Journal of Computer Security 0 (2014) 0
IOS Press

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

Willard Rafnsson[†], Andrei Sabelfeld[‡]
[†] *Carnegie Mellon University, Pittsburgh PA, USA*
[‡] *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 disable any of 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 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 and where it 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.

Keywords: information flow, dynamic enforcement, secure multi-execution, noninterference, transparency

1. Introduction

As modern attacks are becoming more sophisticated, there is an increasing demand for protection measures more advanced than those offerred by standard security practice. We demonstrate the problem with a motivating scenario from web application security, but note that the problem is of a 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 [50]. The tracking was used for targeted 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 [35] 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.

A representative real-life example is the defacement of the Reuters site in June 2014 [40], attributed to "Syrian Electronic Army", which compromised a third-party widget (Taboola). This shows that even established content delivery networks risk being compromised, and these risks immediately extend to all web sites that include scripts from such networks.

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 [34], FlowCaml [49], and SparkAda Examiner [10] that enhance compilers for Java, Caml, and Ada, respectively. Recently, dynamic monitoring techniques have received increased attention (cf. [28,27,48,46,5,6,24]), 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 [43] 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 [19,12,26,25,7,11,22], a runtime enforcement mechanism 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 level 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 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 $\ell \sqsubseteq \ell'$ [18]. 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* [21] demands that low outputs do not depend on high inputs.

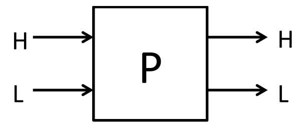

Fig. 1.: Original execution

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.

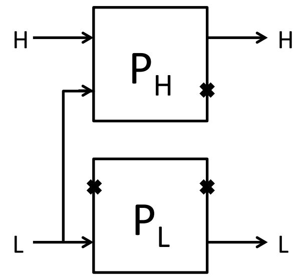

Fig. 2.: SME

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 disable any of 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 w.r.t. 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 and where it 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 counterexamples to noninterference, i.e. attacks, on run time.

This journal paper extends and improves an earlier conference version [39]. The present journal version presents a complete overhaul of Section 5 on declassification in SME, resulting in new results and substantial improvements. Notably, we give a critical evaluation, justification and motivation of our model of declassification. We clarify *full release*, our model of declassification to confine release to data of specified levels of sensitivity. This model was introduced in the conference version, and we now generalize it and develop theoretical results for it. Further, we show that our declassification model provides additional assurance: drawing on the knowledge-based *gradual release* [2,3] model, we demonstrate that information release may only take place at declassification points and nowhere else during the execution. We develop theoretical results for it, including a knowledge-based noninterference which we show is equivalent to the noninterference notion we use throughout the paper. We prove soundness for this model, establishing that our declassifying SME only leaks information through declassification actions. This is the first such result for SME. We generalize the declassification semantics to support multiple channels of declassification, enabling modeling of the semantics of a declassified value in the name of its declassification channel. For the remainder of the paper, we have improved readability by including corrections to the original paper and by revising notation and formatting. We have updated the related work section with work on SME that has appeared since our original publication. Finally, we present the full proofs for all of our technical results.

2. 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 to decide which enforcement mechanism to pick in a particular case and identify the most pressing shortcomings, subject to improvements by this paper. For a more detailed account of the state of the art, we refer the reader to the related work section. We start by listing what we view as the pros of secure multi-execution.

Noninterference by design A significant advantage of secure multi-execution is that it enforces noninterference in a straightforward manner 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 [19]). 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 holds when e.g. the willingness of a program to consume low input is independent of high information: the high run in the secure multi-execution of a program performs the same inputs and outputs as the program does without being securely multi-executed. 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 for channels 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 [19] 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. [12] allow non-total environments but at the price of ignoring information leaks through termination behavior (targeting termination-insensitive noninterference [53]). This implies that the leaks as in the example with the health site are still ignored because the one bit of information of whether the user has visited a heath web site is allowed to be leaked.

This motivates the need for fine-grained secure multi-execution. We remove 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 et al. [26], 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 [19] and at the heart of FlowFox [22], an extension of FireFox to enforce secure information flow in JavaScript. The security theorem in the abstract setting of secure multi-execution [19] 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 event handlers, reactive program snippets that are triggered upon occurrence of events. The low-priority scheduler is fundamentally problematic 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 [22] 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 is indeed what happens in the example program. Using `setTimeout`, the main code creates a new handler whose job is to report the time difference since the start of the program. Then the main code branches on a secret and performs a short or long computation depending on the secret. Upon the exit of the main thread, the handler is triggered to report the time difference to the attacker.

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 the view of the high run), and to input on a low-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 While not required for soundness, the executions at different levels need to make the same nondeterministic choices for secure multi-execution to be transparent. Although this has not been explicitly handled in previous work, a natural possibility is to assign security levels to the source of nondeterminism [36] 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 [19]. Also, as we discuss in Section 7, optimizations are possible for simulating multiple executions by computing on enriched values [7].

3. Framework

We lay the foundation for our technical contributions outlined in the introduction by presenting a framework for information-flow security of interactive programs [36,17,15,38,37].

3.1. Input-output labeled transition systems

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. We write $s \xrightarrow{l} s'$ 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.

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

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

Definition 3.1. An *input-output* LTS (LTS_{IO}) is an LTS $(S, L, \rightarrow)$, with L ranged by a.

Practical languages for which this model of computation is appropriate include Erlang and JavaScript. Bohannon et al. give the semantics of a JavaScript-like language as an LTS_{IO} in [15] and Clark and Hunt give the semantics of an imperative language with I/O (used in our examples) as an LTS_{IO} in [17]. Since all our results apply to any LTS_{IO} state, our contributions are general.

3.2. Interactive programs

Definition 3.1 defines a general model of interaction that places no restrictions on how an LTS_{IO} interacts with its environment. The interactive programs we consider in this paper are a class of LTS_{IO} which interact with their environment in a particular way. We formalize this interaction pattern as three properties: *input-neutral*, *input-blocking*, and *deterministic*. An input-neutral LTS_{IO} will, when it is ready to perform input on a channel, accept any value the environment provides on that channel. An input-blocking LTS_{IO} will, when attempting to perform input on a channel, block until one is available. We formalize this as an input action with a distinguished value $\star \in \mathbb{V}$ (blank); 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). A deterministic LTS_{IO} can either only do a (unique) output, or only do input on a (unique) channel, and in both cases enter a state uniquely defined for the action it took. We require that these properties are preserved through execution, i.e., hold for all reachable states. In the following, let $\mathtt{S}(s)$ be the set of states reachable from s. That is, $s \in \mathtt{S}(s)$, and for all $s' \in \mathtt{S}(s)$ we have for all a and s'' for which $s' \xrightarrow{a} s''$ that $s'' \in \mathtt{S}(s)$.

Definition 3.2. For all s_0,

1. s_0 is *input-neutral* iff $\forall s \in \mathtt{S}(s_0)\,.\,\forall c\,.\,(\exists v\,.\,s \xrightarrow{c?v}) \implies (\forall v\,.\,s \xrightarrow{c?v})$.
2. s_0 is *input-blocking* iff $\forall s \in \mathtt{S}(s_0)\,.\,\forall s', c\,.\,s \xrightarrow{c?\star} s' \implies s' = s$.
3. s_0 is *deterministic* iff $\forall s \in \mathtt{S}(s_0)\,.$
   (a) $\forall a_1, a_2\,.\,s \xrightarrow{a_1} \wedge s \xrightarrow{a_2} \wedge a_1 \neq a_2 \implies \exists c, v_1, v_2\,.\,a_1 = c?v_1 \wedge a_2 = c?v_2$, and
   (b) $\forall s_1, s_2, a\,.\,s \xrightarrow{a} s_1 \wedge s \xrightarrow{a} s_2 \implies s_1 = s_2$.

Point 1 states that if s is ready to perform input on c, s is receptive to any v on c. Point 2 states that input is a blocking operation, by asserting that s does not change state while waiting for input. Point 3a says if $s \xrightarrow{c?v}$, then $s \xrightarrow{a}$ iff $a \in \{c?v \mid v \in \mathbb{V}\}$ (so s can be input-neutral), and implicitly, if $s \xrightarrow{o}$, then $s \xrightarrow{a}$ iff $a = o$. Point 3b says s has no internal nondeterminism. Unless stated otherwise, any s we consider in this paper satisfies Points 1, 2 and 3. We discuss the assumption of Point 2 further in Section 4.

3.3. Traces

We will reason about the behavior or systems in terms of the sequences of actions the system can perform. We therefore formally define such sequences, and ways of comparing them, now. A trace is a (finite) *list* of actions, denoted $\bar{a}$. We let ϵ denote the empty trace, "." as the cons operator, and we usually omit ϵ in nonempty traces. 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. All of these, and other, projections used throughout the paper are given formally in Appendix B. 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 $\bar{a} \leq \bar{a}''$ when, for some $\bar{a}'$, $\bar{a}'' = \bar{a}.\bar{a}'$. Here, $\bar{a}.\bar{a}'$ is the concatenation of $\bar{a}$ and $\bar{a}'$. Throughout the paper, all the relations on traces that we use are defined as $\leq$ or $=$ after first applying projection functions in a particular order on both sides of the relation. To reduce notation, instead of using a new symbol for each such combination of relation and projection functions that we use, we develop a notation for applying projection functions in order to both sides of a relation. 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, $\bar{a} \leq_{!,c} c?3.c!4.c!5$.

3.4. 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; $\pi(c)$ is the level of the *presence* of a message on c, and $\kappa(c)$ is the level of the *content* or *value* of a message on c. We require for all c that $\pi(c) \sqsubseteq \kappa(c)$, the intuition being that an observer who can infer the value of a message can infer that there was a message carrying this value. In examples, we frequently represent a channel by its security labels; we then write $\kappa(c)^{\pi(c)}$ in place of c (in code, $\kappa(c)\pi(c)$). Note, however, that we do not assume that there is only one channel associated with each security level. A classic example is the two-level lattice $\mathcal{L}_{\mathtt{LH}} = \{\mathtt{L}, \mathtt{H}\}$ with $\sqsubseteq = \{(\mathtt{L}, \mathtt{L}), (\mathtt{L}, \mathtt{H}), (\mathtt{H}, \mathtt{H})\}$, L for "low" confidentiality, H for "high". We let H, M, L denote $\mathtt{H}^{\mathtt{H}}$, $\mathtt{H}^{\mathtt{L}}$ and $\mathtt{L}^{\mathtt{L}}$, resp.. Since our results apply to any lattice, our contributions are general. In our technical results, we assume a fixed lattice.

