \) which is neither interacts on some \( r \) in an undesired way. We require that \( \forall \gamma(r) = \gamma(r) \wedge \exists \ell' \cdot \varphi(\gamma(r), \ell') = r \). Read this as "\( r \) releases information from \( \gamma(r) \) to \( \ell' \)." 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\(_R\)) is given in Figure 7. It is parameterized by the release policy \( \rho \). By \( \not\rightarrow \) for any release action which is not permitted by \( \rho \) is a feedback. By \( \gamma \rightarrow \), when a target \( \ell \)-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 \( \gamma \), if a valid release receiver
has already received \( d \) for this release, so does the \( \ell \)-run. Otherwise \( \gamma(r) \) has already done the release, so the \( \ell \)-run receives the right value.

**A. Soundness**

\( \text{SME}_R \) is sound with regards to any release policy.

**Theorem V.4.** \( \forall s \in \text{LTS}_{\text{IO}}^{R}, \sigma, \rho : \text{SME}_R(\sigma, s, \rho) \in \text{TSNI}_p \)

The proof of this theorem follows a similar pattern as the proof of Theorem [V.3] utilizing a lemma which is near-identical to Lemma [V.2].

\( \text{SME}_R \) 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.** \( \forall s \in \text{LTS}_{\text{IO}}^{R}, \sigma, \rho : \text{SME}_R(\sigma, s, \rho) \in \text{TSNI} \)

When no information release is permitted, Definitions [V.2] and [V.3] coincide (\( I_1 \simeq^0_\sigma I_2 \) becomes \( I_1 \simeq_\sigma I_2 \)).

**Corollary V.6.** \( \text{TSNI}_0 = \text{TSNI} \).

\( \text{SME}_R \) prevents all downward flows in all \( s \) which do not announce information release through release actions. This is a corollary of Theorem [V.3] since no step of in a trace from \( s \) is derived using a rule from Figure 7.

**Corollary V.7.** \( \forall s \in \text{LTS}_{\text{IO}}^{C}, \sigma, \rho : \text{SME}_R(\rho, \sigma, s) \in \text{TSNI} \).

**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 \( \text{SME}_R(\rho, s) \) never receives the declassified value. With \( \rho = \{ (\text{H}, L) \} \), consider this \( s \).

\[
\frac{\gamma_{v} \vdash s}{s \vdash s'} \quad \frac{\gamma_{v} \vdash s}{s \vdash s'}
\]

Theorem [V.9] states that if \( s \in \text{highlead} \) and \( \rho = \{ (\text{H}, \text{L}) \} \), then \( \text{SME}_R(\rho, s) \) stops the forbidden declassification, retaining soundness. In program \( p_{d1} \), the \( \text{SME}_R(\rho, s) \) allows the declassification.

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

\[
\text{in M h; \ out L h} \quad / / p_{d2}
\]

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.

**Definition V.8.** \( s \in \text{TSNI}_{\text{mod}} \) if

\[
\forall v, \gamma_{v}, \sigma_{v}, \rho_{v} : \text{TSNI}_p \implies \text{TSNI}_{\text{mod}}
\]

\( \text{Let } \rho(s) = \{ \gamma_{v} \vdash s \} \implies \text{TSNI} \) be the releases of \( s \). If \( \rho(s) \subseteq \rho \), then \( \text{SME}_R \) will not prevent any declassifications; this enables transparency.

**Theorem V.9.** \( \forall s \in \text{LTS}_{\text{IO}}^{R} \cap \text{TSNI}_{\text{mod}}, \sigma \in \text{highlead}, \rho, \alpha \)

\( a) \quad \exists \alpha', I \models \text{F}(a) \alpha' \implies \)

\[
\exists \alpha', I \models \text{SME}_R(\rho(s), \sigma, \alpha) \alpha' \forall \ell \alpha \leq_{\sigma, 1} \alpha' \]

\( b) \quad I \models \text{F}(a) \alpha' \forall \ell \alpha \leq_{\sigma, 1} \alpha' \]

Program \( p_{d2} \) satisfies \( \text{TSNI}_{\text{mod}} \). It is easy to see that if \( \sigma \in \text{highlead} \) and \( \rho = \{ (\text{H}, \text{L}) \} \), then \( \text{SME}_R(\rho, s) \) routes the value announced by the \( \ell \)-run (which is the value received on \( M \)) to the \( \ell \)-run in the desired way, thus yielding a transparent run. When \( \rho = \emptyset \), \( \text{SME}_R(\rho, s) \) stops the forbidden declassification, retaining soundness. In program \( p_{d1} \), the \( \text{SME}_R(\rho, s) \) allows the declassification.
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.

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 (4)), 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-observable different action, we note an attack in a1. In 2), we note a timeout in a2 (which might be an attack). Input streams are constructed from traces using \( t \), defined as

\[
\left( u \left( \bar{a} \right) \right)_t = \left( \bar{a} \left| t, \alpha, \epsilon \right. \right)_0. (\epsilon \left| \epsilon \right. )^\infty.
\]

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

1) \( I_1 \models s \bar{a}_1 \), and
2) \( \forall \bar{a}_2 \cdot I_2 \models s \bar{a}_2 \implies \bar{a}_2 \notin t \bar{a}_1 \).