Figure 3 illustrates the flow of information in the case of $\mathcal{L}_{\mathtt{LH}}$. Input on M is depicted in-between the H and L input. The presence of such an input is L (this dependency on L is illustrated by the dashed line) while the content is H (this dependency on H is illustrated by the solid line). Similarly, output on M is depicted in-between the H and L output. Its presence is observable at L level (dashed arrow), and its value is observable at H level (solid arrow).

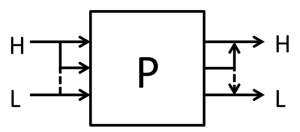

Fig. 3.: Original execution with fine-grained security

The security labels express *who* can observe *what*. An observer is associated with a security level ℓ. An ℓ-observer is capable of observing the presence (resp. content) of a message on c iff $\pi(c) \sqsubseteq \ell$ (resp. $\kappa(c) \sqsubseteq \ell$). $\bar{a}\restriction_\ell$ removes ℓ-unobservable parts of actions in $\bar{a}$. For $\bar{a} = \bullet.\mathtt{L}?0.\mathtt{H}!1.\mathtt{M}?\star.\mathtt{M}!2.\mathtt{H}?\star.\bullet$, $\bar{a}\restriction_{\mathtt{L}} = \bullet.\mathtt{L}?0.\bullet.\mathtt{M}?\mathtt{d}.\mathtt{M}!\star.\bullet.\bullet$. Here, H!1 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 $\mathtt{d} \in \mathbb{V}$), since a L-observer only observes presence of messages on M (all $a \in \{\mathtt{M}!v \mid v \in \mathbb{V}\}$ 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. Trace equivalence defines the class of attackers such a property guarantees security against. We consider two classes of attackers: timing- and progress-sensitive ones. Since each action takes a unit of time, these two attackers differ in whether or not they observe non-interaction. Since no message is passed when a system is blocking on input, we treat blank input as non-interaction. $\bar{a}\restriction_\star$ replaces all $c?\star$ with $\bullet$. With $\bar{a}$ defined as above, $\bar{a}\restriction_\star = \bullet.\mathtt{L}?0.\mathtt{H}!1.\bullet.\mathtt{M}!2.\bullet.\bullet$.

A *timing-sensitive* (e.g. [1,19]) attacker measures time between observables in a trace. $\bar{a}\restriction_{\vec{\bullet}}$ removes trailing $\bullet$ from $\bar{a}$. E.g. $\bar{a}\restriction_{\vec{\bullet}} = \bullet.\mathtt{L}?0.\mathtt{H}!1.\mathtt{M}?\star.\mathtt{M}!2.\mathtt{H}?\star$. Define *timing-sensitive ℓ-equivalence* $\simeq_\ell$ as $=_{\star,\ell,\vec{\bullet}}$ (see Section 3.3 for a definition of $=_{\star,\ell,\vec{\bullet}}$). Let $\bar{a}_1 = \mathtt{H}?\star.\mathtt{H}?1.\mathtt{L}!0$ and $\bar{a}_2 = \mathtt{H}?1.\mathtt{L}!0$, and consider

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

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

A *progress-sensitive* (e.g. [36,3,38]) attacker observes whether more observables are forthcoming. $\bar{a}\restriction_\bullet$ removes all $\bullet$ from $\bar{a}$. With $\bar{a}$ as above, $\bar{a}\restriction_\bullet = \mathtt{L}?0.\mathtt{H}!1.\mathtt{M}?\star.\mathtt{M}!2.\mathtt{H}?\star$. 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; $\bar{a}_1\restriction_{\star,\mathtt{L},\bullet} = \mathtt{L}!0 = \bar{a}_2\restriction_{\star,\mathtt{L},\bullet}$ (in both traces, L!0 eventually appears). A progress-sensitive timing-insensitive attacker is strictly weaker than a timing-sensitive one since $(\simeq_\ell) \subsetneq (\approx_\ell)$.

3.5. Environments

Inputs to our systems come from the environment in which our systems run. Clark and Hunt [17] have demonstrated that when analyzing *deterministic* programs for security, an environment does not need to be adaptive to provoke a particular (leaking) behavior. Thus it is sufficient *for our purposes* to consider environments represented as a *stream* (infinite list) of inputs for each channel. So, an environment e 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 driven by varying the timing of input.

The jth element in $e(c)$ represents what e provides on c in time unit j; if this element is $c?v$, then e provided value v on c in time unit j; if the element is $c?\star$, then e provided no value in time unit j. Since any program we consider can consume at most one input per computation step, we do not need to consider scenarios (or their security implications) where e provides input at a rate faster than the program can consume them. Since a program is not necessarily ready to receive an input when the environment provides it, we assume that the interface between e and the program manages a buffer for each input channel, and that the program draws input on a channel c from the c-buffer (in FIFO order), drawing a blank when the c-buffer is empty. We also assume that this interface never prevents a program from performing an output. The interaction between a program and its environment is therefore asynchronous.

We formalize this interaction using a relation that defines when a trace $\bar{a}$ is possible under a given environment e. Intuitively, $\bar{a}$ is possible under e if, for any c, *1)* the c-inputs in $\bar{a}$ are (in value and order) as provided by e, *2)* each c-input occurs no sooner in $\bar{a}$ than e could have provided it, and *3)* a blank is drawn only when no c-input is buffered. Let $e_c = e(c)$, and let $e_{c,n}$ denote the prefix of length n of e_c. Formally, $\bar{a}$ is *consistent* with e, written $e \models \bar{a}$, iff for all c, we have for all $\forall \bar{a}' \leq \bar{a}$, with $\bar{i}' = e_{c,|\bar{a}'|}$, that

$$\bar{a}' \neq \_.c?\star \implies \bar{a}' \leq_{?,c,\star,\bullet} \bar{i}', \text{ and}$$

$$\bar{a}' = \_.c?\star \implies \bar{a}' =_{?,c,\star,\bullet} \bar{i}' \wedge \bar{i}' = \_.c?\star.$$

This states that for each prefix $\bar{a}'$ of $\bar{a}$ and each c, the list of non-blank inputs in $\bar{a}'$ must prefix the list of non-blank inputs in the same-length prefix $\bar{i}'$ of $e(c)$, and that $\bar{a}'$ ends with a blank input on c only if all c-inputs provided by e up to this point are in $\bar{a}'$ and e provided no new c-inputs in time unit $|\bar{a}'|$ ($\_$ is a wildcard). The quantification and definition of $\bar{a}'$ and $\bar{i}'$ formalizes *2)*, the trace equivalences formalize *1)*, and the $\bar{a}' = \_.c?\star$ case addresses *3)*. Since running s under e constrains the traces which s can perform, we obtain a definition of interaction as follows: *s performs $\bar{a}$ under e*, written $e \models s \xrightarrow{\bar{a}}$, iff $s \xrightarrow{\bar{a}}$ and $e \models \bar{a}$. Let s be the initial state of the following program.

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

Let $e_{1c}, \ldots, e_{5c}$ be defined as on the left below. Then the judgments on the right hold.

$$e_{1c} = (c?\star)^{\infty} \qquad e_1 \models s \xrightarrow{(c?\star)^n}, \text{ for all } n \in \mathbb{N}$$

$$e_{2c} = c?0.c?0.(c?\star)^{\infty} \qquad e_2 \models s \xrightarrow{c?0.\bullet.c?0}$$

$$e_{3c} = c?2.c?0.(c?\star)^{\infty} \qquad e_3 \models s \xrightarrow{c?2.\bullet.\bullet.\bullet.\bullet.\bullet.c?0}$$

$$e_{4c} = c?0.c?\star.c?\star.c?\star.c?0.(c?\star)^{\infty} \qquad e_4 \models s \xrightarrow{c?0.\bullet.c?\star.c?\star.c?0}$$

$$e_{5c} = c?2.c?\star.c?\star.c?\star.c?0.(c?\star)^{\infty} \qquad e_5 \models s \xrightarrow{c?2.\bullet.\bullet.\bullet.\bullet.\bullet.c?0}$$

These judgments hold regardless of how $e_{kc'}$ for $1 \leq k \leq 5$ and $c' \neq c$ are defined (s only reads from c).

Equivalence We compare the streams in the environments the same way we compare traces. So we define observational equivalence of environments as follows:

$$e_1 \simeq_\ell e_2 \text{ iff } \forall c \,.\, \pi(c) \sqsubseteq \ell \implies e_1(c) \simeq_\ell e_2(c).$$

$$e_1 \approx_\ell e_2 \text{ iff } \forall c \,.\, \pi(c) \sqsubseteq \ell \implies e_1(c) \approx_\ell e_2(c).$$

Totality An environment is total if it always provides a system with input whenever the system needs it. In our framework, e is total if $\star$ does not occur in any e_c. Previous work on security for interactive programs [36,17] assume environments are total. However, as we have demonstrated previously [37], this assumption limits (undesirably) the space of possible attacks on input-blocking interactive programs, since the presence of a message can depend on high data. Consider the program in Section 3.4. Let $e_{1\mathsf{H}} = \mathsf{H}?0.(c?\star)^\infty$ and $e_{2\mathsf{H}} = e_{1c} = e_{2c} = (c?\star)^\infty$ for all $c \neq \mathsf{H}$. While this program can perform $\mathsf{H}?0.\mathsf{L}!0$ under e_1, the program cannot perform an $\approx_\mathsf{L}$-equivalent trace under e_2. Since a program can encode a bit in the presence of a message, these attacks become crucial in an interactive setting. To emphasise the gravity of this, we give an example, from [37], of three interactive programs, each secure under total environments, which, when run in parallel (e.g. with the semantics in [37, Figure 2]), 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 summary, 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 e) must be considered. This in part motivates our fine-grained security levels; since no low observables are allowed to occur after a high input in deterministic input-blocking systems, the only way for such a system to input a high value before performing low observables is if the presence level of the input is low [37].

3.6. Noninterference

As mentioned earlier, our target policy is noninterference. The idea behind noninterference is the following. Assume an ℓ-observer observes all system effects which he is privileged to observe, i.e. all ℓ-observable actions. A system is *noninterfering* if, by observing ℓ-observable actions, the ℓ-observer learns nothing he is not privileged to learn, i.e., unobservable input does *not interfere* with observable behavior. Noninterfering systems are thereby not responsible for leaks. The formalization of this idea we use is that of *possibilistic noninterference* for interactive systems [37,17,36]. It states that, for any two ℓ-equivalent input environments, the respective sets of traces produced under either of them are ℓ-equivalent. Thus, an ℓ-observer, by observing ℓ-observables, cannot tell one input environment apart from any other input environment that differs only in ℓ-unobservable input. The following definition is a definition schema, parameterized by an security-level-indexed equivalence relation $\mathcal{R}$ defined on traces and environments. We will later instantiate this schema with $\simeq$ and $\approx$ (both security-level-indexed equivalence relations on traces and environments) in place of $\mathcal{R}$.

Definition 3.3. *s is $\mathcal{R}$-noninterfering* ($s \in \text{NI}^{\mathcal{R}}$) iff
$\forall \ell, e_1, e_2 \,.\, e_1 \,\mathcal{R}_\ell\, e_2 \implies \forall \bar{a}_1 \,.\, e_1 \models s \xrightarrow{\bar{a}_1} \implies \exists \bar{a}_2 \,.\, e_2 \models s \xrightarrow{\bar{a}_2} \wedge\, \bar{a}_1 \,\mathcal{R}_\ell\, \bar{a}_2.$

Note that for nondeterministic programs, possibilistic noninterference is a somewhat weak notion of security, since it requires that a system can, by making the appropriate nondeterministic choices, match behavior. Since our systems are deterministic, however, the *stream* of actions a system can perform under a fixed e is unique. Possibilistic noninterference thus requires that, for any $e_1 \,\mathcal{R}_\ell\, e_2$, any prefix of *the* stream of actions s performs under e_1 must be matched by some prefix of *the* stream of actions s performs under e_2, *and vice versa*, since $e_2 \,\mathcal{R}_\ell\, e_1$. Possibilistic noninterference (due to limited possibilities) is thus rather strong in our setting.

By instantiating this definition with the two ℓ-equivalence relations from Section 3.4, we get the two definitions of noninterference we will consider in this paper.

Definition 3.4. *s is timing-sensitive, progress sensitive noninterfering* ($s \in \text{TSNI}$) iff $s \in \text{NI}^{\simeq}$.

Definition 3.5. *s is timing-insensitive, progress sensitive noninterfering* ($s \in \text{PSNI}$) iff $s \in \text{NI}^{\approx}$.

For the system models under consideration in this paper, PSNI is a weakening of TSNI. The details of this, and all other proofs, are in Appendix C.

Theorem 3.6. $\forall s \,.\, s \in \text{TSNI} \implies s \in \text{PSNI}.$

Note that this result (and others) is sensitive to the assumptions we make in Definition 3.2. Consider a system s that times the arrival of an input on a L input channel (by counting $\star$s), and outputs the time on a L channel. Such a system is TSNI-secure, and not input blocking. However, under $\approx_{\text{L}}$-equivalent environments, the timing of L input is allowed to differ; under such environments, s leaks the timing difference into a value in L output, which a progress-sensitive observer can observe; a PSNI-insecurity.

4. Fine-grained secure multi-execution

The opening series of our contributions develops a generalization of SME [19] with respect to several dimensions. We lift the assumption on the totality of the input environment (i.e. that there is always assumed to be input) and on cooperative scheduling. Furthermore, we distinguish between security levels of presence and content of messages. In addition, we generalize SME to arbitrary deterministic LTS_{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 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).

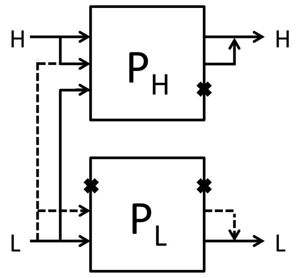

Fig. 4.: SME with fine-grained security

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 $\pi(c)$-run can input on c. An ℓ-run which can consume a c-input with $\pi(c) \not\sqsubseteq \ell$ 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 $\pi(c) \sqsubset \ell$ gets a copy of the nth input consumed by the $\pi(c)$-run, unless $\pi(c)$ is yet to consume n c-inputs, in which case the ℓ-run blocks until the $\pi(c)$-run has done so. However, if $\kappa(c) \not\sqsubseteq \ell$, 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 $\pi(c)$-run can output on c. A c-output produced by a ℓ-run for which $\pi(c) \neq \ell$ is discarded by SME (as opposed to being sent to the environment). When an ℓ-run produces its nth c-output with $\ell = \pi(c)$, this output is sent straight to the environment, except when $\pi(c) \neq \kappa(c)$; in that case SME first checks whether the $\kappa(c)$-run has 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 $\kappa(c)$-run. Otherwise, the value is $\mathtt{d}$.

4.1. Semantics

We now give a semantics for the SME of an arbitrary s satisfying the assumptions in Definition 3.2, as an $\mathrm{LTS}_{\mathrm{IO}}$ state. Concurrent executions of ℓ-runs of s are scheduled by a *scheduler*. We consider schedulers which are autonomous and independent of input and of the behavior of ℓ-runs, since letting scheduling decisions depend on these can make SME susceptible to subtle timing attacks (as demonstrated by Kashyap et al. [26], and discussed further in Section 4.2). Furthermore, to make SME transparent, we require that schedulers are *fair*, in the sense that the scheduler schedules all ℓ-runs infinitely often, and *deterministic*, in the sense that the sequence of scheduling decisions the scheduler makes is unique.

Definition 4.1. A *scheduler* LTS, $\mathrm{LTS}_{\mathrm{S}}$, is an LTS with labels ranged by ℓ. A *scheduler*, σ, is a state of an $\mathrm{LTS}_{\mathrm{S}}$. For all σ_0,

1. σ_0 is *deterministic* iff $\forall \sigma \in \mathsf{S}(\sigma_0) \,\mathbf{.}$
   (a) $\forall \ell, \sigma_1, \sigma_2 \,\mathbf{.}\, \sigma \xrightarrow{\ell} \sigma_1 \wedge \sigma \xrightarrow{\ell} \sigma_2 \implies \sigma_1 = \sigma_2$, and
   (b) $\forall \ell, \ell' \,\mathbf{.}\, \sigma \xrightarrow{\ell} \wedge \sigma \xrightarrow{\ell'} \implies \ell = \ell'$
2. σ_0 is *fair* iff $\forall \sigma \in \mathsf{S}(\sigma_0) \,\mathbf{.}$
   $*$ $\forall \bar{\ell} \,\mathbf{.}\, \sigma \xrightarrow{\bar{\ell}} \implies \forall \ell \,\mathbf{.}\, \exists \bar{\ell}' \,\mathbf{.}\, \sigma \xrightarrow{\bar{\ell}.\bar{\ell}'.\ell}$.

Point 1a states that the way future decisions is made is uniquely defined by the next choice, and Point 1b stipulates that the choice of which ℓ-run to schedule next is unique. Point 2 states that no matter how many scheduling decisions have been made, each ℓ-run will eventually be scheduled. Unless stated otherwise, σ is deterministic and fair. Thus, since ℓ and σ' for which $\sigma \xrightarrow{\ell} \sigma'$ are unique and always exist, we use list and stream notation to represent and manipulate schedulers. An example of deterministic and fair schedulers is the *round-robin schedulers*. For instance, for $\mathcal{L}_{\mathtt{LH}}$, $(\mathtt{H}.\mathtt{L})^{\infty}$ and $(\mathtt{L}.\mathtt{H})^{\infty}$ are deterministic fair schedulers (that repeat $\mathtt{H}.\mathtt{L}$ resp. $\mathtt{L}.\mathtt{H}$, infinitely).

The semantics of SME is an $\mathrm{LTS}_{\mathrm{IO}}$ of SME-states. 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 transition relation for the $\mathrm{LTS}_{\mathrm{IO}}$ is given in Figure 5. The transition relation is presented in three layers, in Figure 5c, Figure 5b and Figure 5a respectively. Each rule in each layer uses only rules in the layer directly above it. We start with the bottom-most layer, Figure 5c, and work our way upwards. The derivation of any SME transition begins with the (history) rule (the judgment in the premise, which we will turn to momentarily, is defined in Figure 5b). The purpose of (history) is to keep track of the interaction $\bar{a}$ which $\mathsf{SME}(\sigma, s)$ has had with the environment. Note

$$\frac{s \not\rightarrow}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{\bullet} (\bar{a}_\ell.\bullet, s)}\text{dead} \qquad \frac{s \xrightarrow{\bullet} s'}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{\bullet} (\bar{a}_\ell.\bullet, s')}\text{silent}$$

$$\frac{s \xrightarrow{c?v} s' \quad \bar{a}_\ell.c?v \leq_{\star,\ell,\bullet,?,c} \bar{a} \quad \textbf{if } \kappa(c) \not\sqsubseteq \ell \textbf{ then } v = \mathtt{d} \textbf{ else } v \neq \star}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{\bullet} (\bar{a}_\ell.c?v, s')}\text{i-old} \qquad \frac{s \xrightarrow{c!v} s' \quad \pi(c) \neq \ell}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{\bullet} (\bar{a}_\ell.c!v, s')}\text{o-old}$$

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

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

(a) SME ℓ-stepper

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

$$\frac{\begin{array}{c}(\bar{a},\ell) \models S(\ell) \xrightarrow{c?v} \qquad\qquad \pi(c) = \ell \\ \forall \ell' \,.\, \textbf{if } \ell \sqsubseteq \ell' \wedge (\bar{a},\ell') \models S(\ell') \xrightarrow{c?v} (\bar{a}', s') \textbf{ then } S'(\ell') = (\bar{a}', s') \textbf{ else } S'(\ell') = S(\ell')\end{array}}{\bar{a} \models (\ell.\sigma, S) \xrightarrow{c?v} (\sigma, S')}\text{i}$$