A. Soundness

SME\( _T \) enforces a timing-insensitive noninterference notion. This is easily seen by observing that the traces produced by \( I \models SME_T(s) \) and \( I \models s \) are \( \approx_L \)-equivalent.

Theorem VI.2. \( \forall s, \sigma, s \models SME_T(s) \in PSNI \)

By allowing the L-run to wait up to \( t \) steps for the H-run to match an L-observable, SME\( _T(s) \) introduces a timing leak into \( s \), and thus does not enforce TSNI. We note however that SME\( _T(s) \) 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 \( \bullet \) infinitely, by (conflict). A more practical approach would be to instead have the H-run behave like it would under SME\( (s) \) henceforth. We hypothesize (but do not prove) that this modification of SME\( _T \) yields a sound enforcement.

B. Transparency

Modulo \( \bullet \), SME\( _T(s) \) and \( s \) produce the same sequence of I/O (up to an attack or timeout in SME\( _T(s) \)). In contrast to e.g. Devriese and Piessens [DP10], who swap the order of outputs in the following two programs (linearization \( B \subseteq A \)),

\[
\text{out } a^4 1 ; \text{ out } B^8 1
\]

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

Theorem VI.3. \( \forall s, \sigma, I, \bar{a} \).

\( a) I \models s \bar{a} \implies \exists \bar{a}' \cdot I \models SME_T(s, \bar{a}) \rightarrow (\bar{a}', \epsilon, \epsilon) \wedge |\bar{a}'|_s \leq |\bar{a} |_s \wedge |\bar{a}'|_s \not\leq |\bar{a} |

(\bar{a}' \approx \bar{a} ) \)

\( b) I \models SME_T(s, \bar{a}) \rightarrow (\bar{a}, \epsilon, \epsilon) \implies \exists \bar{a}' \cdot I \models s \bar{a}' \rightarrow (\bar{a}', \approx \bar{a}' ) \)

It is easy to see that if \( s \in TSNI \), then SME\( _T \) never generates attacks, and thus, \( s \) and SME\( _T(s) \) have the same (that is, \( \approx_L \)-equivalent) I/O behavior.

C. Attacks

Any match deviation found by SME\( _T(s) \) (before a timeout), forms the basis of a concrete proof that \( s \notin PSNI \).

Theorem VI.4. \( \forall s, \sigma, I, I \models SME_T(s, \bar{a}) \rightarrow s \models (\bar{a}, \alpha, \epsilon) \implies s \models \alpha \) is an L-attack on \( s \).
If a timeout (a₁₂) is discovered before the discrepancy (a₁), 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 SME₇ finds on them. Consider

\begin{align*}
\text{in} & \quad M \quad \text{h}; \quad \text{out} & \quad L \quad \text{h} \quad \text{if} \quad d = 0, \quad t = 100 \quad \text{and initial input} 1, \quad \text{SME₇}(s) \quad \text{generates an} \quad \text{attack} \quad (L, I₁, I₂, M?d.L!0) \quad \text{on} \quad s \quad \text{in} \quad a₁, \quad \text{where} \quad I₁(M) = M?d.(M?*)∞, \quad I₂(M) = M?1.(M?*)∞. \quad \text{Now consider} \\
\text{in} & \quad M \quad \text{h}; \quad \text{while} \quad h \quad \text{;} \quad h = 0 \quad \text{h} = h - 1 \quad \text{if} \quad \text{out} & \quad L \quad \text{0} \quad \text{d = 0, t = 100} \quad \text{and initial input} -1, \quad \text{SME₇}(s) \quad \text{generates an} \quad \text{attack} \quad (L, I₁, I₂, M?d.L!0) \quad \text{on} \quad s \quad \text{in} \quad a₁₂, \quad \text{where} \quad I₁(M) = M?d.(M?*)∞, \quad I₂(M) = M?1.(M?*)∞. \quad \text{With initial input 500, however, an attack is generated which is not an attack on} \quad s \quad \text{(it just took “too long” for the} \quad h \text{-run to match L!0).}
\end{align*}

VII. RELATED WORK

Referring the reader for general overviews on language-based information-flow security [SM03], on dynamic information-flow control [Que07], 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 fₜ(h, l) and a ‘low security part’ fₙ(l) that does not use any of the high-level inputs h. As a result, noninterference can be proven by transforming the program into a special form that does not depend on the high-level inputs. They propose relaxed noninterference that allows information release through a set of defined 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.

Biellová 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. 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


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

APPENDIX


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 in Listing 1.

Listing 1: FlowFox policy

```javascript
function leak() {
  var d = new Date();
  var n = d.getTime();
  var x = n-m;
  var s = new Image();
  s.src = "http://attacker.com?v=" + encodeURIComponent(x);
}
</script>
<head><head>
<body>One-bit timing leak</body>
</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 [GDP12]. 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.