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

(b) SME ℓ-chooser

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

(c) SME history

Fig. 5. Semantics of SME

that $\mathsf{SME}(\sigma, s) \xrightarrow{\bar{a}} (\bar{a}', \sigma', s') \implies \bar{a} = \bar{a}'$, 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 $\bar{a}$ (we apply the same convention throughout the paper; this use of symbol "$\models$" is not to be confused with the use of "$\models$" in Section 3.5). The rules at the Figure 5b-layer are responsible for, by use of σ, deciding which ℓ-run takes a step next, using the rules in the third (i.e. top) layer, Figure 5a. This third layer is responsible for hiding from the Figure 5b-layer all the different ways which an ℓ-run can take a step without interacting with the environment ((dead), (silent), (o-old), (i-old)), and signaling to the Figure 5b-layer when the ℓ-run requires I/O with the environment to proceed ((o-new), (i-new)). Each rule at the Figure 5a-layer appends to the ℓ-run trace the action the ℓ-run performed during the step (which is not necessarily the same action as the one performed by the whole 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. [26]). The Figure 5a-layer stores output on channels with presence $\neq \ell$ without forwarding it to the environment, as per (o-old). (i-old) covers multiple scenarios for the ℓ-run performing an input action on a channel which does not result in the SME-state consuming input from the environment on c in this step. When $\pi(c) \not\sqsubseteq \ell$, input $\mathtt{d}$. When $\pi(c) \sqsubseteq \ell$, this rule is only applicable if the ℓ-run has not already read all the c-inputs which the $\pi(c)$-run has read. When $\kappa(c) \sqsubseteq \ell$, input the same value received from the environment when the

$\pi(c)$-run performed the corresponding input action. Otherwise, input d instead. (i-new) indicates that the ℓ-run requires input to proceed, and for the value v received from the Figure 5b-layer, indicated on the transition label, instead feeds d to the ℓ-run iff $v \neq \star \wedge \kappa(c) \not\sqsubseteq \ell$ (v_ℓ is a function of c, ℓ, and v). The previous layer has two rules for this scenario. When $\pi(c) \sqsubset \ell$, then the ℓ-run blocks until the $\pi(c)$-run reads on c, by (i-block). When $\pi(c)$ reads on c, the input is fed to every $\sqsupseteq \pi(c)$-run blocking on c, by (i). (o-new) notifies the Figure 5b-layer that the ℓ-run has a fresh output for the environment. Rule (o) in the previous layer handles this scenario, checking if the $\kappa(c)$-run has already provided content for this output, and if so, replaces the value in the output with the value in the corresponding $\kappa(c)$-run c-output.

When $\kappa(c) \neq \pi(c)$, the output value is replaced by d iff $\kappa(c)$ has not yet produced the corresponding value. Having the $\pi(c)$-run instead wait for the $\kappa(c)$-run to reach the corresponding c-output, or giving responsibility of producing the c-output to the $\kappa(c)$-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 H-run to produce the M-output (for nonnegative h), and whether H produces the output at all (for negative h), depends on h, in the SME of this program.

While Figure 5b indicates that SME controls each step of each ℓ-run, in practice the responsibility of SME can be distributed to the ℓ-runs, such that each ℓ-run is autonomous, as follows. Each ℓ-run is made responsible for environment I/O on all $c \in \pi^{-1}(\ell)$, since $\mathsf{SME}(s)$ performs I/O iff the ℓ-run of s performs it. Each ℓ-run makes input on each $c \in \pi^{-1}(\ell)$ and output on each $c \in \kappa^{-1}(\ell)$ available in a shared resource (e.g. memory) such that $\sqsupset$ ℓ-runs can obtain a copy of the input when they need it, and an $\pi(c)$-run can obtain the actual value to output on c. Each ℓ-run processes (after sharing, when $\ell = \pi(c)$) d in place of the inputted value when $\pi(c) \sqsubseteq \ell$ and $\kappa(c) \not\sqsubseteq \ell$, and outputs d when the $\kappa(c)$-run is yet to share the value to put into the output when $\ell = \pi(c) \sqsubset \kappa(c)$. This approach is taken in a SME benchmark by Devriese and Piessens [19]. Forcing ℓ-runs to diverge and recording full traces can be avoided [26,19]. 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 $c \in \kappa^{-1}(\ell)$ cannot impede 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 e 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 [19] is input blocking; if an ℓ-run is scheduled before a ℓ'-run with $\ell \not\sqsubseteq \ell'$, the nonpresence of input on $c \in \pi^{-1}(\ell)$ can interfere with the ℓ'-run. This hinders sound scheduling of runs for arbitrary nonlinear lattices.

4.2. Soundness

By design, SME enforces timing-sensitive noninterference.

Theorem 4.2. $\forall \sigma, s \,.\, \mathsf{SME}(\sigma, s) \in \text{TSNI}$.

We first highlight the implication of this result by contrasting it with similar results in related work. Afterwards, we give an intuition for why this theorem is true.

In contrast to Devriese and Piessens [19], who prove soundness for a cooperative scheduler for linear lattices, and to Kashyap et al. [26], 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 linearization of a nonlinear lattice, Kashyap et al. have shown that $\texttt{select}_{\text{lowprio}}$ introduces a timing dependency between ℓ-runs at incomparable levels in nonlinear lattices. For instance, with $\mathcal{L} = \mathcal{L}_{\texttt{AB}} \overset{\text{def}}{=} \{\texttt{H}, \texttt{A}, \texttt{B}, \texttt{L}\}$ and $\sqsubseteq$ being the reflexive transitive closure of $\{(\texttt{L}, \texttt{A}), (\texttt{L}, \texttt{B}), (\texttt{A}, \texttt{H}), (\texttt{B}, \texttt{H})\}$, with linearization $\texttt{L} \sqsubseteq \texttt{A} \sqsubseteq \texttt{B} \sqsubseteq \texttt{H}$ and $\texttt{d} = 0$, the time it takes for $\texttt{B}^{\texttt{B}}!1$ to emerge from the SME of the following program is, under $\texttt{select}_{\text{lowprio}}$, a function of the input on $\texttt{A}^{\texttt{A}}$.

```
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; in the SME of the following program under $\texttt{select}_{\text{lowprio}}$, the presence of input on $\texttt{A}^{\texttt{A}}$ leaks to $\texttt{B}^{\texttt{B}}$.

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

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

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

We show in Appendix A that assuming that any ℓ-run can run to completion of all of its inputs (necessary for $\texttt{select}_{\text{lowprio}}$ to schedule ℓ-runs soundly for linear lattices) is problematic when program input arrives arbitrarily in time.

The proof of Theorem 4.2 is a corollary of the following lemma, which can be obtained by removing the last two elements in the conjunction in the conclusion of the lemma, and comparing the result with Definition 3.4. We write $S_1 =_\ell S_2$ iff $\forall \ell' \sqsubseteq \ell \,.\, S_1(\ell') = S_2(\ell')$.

Lemma 4.3. $\forall \sigma, s, \ell, e_1, e_2 \,.\, e_1 \simeq_\ell e_2 \implies$
$\forall \bar{a}_1, \sigma_1, S_1 \,.\, e_1 \models \mathsf{SME}(\sigma, s) \to (\bar{a}_1, \sigma_1, S_1) \implies$
$\exists \bar{a}_2, \sigma_2, S_2 \,.\, e_2 \models \mathsf{SME}(\sigma, s) \to (\bar{a}_2, \sigma_2, S_2) \wedge$
$\bar{a}_1 =_\ell \bar{a}_2 \;\wedge\; S_1 =_\ell S_2 \;\wedge \sigma_1 = \sigma_2$

Such a strong correspondence is achievable since, for each $\ell' \sqsubseteq \ell$, the ℓ'-run of s, in $e_1 \models \mathsf{SME}(\sigma, s)$ and $e_2 \models \mathsf{SME}(\sigma, s)$, behaves as $e\!\upharpoonright_{\ell'} \models s$, where

$$(e\!\upharpoonright_{\ell'})(c) = \begin{cases} (c?\texttt{d})^\infty & \text{, if } \pi(c) \not\sqsubseteq \ell' \\ e(c)\!\upharpoonright_\ell & \text{, otherwise.} \end{cases}$$

(While for $\pi(c) \sqsubseteq \ell'$, the number of $c?\star$ preceding a $c?v$ of an ℓ'-run in $e_j \models \mathsf{SME}(\sigma, s)$ compared to $e\!\upharpoonright_{\ell'} \models s$ can differ due to σ, this number is the same in $e_1 \models \mathsf{SME}(\sigma, s)$ compared to $e_2 \models \mathsf{SME}(\sigma, s)$. 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 $e\!\upharpoonright_{\ell'}$). Thus, since $e_1 \models \mathsf{SME}(\sigma, s)$ and $e_2 \models \mathsf{SME}(\sigma, s)$ are both run under the same σ, we have that 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.

4.3. Transparency

We show that SME does not adversely modify the I/O behavior of a *progress*-sensitive noninterfering program (and therefore also a of timing-sensitive noninterfering program). Let $s \in$ PSNI, ℓ, e and σ be arbitrary. In s and $\mathsf{SME}(\sigma, s)$, the interaction on ℓ-presence channels is ℓ-equivalent.

Theorem 4.4. $\forall \sigma, s \in \textsc{psni}, e, \bar{a}$.

$$\begin{array}{lll} a)\ e \models s \xrightarrow{\bar{a}} & \Longrightarrow \exists \bar{a}' \,.\, e \models \mathsf{SME}(\sigma, s) \xrightarrow{\bar{a}'} & \wedge \forall \ell \,.\, \bar{a} \leq_{\star,\ell,\pi^{-1}(\ell),\bullet} \bar{a}' \\ b)\ e \models \mathsf{SME}(\sigma, s) \xrightarrow{\bar{a}} & \Longrightarrow \exists \bar{a}' \,.\, e \models s \xrightarrow{\bar{a}'} & \wedge \forall \ell \,.\, \bar{a} \leq_{\star,\ell,\pi^{-1}(\ell),\bullet} \bar{a}' \end{array}$$

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 [19] which establishes only the *a)*-part of Theorem 4.4 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 $\mathcal{L}_{\mathtt{AB}}$ and nontotal environments, $\mathtt{select}_{\text{lowprio}}$ 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 $\kappa(c) \sqsupset \pi(c)$, $\mathsf{SME}(\sigma, s)$ might replace the value in its corresponding output with $\mathtt{d}$. Thus, the timing behavior of $s \in \textsc{psni}$ can impede the ability of a σ to *soundly* schedule runs in $\mathsf{SME}(\sigma, s)$ *such that* the $\kappa(c)$-run reaches the output before the $\pi(c)$-run does irrespective of previously inputted values. In $\textsc{tsni}$ programs, however, all ℓ-runs for which $\pi(c) \sqsubseteq \ell$ will reach an output on c after the same number of reduction steps. This includes the $\kappa(c)$-run, since $\pi(c) \sqsubseteq \kappa(c)$. Thus, if we ensure that any ℓ-run never "outruns" its parent-runs (in the sense that the ℓ-run has made more progress on its (nonblocking) actions than $\ell \sqsupset$-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 $\mathtt{L}$-presence input (we do not need to worry about a $\mathtt{H}$-run blocking on $\mathtt{H}$-presence input, because for a deterministic program to satisfy $\textsc{psni}$ (and therefore $\textsc{tsni}$), it must be impossible for any $\mathtt{L}$ effect to follow a $\mathtt{H}$-presence input [37]). For instance, the $\textsc{sme}$ of the following program under any σ satisfying $\mathtt{H}.(\mathtt{H})^n.\mathtt{L}.\mathtt{L} \leq \sigma$ for *any* $n > 1$ is not transparent.

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

What we need is a schedule which ensures that when the $\mathtt{L}$-run reaches an action involving interaction on a $\mathtt{L}$-presence channel, $\mathtt{H}$ is either already blocking on input on said channel (in case the $\mathtt{L}$-run is performing input on it), or has already moved past the output on said channel (in case the $\mathtt{L}$-run is performing output on it). We achieve this by making sure that, before scheduling $\mathtt{L}$, we have already scheduled $\mathtt{H}$ *since* $\mathtt{L}$ was last scheduled. The following predicate formalizes this; when invoked as $\phi(\ell_{\mathtt{H}}, \ell_{\mathtt{L}}, 0, \bar{\ell})$, the predicate yields 1 when $\ell_{\mathtt{L}}$ and $\ell_{\mathtt{H}}$ have been scheduled in such a manner in $\bar{\ell}$, and 0 otherwise.

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

This formalization achieves this effect by walking through $\bar{\ell}$, assigning a bit to 1 upon encountering a $\ell_{\mathtt{H}}$ (signifying that $\ell_{\mathtt{H}}$ has been scheduled since $\ell_{\mathtt{L}}$ was last scheduled), and assigning the bit to 0 upon encountering a $\ell_{\mathtt{L}}$ *only* if the bit is 1 (if it is 0, the schedule is rejected).

With this predicate, we can formalize schedulers which never allow runs to "outrun" their parent runs, for any lattice. We call these *high-lead schedulers*.

Definition 4.5. *σ is a high-lead scheduler* ($\sigma \in \textsc{hls}$) iff
$\forall \bar{\ell} \centerdot \sigma \xrightarrow{\bar{\ell}} \implies \forall \ell_{\mathtt{L}}, \ell_{\mathtt{H}} \centerdot \ell_{\mathtt{L}} \sqsubset \ell_{\mathtt{H}} \implies \phi(\ell_{\mathtt{H}}, \ell_{\mathtt{L}}, 0, \bar{\ell})$.

An example of a high-lead scheduler is the round-robin scheduler which schedules levels in (increasing) order of maximal descendancy from top (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, when $s \in \textsc{tsni}$, the interaction on ℓ-presence channels in and $\mathsf{SME}(\sigma, s)$ is the same, for all ℓ.

Theorem 4.6. $\forall \sigma \in \textsc{hls}, s \in \textsc{tsni}, e, \bar{a} \centerdot$

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

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

5. Declassification

Many systems intentionally leak information as part of their function [47]. For instance, systems with a login interface leak the (mis)match of a username-password pair through the success/failure of a login attempt, systems that release the average salary of workers for statistics purposes leak information about the salary of each worker, and displaying the popularity of an item in an online shop leaks information about the purchases of customers. The target properties of interest for such systems are ones which stipulate that all leaks (if any) are *intentional*. An intended leak is referred to as a *release*, and we refer to the act of releasing information as a *declassification* (note: while these words are typically synonymous in literature, we use them as prescribed here). Such properties, i.e. models of declassification, have proven to be highly scenario-specific; as of yet, there is no *one* model universally applicable. Exploring such models is an active area of research, the state of the art being a classification of models by declassification goals (called *dimensions* of declassification), and a set of *principles* which a model should follow [47].

As is, SME enforces noninterference, which disallows all leaks, intentional or otherwise. When applied on a system with intentional leaks, SME removes the leaks, thus removing *a feature* from the system. This makes SME impractical for this broad class of systems. To remedy this, we present a variant of SME, SME$_{\mathrm{D}}$, which supports declassification. A declassification model which would be a good fit for our scenario is one which requires that *1)* declassification only takes place *where* the system needs it, and *2) what* leaks are allowed must be defined outside the system (so they can be blackbox enforced by SME$_{\mathrm{D}}$). These goals are primarily along the *where* (in the sense of where in the code the release may take place and where in the security lattice the affected information might be) and *what* (in the sense of what information may be released) dimensions of declassification [47], although the *who* dimension plays a role since attacker-controlled code and input must not affect which leaks are allowed. While models along the *when* dimension (which e.g. strive to delay release, release infrequently, or release only after a session has completed) provide useful guarantees, we find that facilitating them requires nontrivial extensions to our semantic and information-flow model for goals which are ultimately domain-specific.

While looking for a model matching the above goals, we discovered that none of the models we studied and their accompanying enforcements proved to be an ideal fit for our scenario. Whereas SME is a blackbox dynamic enforcement operating on interactive systems, most models of declassification are designed for substantially different system models, and most enforcements are whitebox and static [47].

Furthermore, in many models of declassification, intended leaks are *defined* through annotations (e.g. `declassify`) in the program (and thereby assumed to be *desired* by principals); such an approach would not work in our scenario, since the systems SME operates on are possibly-malicious black boxes.

To show that SME_D enforces our desired goals, we prove it sound w.r.t. two models of declassification. One is *gradual release* [2], a *where* model which we have adapted to our scenario. Gradual release, along with an accompanying static enforcement, has previously been given for a semantic model for noninteractive imperative programs [2], and a generalization of gradual release with *what* goals with an accompanying hybrid enforcement, has previously been given for interactive programs [3]. In both cases, enforcements are whitebox, intended leaks are defined by the program, and the models were timing-insensitive. One reason for the above-mentioned generalization is that gradual release allows leaking arbitrary contextual information through a declassification action. Addressing the *what* dimension of declassification, we introduce a new model of release, *full release*, which is a blackbox *what* model that disallows leaking arbitrary contextual information through a declassification. Essentially, SME_D satisfies full release by virtue of how SME *separates* input at different security levels into independent runs. To this end, while being primarily a *what* condition, full release has aspects of *where in the security lattice* [47] since it limits the effects of declassification to the declared security levels.

This very separation makes declassification in SME nontrivial, since, by design, e.g. the L run does not have H information as part of its state when H information is to be declassified to L; L effects are controlled (only) by the L run, and H input is fed (only) to the H run. To let L effects depend on declassified H information in SME_D, we need to move (only) the declassified information from the H run to the L run in a controlled way, without breaking the kind of guarantee SME was originally designed to provide. It turns out that the communication model from Section 4 is an excellent fit for secure communication between the runs at different levels. Recall that we wish to prevent the *occurrence* of declassification events from leaking information about the context while allowing intended release of the *value* to be declassified [47]. This is exactly the problem we needed to solve to properly handle channels with different security levels for *presence* and *content*! Conveniently, the same approach works here. The core idea is depicted in Figure 6. Declassification corresponds to routing an M output (containing the declassified value) from the H run into the L run. As with M output in SME, the occurrence of the declassification in the H run in SME_D is not leaked.

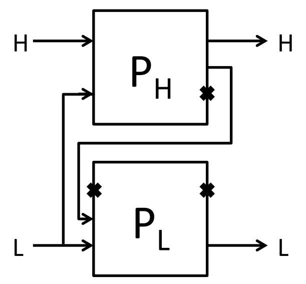

Fig. 6.: SME with declassification

We begin this section with a brief departure from SME to formalize our declassification goals in Sections 5.1 and 5.2. We then return to SME in Sections 5.3-5.5 where we give a sound and transparent variant of SME, SME_D, that enforces these declassification goals, and discuss a variant SME'_D in Section 5.6.

5.1. Gradual release

One of our declassification goals is that information release only takes place *where* the system needs it. However, the systems we consider are black boxes which send and receive messages; at the point the system performs a declassification, the system is performing a noninteraction. When the effect of the declassification is then observed (much) later in the system-environment interaction, it is difficult to assess whether the observed leak was caused by the declassification or not. To address this issue, gradual release assumes that systems make declassifications an observable *effect*, thus providing the released information to its observer right where the declassification occurs. These effects form the *interface* the system uses to *declare release intent to the gradual release property*. The system then only leaks where needed if the system only leaks through these release actions, which is what gradual release stipulates.

We model s which make declassifications (i.e. the declarations of release intent) observable effects in our semantic model as follows. We use our channel abstraction to define release actions. Let $\mathbb{C}_{\mathtt{R}} \subseteq \mathbb{C}$ be the set of *release channels*, ranged by $c_{\mathtt{R}}$, and let $\mathbb{A}_{\mathtt{R}} = \{c_{\mathtt{R}}?v, c_{\mathtt{R}}!v \mid c_{\mathtt{R}} \in \mathbb{C}_{\mathtt{R}} \wedge v \in \mathbb{V}\}$ be the set of release-actions, ranged by $a_{\mathtt{R}}$. A release action leaks information *from* one security level, *to* another. Let $\varphi(c_{\mathtt{R}})$ denote the from-level, and $\pi(c_{\mathtt{R}}) = \kappa(c_{\mathtt{R}})$ denote the to-level of $c_{\mathtt{R}}$. The set of actions that release information to an ℓ-observer is then given as follows.

$$\mathbb{A}^{\ell}_{\mathtt{R}} = \{a_{\mathtt{R}} \in \mathbb{A}_{\mathtt{R}} \mid \pi(a_{\mathtt{R}}) \sqsubseteq \ell \wedge \varphi(a_{\mathtt{R}}) \not\sqsubseteq \ell\}.$$

The following inference rule illustrates how the semantics of declassification, in a simple imperative language with I/O, can be given such that declassification becomes an effect.

$$\frac{(m, x := t) \xrightarrow{\bullet} (m', \mathtt{skip}) \qquad m'(x) = v}{(m, x := \mathtt{declassify}(t, c_{\mathtt{R}})) \xrightarrow{c_{\mathtt{R}}!v} (m', \mathtt{skip})}$$

The action label $c_{\mathtt{R}}!v$ on the transition signals, to gradual release, that the release is intended. This declassify construct is more fine-grained than the standard one as it specifies both the *from* level $\varphi(c_{\mathtt{R}})$ and the *to* level $\pi(c_{\mathtt{R}})$ of the declassification operation. Furthermore, the ability to support multiple channels releasing information from, say, `H` to `L`, enables us to encode the *semantics* of the value being released in the name of the channel; one channel could release age category, another could release gender, etc.

We say that s is *releasing* if it can perform a release action, and that s is *release-compliant* if all its release actions are output actions (s does not ask for permission to release; it just declares it; hence the release action must be an output).

Definition 5.1. For all s_0,

1. s_0 *is release-compliant* iff $\forall s \in \mathsf{S}(s_0) \,.\, \forall \bar{a}, i \,.\, s \xrightarrow{\bar{a}.i} \implies i \notin \mathbb{A}_{\mathtt{R}}$.
2. s_0 *is releasing* ($s \in \mathtt{LTS}^{\mathtt{R}}_{\mathtt{IO}}$) iff $\exists \bar{a}, a_{\mathtt{R}} \,.\, s \xrightarrow{\bar{a}.a_{\mathtt{R}}}$.

Unless stated otherwise, any s we consider in this paper is release-compliant.

Gradual release stipulates that an ℓ-observer only learns about ℓ-unobservables at points where the system performs a declassification. To formalize this, we first formalize what it means for the attacker to gain knowledge (cf. [2,3,9,20]). Recall the idea behind noninterference from Section 3.6. Assume the attacker knows the semantics of a system s and knows (or *chooses*) the ℓ-observables in the input environment e. From the attacker's point of view, the input environment could be any one of $\{e' \mid e\, \mathcal{R}_\ell\, e'\}$. The goal of the attacker is to rule out elements of this set as possible input environments, by observing the ℓ-observable part of the interaction of s with e. If the attacker cannot, then s satisfies *noninterference*. If the attacker can rule out an environment $e'\, \mathcal{R}_\ell\, e$ as a possible input environment, then the attacker obtains the *knowledge* that the ℓ-unobservables in e are *not* as defined by e' (thus learning things he is not privileged to learn). We define the knowledge of an ℓ-observer after observing a sequence of actions

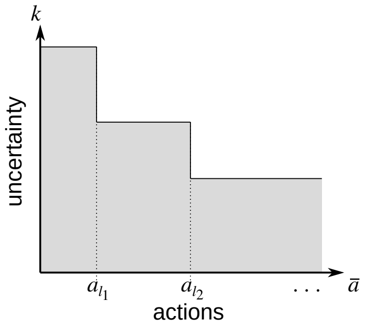

Fig. 7.: Leaks through actions a_{l_1} and a_{l_2} reduce uncertainty about secrets

$\bar{a}$ performed by a system s under environment e as the set $k_\ell^{\mathcal{R}}(s, e, \bar{a})$ of input environments under which those observables were possible:

$$k_\ell^{\mathcal{R}}(s, e, \bar{a}) = \{e' \mid e \,\mathcal{R}_\ell\, e' \wedge \exists \bar{a}' \,.\, e' \models s \xrightarrow{\bar{a}'} \wedge\, \bar{a} \,\mathcal{R}_\ell\, \bar{a}'\}.$$

The smaller this set is, the less uncertainty there is about the input environment, and thus, the greater the knowledge of the ℓ-observer. This is depicted in Figure 7. There, actions a_{l_1} and a_{l_2}, when observed by the ℓ-observer, reduce the uncertainty about the secrets in the input environment, and thus *leak* information to ℓ.

As more actions are observed, knowledge monotonely increases.

Lemma 5.2. $\forall \mathcal{R} \in \{\simeq, \approx\}, \ell, s, e, \bar{a}, a \,.\, k_\ell^{\mathcal{R}}(s, e, \bar{a}) \supseteq k_\ell^{\mathcal{R}}(s, e, \bar{a}.a)$

As hinted above, if an ℓ-observer never obtains knowledge about ℓ-unobservable input by observing ℓ-observable parts of the system-environment interaction, then the system satisfies noninterference.

Definition 5.3. *s is $\mathcal{R}$-noninterfering$_k$* ($s \in \mathrm{NI}_k^{\mathcal{R}}$) iff
$\forall \ell, e, \bar{a}, a \,.\, e \models s \xrightarrow{\bar{a}.a} \implies k_\ell^{\mathcal{R}}(s, e, \bar{a}) = k_\ell^{\mathcal{R}}(s, e, \bar{a}.a)$.

As the name suggests, this knowledge-based formalization of noninterference is equivalent to the two-run formalization given in Section 3.6.

Theorem 5.4. $\forall \mathcal{R} \in \{\simeq, \approx\}, s \,.\, s \in \mathrm{NI}_k^{\mathcal{R}} \iff s \in \mathrm{NI}^{\mathcal{R}}$.

Gradual release states that an ℓ-observer can only gain knowledge by observing release actions. In terms of Figure 7, gradual release requires that $a_{l_1}, a_{l_2} \in \mathbb{A}_{\mathrm{R}}^{\ell}$.

Definition 5.5. *s satisfies $\mathcal{R}$-gradual release* ($s \in \mathrm{GR}^{\mathcal{R}}$) iff
$\forall \ell, e, \bar{a}, a \,.\, e \models s \xrightarrow{\bar{a}.a} \implies a \notin \mathbb{A}_{\mathrm{R}}^{\ell} \implies k_\ell^{\mathcal{R}}(s, e, \bar{a}) = k_\ell^{\mathcal{R}}(s, e, \bar{a}.a)$.

By comparing Definitions 5.5 and 5.3, it is apparent that $\mathrm{GR}^{\mathcal{R}}$ weakens $\mathrm{NI}^{\mathcal{R}}$ by strengthening the antecedent of the implication such that the consequent needs only hold for nonrelease actions. Thus, for the class of s which do not release information, gradual release and noninterference are equivalent. This is known as the *conservativeness* principle of declassification [47] that stipulates that for systems with no information release, the security condition is equivalent to noninterference.

Corollary 5.6. $\forall \mathcal{R} \in \{\simeq, \approx\}, s \notin \mathrm{LTS}_{\mathrm{IO}}^{\mathrm{R}} \,.\, s \in \mathrm{GR}^{\mathcal{R}} \iff s \in \mathrm{NI}^{\mathcal{R}}$.

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 withholds release actions, as per $\mathsf{W}(s)$, given below.

$$\frac{s \xrightarrow{a} s' \qquad a \in \mathbb{A}_{\mathrm{R}}}{\mathsf{W}(s) \xrightarrow{\bullet} \mathsf{W}(s')} \qquad \frac{s \xrightarrow{a} s' \qquad a \notin \mathbb{A}_{\mathrm{R}}}{\mathsf{W}(s) \xrightarrow{a} \mathsf{W}(s')}$$

We let $\mathsf{W}(\epsilon) = \epsilon$, $\mathsf{W}(a.\bar{a}) = \bullet.\mathsf{W}(\bar{a})$ if $a \in \mathbb{A}_{\mathrm{R}}$, and $\mathsf{W}(a.\bar{a}) = a.\mathsf{W}(\bar{a})$ otherwise. When a system satisfies gradual release, withholding its release actions can only reduce the amount of information it releases.

Corollary 5.7. $\forall \mathcal{R} \in \{\simeq, \approx\}, s \,.\, s \in \mathrm{GR}^{\mathcal{R}} \implies \forall \ell, e, \bar{a} \,.\, k_\ell^{\mathcal{R}}(\mathsf{W}(s), e, \mathsf{W}(\bar{a})) \supseteq k_\ell^{\mathcal{R}}(s, e, \bar{a})$.

5.2. Full release

Gradual release provides a natural way to formalize *where* goals in untrusted blackbox systems. However, on its own, gradual release does *not* satisfy our *what* declassification goal. To see why, first consider the following program p_{d1}, with from-level $\varphi(c_{\mathtt{R}}) = \mathtt{B}$ and presence-level $\pi(c_{\mathtt{R}}) = \mathtt{L}$.

```
in A^L a; in B^L b;                                              // p_d1
if a mod 2 { l := declassify(b, c_R) }
out L l
```

This program is not secure; it leaks *contextual* information a through the *presence* of release action $c_{\mathtt{R}}$!b. If a L-observer does not observe $c_{\mathtt{R}}$!b (after a certain number of computation steps, or upon observing the L output in the progress-sensitive timing-insensitive case), then the L-observer can infer that a is even. Fortunately, gradual release rejects p_{d1}. However, the following variant p_{d2} *does* satisfy gradual release.

```
in A^L a; in B^L b;                                              // p_d2
if a mod 2 { l := declassify(o(b), c_R) }
else       { l := declassify(e(b), c_R) }
out L l
```

Let o (resp. e) be a bijection between the integers and the odd (resp. even) integers. Then l is odd iff a mod 2 = 1. While the presence of the release action is invariant of L-unobservables, this program leaks the sensitive *context*, a, through the *value* of the release action. Being a pure *where* condition, gradual release abstracts away the fact that these are different declassification statements (occurring in different lexical contexts). This is easier to see if we modify the program again, to p_{d3}, where $\varphi(c_{\mathtt{RA}}) = \mathtt{A}$, $\varphi(c_{\mathtt{RB}}) = \mathtt{B}$ and $\pi(c_{\mathtt{RA}}) = \pi(c_{\mathtt{RB}}) = \mathtt{L}$, which is likewise not secure but accepted by gradual release.

```
in M h; in A^L a; in B^L b;                                      // p_d3
if h mod 2 { l := declassify(o(a), c_RA) }
else       { l := declassify(e(b), c_RB) }
out L l
```

Our declassification goals require further restrictions in addition to gradual release to limit *what* can be released during a release action.

To address the *what* dimension, we introduce a new model of declassification: *full release*. Unlike gradual release, full release has no particular interface a system must make use of. Full release treats a system as a black box, operates only on its inputs and outputs, and stipulates that all leaks in a system are in accordance with a predefined *release policy*, which is external to the system. This is a good match for our *what* declassification goal, which states that permitted leaks must be (able to be) provided independently of the system, since systems are not trusted.

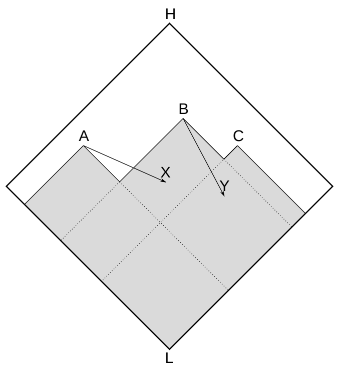

Fig. 8.: Permitted flows to C given $\rho = \{(\mathtt{A},\mathtt{X}),(\mathtt{B},\mathtt{Y})\}$

A *release policy* ρ is a set of pairs (ℓ, ℓ') for which $\ell \not\sqsubseteq \ell'$. These pairs define exceptions to the standard only-upwards flows policy that noninterference stipulates. For instance, when $\rho = \{(\mathtt{A},\mathtt{L})\}$ in $\mathcal{L}_{\mathtt{AB}}$, ρ permits the standard only-upwards flows of information that noninterference permits, but furthermore *permits* information to *leak* from A to L. A release policy is therefore similar to the intransitive downgrading relation $\rightsquigarrow$ used in intransitive noninterference [30,31]. Let $(\sqsubseteq^\rho)$ be defined as the reflexive *transitive* closure of $(\sqsubseteq) \cup \rho$. This relation expresses which flow paths become permitted once the leaks in ρ are permitted. This is illustrated in Figure 8. When $\rho = \emptyset$, then the permitted flow paths become $(\sqsubseteq^\rho) = (\sqsubseteq)$, same as for noninterference. Thus (only) information with security levels in the C-L rectangular region of the lattice is permitted to flow to C. However, if information is permitted to leak from A to X and from B to Y, then $\rho = \{(\mathtt{A}, \mathtt{X}), (\mathtt{B}, \mathtt{Y})\}$. Since $\mathtt{X} \sqsubseteq \mathtt{B}$ and $\mathtt{Y} \sqsubseteq \mathtt{C}$, information with a label anywhere in the gray region of the lattice is permitted to flow to C. Assuming an ℓ-observer observes all information which can travel along permitted flow paths to ℓ, this leads to the following new notion of equivalence of environments.

Definition 5.8. *e_1 and e_2 are ρ-ℓ-$\mathcal{R}$-equivalent* ($e_1\, \mathcal{R}^\rho_\ell\, e_2$) iff $\forall \ell' \,.\, \ell' \sqsubseteq^\rho \ell \implies e_1\, \mathcal{R}_{\ell'}\, e_2$.

Observe that we already have $\forall \mathcal{R} \in \{\simeq, \approx\}, \ell, e_1, e_2 \,.\, e_1\, \mathcal{R}_\ell\, e_2 \implies \forall \ell' \,.\, \ell' \sqsubseteq \ell \implies e_1\, \mathcal{R}_{\ell'}\, e_2$. The minute, yet key, difference is the ρ on the $\sqsubseteq$, which requires $e_1\, \mathcal{R}_{\ell'}\, e_2$ furthermore holds for $\ell' \not\sqsubseteq \ell$ for which $\ell' \sqsubseteq^\rho \ell$.

Corollary 5.9. $\forall \mathcal{R} \in \{\simeq, \approx\}, \ell, e_1, e_2 \,.\, e_1\, \mathcal{R}^\emptyset_\ell\, e_2 \iff e_1\, \mathcal{R}_\ell\, e_2$.

The more leaks that are allowed, the stronger ρ-ℓ-$\mathcal{R}$-equivalence becomes.

Corollary 5.10. $\forall \mathcal{R} \in \{\simeq, \approx\}, \ell, \rho, \rho' \,.\, (\sqsubseteq^\rho) \subseteq (\sqsubseteq^{\rho'}) \implies \forall e_1, e_2 \,.\, e_1\, \mathcal{R}^{\rho'}_\ell\, e_2 \implies e_1\, \mathcal{R}^\rho_\ell\, e_2$.

Together, this gives that ρ-ℓ-$\mathcal{R}$-equivalence is most relaxed when no leaks are allowed.

Corollary 5.11. $\forall \mathcal{R} \in \{\simeq, \approx\}, \ell, e_1, e_2, \rho \,.\, e_1\, \mathcal{R}^\rho_\ell\, e_2 \implies e_1\, \mathcal{R}_\ell\, e_2$.

Full release states that inputs which are not permitted to flow to an ℓ-observer must not interfere with ℓ-observables.

Definition 5.12. *s satisfies $\mathcal{R}$-ρ-full release* ($s \in \mathrm{FR}^{\mathcal{R}}_\rho$) iff
$\forall \ell, e_1, e_2 \,.\, e_1\, \mathcal{R}^\rho_\ell\, e_2 \implies \forall \bar{a}_1 \,.\, e_1 \models s \xrightarrow{\bar{a}_1} \implies \exists \bar{a}_2 \,.\, e_2 \models s \xrightarrow{\bar{a}_2} \wedge \bar{a}_1\, \mathcal{R}_\ell\, \bar{a}_2$.

By comparing Definitions 5.12 and 3.3, it is apparent that $\mathrm{FR}^{\mathcal{R}}_\rho$, for any ρ, weakens $\mathrm{NI}^{\mathcal{R}}$ by strengthening the antecedent of the leftmost implication such that its consequent needs to hold for fewer environment pairs. To make this *conservativeness* of declassification [47] result clearer, we need to make two observations. First, when $\rho = \emptyset$, $(\sqsubseteq^\rho) = (\sqsubseteq)$, so $(\mathcal{R}^\rho_\ell) = (\mathcal{R}_\ell)$, and thus, $\emptyset$-full release is just noninterference.

Corollary 5.13. $\forall \mathcal{R} \in \{\simeq, \approx\}, s \,.\, s \in \mathrm{FR}^{\mathcal{R}}_\emptyset \iff s \in \mathrm{NI}^{\mathcal{R}}$.

Second, the more leaks are allowed, the weaker full release becomes. Full release is thus monotonely weakening in the size of $(\sqsubseteq^\rho)$.

Corollary 5.14. $\forall \mathcal{R} \in \{\simeq, \approx\}, \ell, \rho, \rho' \,.\, (\sqsubseteq^\rho) \subseteq (\sqsubseteq^{\rho'}) \implies \forall s \,.\, s \in \mathrm{FR}^{\mathcal{R}}_\rho \implies s \in \mathrm{FR}^{\mathcal{R}}_{\rho'}$.

Together this shows that full release follows the conservativeness principle of declassification, for any ρ.

Corollary 5.15. $\forall \mathcal{R} \in \{\simeq, \approx\}, s \,.\, s \in \mathrm{NI}^{\mathcal{R}} \implies \forall \rho \,.\, s \in \mathrm{FR}^{\mathcal{R}}_\rho$.

We close this section with a few examples that contrast full release against gradual release, highlight the main shortcoming of full release (that it is *coarse-grained*) and show the way these declassification goals complement each other.

Full release stipulates *what* information is allowed to leak. Thus, when the release policy does not allow a leak from ℓ to ℓ', then a system satisfying full release can by *no* means leak from ℓ to ℓ'. For instance, full release rejects p_{d1} and p_{d2} when $\rho = \{(\mathtt{B}, \mathtt{L})\}$ (correctly identifying the implicit flow of A-information, present in the context of the declassification, to B), and rejects p_{d3} when $\rho = \{(\mathtt{A}, \mathtt{L}), (\mathtt{B}, \mathtt{L})\}$. However, full release does not place demands or restrictions on how leaks are made, i.e. whether or not they only occur *where* the system declares intent by performing a declassification action. For instance, the following two programs p_{d4} and p_{d5} are treated exactly the same way by full release, both accepted if $\mathtt{H} \sqsubseteq^\rho \mathtt{L}$, and both rejected otherwise (with $\varphi(c_\mathtt{R}) = \mathtt{H}$ and $\pi(c_\mathtt{R}) = \mathtt{L}$).

```
in M h; out L h                                                    // p_d4
```

```
in M h; l := declassify(h, c_R); out L l                           // p_d5
```

Since gradual release rejects the former program (which leaks without declaring intent), and accepts the latter (which declares intent), the two models of declassification complement each other to a large extent. However, full release is restrictive when it comes to systems that declare intent for *some* leaks from ℓ to ℓ', but not for other such leaks. For instance, consider the following variant p_{d6} of programs p_{d1} to p_{d3}.

```
in M h_1; in M h_2;                                                // p_d6
if h_1 mod 2 { l := declassify(o(h_2), c_R) }
else         { l := declassify(e(h_2), c_R) }
out L l
```

Program p_{d6} satisfies gradual release, despite the leak. However, there is no ρ which makes full release accept p_{d5} and reject p_{d6}; full release will always accept both of them if $\mathtt{H} \sqsubseteq^\rho \mathtt{L}$, and reject both otherwise. Full release is therefore *coarse-grained*, allowing leaks in an all-or-nothing manner. It is, however, necessarily so; since systems are black boxes, full release cannot tell whether or not a system that leaks intentionally from H to L is unintentionally (or maliciously) encoding *all* H input in that one leak.

5.3. Semantics

We now return to SME to formalize a variant SME with declassification support, SME_D, which, in addition to enforcing TSNI, enforces the declassification goals developed in the previous sections. The main challenge in developing SME_D is the fact that ℓ-runs are black boxes: Consider a system which must leak a function of some H inputs as part of its function. The only interface SME has to the system are its effects (i.e. action labels); since systems are black boxes, SME as-is cannot tell the system apart from another system which leaks a different function of H inputs as part of its function, or one that does not need to leak at all, and therefore cannot provide to the L-run only the (parts of) H input the system needs to perform its function. If SME provides some H input to the L run, a leak may occur. If SME provides no H input to the L run, system functionality may break.

Our approach is to add to SME an *interface* which a wrapped system can utilize to ask for permission to declassify information. The semantics of SME_D are obtained by adding rules to the semantics of SME for managing this interface. The SME_D declassification interface assumes that the wrapped system exposes a piece of its plumbing – the declassifications – as effects which describe the declassifications and allow the context (in our case, SME_D) to manipulate them. Any attempt by an ℓ-run to leak information that SME_D has not explicitly allowed will fail, due to the way SME_D, like SME, separates input at different security levels into independent runs.

Analogous to our approach to define the release interface between a system and gradual release in Section 5.1, we turn to our channel abstraction to define the effects of the declassification interface. The similarity of the two formalizations is no accident; we will later build an interface to gradual release on top of our declassification interface. Let $\mathbb{C}_{\mathtt{D}} \subseteq \mathbb{C}$ be the set of *declassification channels*, ranged by $c_{\mathtt{D}}$, and let $\mathbb{A}_{\mathtt{D}} = \{c_{\mathtt{D}}?v, c_{\mathtt{D}}!v \mid c_{\mathtt{D}} \in \mathbb{C}_{\mathtt{D}} \wedge v \in \mathbb{V}\}$ be the set of *declassification actions*, ranged by $a_{\mathtt{D}}$. A declassification leaks information *from* one security level, *to* another. Let $\varphi(c_{\mathtt{D}})$ denote the from-level, and $\pi(c_{\mathtt{D}}) = \kappa(c_{\mathtt{D}})$ denote the to-level of $c_{\mathtt{D}}$ (φ is defined on both release- and declassification channels) (we will return to the $\pi(c_{\mathtt{D}}) = \kappa(c_{\mathtt{D}})$ assumption in Section 5.5). A particular $c_{\mathtt{D}}$ can represent something as general as "declassify some $\mathtt{H}$ information to $\mathtt{L}$", in which case $\varphi(c_{\mathtt{D}}) = \mathtt{H}$ and $\pi(c_{\mathtt{D}}) = \mathtt{L}$, or something as specific as "declassify the first character in Alice's password to Bob", in which case e.g. $\varphi(c_{\mathtt{D}}) = \mathtt{A}$ (Alice) and $\pi(c_{\mathtt{D}}) = \mathtt{B}$ (Bob).

The semantics of $\mathtt{SME}_{\mathtt{D}}$ requires that systems expose their declassification plumbing to the declassification interface through declassification actions as follows. To declassify a value v from ℓ to ℓ', the system first performs output $c_{\mathtt{D}}!v$, where $c_{\mathtt{D}}$, for which $\varphi(c_{\mathtt{D}}) = \ell$ and $\pi(c_{\mathtt{D}}) = \ell'$, represents the information being declassified. The systems should then block on input from $c_{\mathtt{D}}$. The following inference rule illustrates how the semantics of declassification, in a simple imperative language with I/O, can be given in a manner satisfying this requirement. Again, this construct is more fine-grained than the standard one, for the same reasons as the declassification construct given in Section 5.1 is.

$$\frac{(m, \mathtt{out}\ c_{\mathtt{D}}\ t) \xrightarrow{c_{\mathtt{D}}!v} (m, \mathtt{skip})}{(m, x := \mathtt{declassify}(t, c_{\mathtt{D}})) \xrightarrow{c_{\mathtt{D}}!v} (m, x := \mathtt{in}\ c_{\mathtt{D}})}$$

To restore the typical semantics of declassification (which does not make declassification an effect) in a system s satisfying the above requirement, we simply place s in a wrapper which makes communication on declassification channels a feedback, as per $\mathtt{F}(s)$, given below.

$$\frac{s \xrightarrow{c_{\mathtt{D}}!v} s'}{\mathtt{F}(s) \xrightarrow{\bullet} \mathtt{F}(c_{\mathtt{D}}?v, s')} \qquad \frac{s \xrightarrow{c_{\mathtt{D}}?v} s'}{\mathtt{F}(c_{\mathtt{D}}?v, s) \xrightarrow{\bullet} \mathtt{F}(s')} \qquad \frac{s \xrightarrow{a} s' \quad a \notin \mathbb{A}_{\mathtt{D}}}{\mathtt{F}(s) \xrightarrow{a} \mathtt{F}(s')}$$

However, the point of this requirement is that the immediate context of s, in our case $\mathtt{SME}_{\mathtt{D}}$, can control which value actually gets declassified by feeding any value v' back into s in place of v. To illustrate how $\mathtt{SME}_{\mathtt{D}}$ uses this feature to only release values which pass through this declassification interface and that $\mathtt{SME}_{\mathtt{D}}$ allows, let $\varphi(c_{\mathtt{D}}) = \mathtt{A}$, $\pi(c_{\mathtt{D}}) = \mathtt{B}$, and assume the $\mathtt{B}$-run has announced a $c_{\mathtt{D}}$-declassification by performing $c_{\mathtt{D}}!v_{\mathtt{B}}$. If the $\mathtt{A}$-run has already performed a corresponding $c_{\mathtt{D}}!v_{\mathtt{A}}$, and $\mathtt{SME}_{\mathtt{D}}$ allows this declassification, $\mathtt{SME}_{\mathtt{D}}$ has the $\mathtt{B}$-run perform $c_{\mathtt{D}}?v_{\mathtt{A}}$ as its next action. Otherwise, $\mathtt{SME}_{\mathtt{D}}$ has the $\mathtt{B}$-run perform $c_{\mathtt{D}}?\mathtt{d}$ as its next action. Thereby, the system only leaks at the point of declassification (since declassification is nonblocking), and by separation, only leaks information available to the $\mathtt{A}$-run to the $\mathtt{B}$-run (as opposed to leaking, say, arbitrary $\mathtt{H}$ information from the lexical context of the declassification statement). We will discuss the rationale for $\mathtt{SME}_{\mathtt{D}}$ making declassification actions nonblocking in Section 5.6.

Same inference rules as in the SME ℓ-stepper semantics, with $c \notin \mathbb{C}_{\mathtt{D}}$ added as a premise to every rule where c occurs, and then with the following two inference rules added:

$$\frac{s \xrightarrow{c_{\mathtt{D}}!v} s'}{(\bar{a}, \ell) \models (\bar{a}_\ell, s) \xrightarrow{\bullet} (\bar{a}_\ell.c_{\mathtt{D}}!v, s')}\,\texttt{D-o} \qquad \frac{s \xrightarrow{c_{\mathtt{D}}?v} s'}{(\bar{a}, \ell) \models (\bar{a}_\ell, s) \xrightarrow{c_{\mathtt{D}}?v} (\bar{a}_\ell.c_{\mathtt{R}}?v, s')}\,\texttt{D-i}$$

(a) SME$_{\mathtt{D}}$ ℓ-stepper.

Same inference rules as in the SME ℓ-chooser semantics, with $c \notin \mathbb{C}_{\mathtt{D}}$ added as a premise to every rule where c occurs, and then with the following three inference rules added:

$$\frac{(\bar{a}, \ell) \models S(\ell) \xrightarrow{c_{\mathtt{D}}?v} (\bar{a}_\ell.c_{\mathtt{D}}!v.c_{\mathtt{D}}?v, s) \qquad c_{\mathtt{D}} \notin \delta_\ell}{\bar{a} \models (\ell.\sigma, S) \xrightarrow{\bullet} (\sigma, S[\ell \mapsto (\bar{a}_\ell.c_{\mathtt{D}}!v.c_{\mathtt{D}}?v, s)])}\,\texttt{D-no}$$

$$\frac{(\bar{a}, \ell) \models S(\ell) \xrightarrow{c_{\mathtt{D}}?\mathtt{d}} (\bar{a}_\ell.c_{\mathtt{D}}?\mathtt{d}, s) \qquad c_{\mathtt{D}} \in \delta_\ell \qquad S(\varphi(c_{\mathtt{D}})) = (\bar{a}_\varphi, \_) \qquad |\bar{a}_\varphi\restriction_{!,c_{\mathtt{D}},\bullet}| < |\bar{a}_\ell\restriction_{!,c_{\mathtt{D}},\bullet}|}{\bar{a} \models (\ell.\sigma, S) \xrightarrow{\bullet} (\sigma, S[\ell \mapsto (\bar{a}_\ell.c_{\mathtt{D}}?\mathtt{d}, s)])}\,\texttt{D-yes}_{\mathtt{d}}$$

$$\frac{\begin{array}{c}(\bar{a}, \ell) \models S(\ell) \xrightarrow{c_{\mathtt{D}}?v} (\bar{a}_\ell.c_{\mathtt{D}}!v_\ell.c_{\mathtt{D}}?v, s) \qquad c_{\mathtt{D}} \in \delta_\ell \qquad S(\varphi(c_{\mathtt{D}})) = (\bar{a}_\varphi.c_{\mathtt{D}}!v_\varphi.\_, \_) \qquad |\bar{a}_\varphi\restriction_{!,c_{\mathtt{D}},\bullet}| = |\bar{a}_\ell\restriction_{!,c_{\mathtt{D}},\bullet}| \\ \textbf{if } (\exists \ell_{\mathtt{d}}, \bar{a}_{\mathtt{d}} \centerdot c_{\mathtt{D}} \in \delta_{\ell_{\mathtt{d}}} \wedge S(\ell_{\mathtt{d}}) = (\bar{a}_{\mathtt{d}}.c_{\mathtt{D}}!\_.c_{\mathtt{D}}?\mathtt{d}.\_, \_) \wedge |\bar{a}_\varphi\restriction_{!,c_{\mathtt{D}},\bullet}| = |\bar{a}_{\mathtt{d}}\restriction_{!,c_{\mathtt{D}},\bullet}|) \textbf{ then } v = \mathtt{d} \textbf{ else } v = v_\varphi\end{array}}{\bar{a} \models (\ell.\sigma, S) \xrightarrow{\bullet} (\sigma, S[\ell \mapsto (\bar{a}_\ell.c_{\mathtt{D}}!v_\ell.c_{\mathtt{D}}?v, s)])}\,\texttt{D-yes}$$

(b) SME$_{\mathtt{D}}$ ℓ-chooser

Same inference rules as in the SME history semantics.

(c) SME$_{\mathtt{D}}$ history

Fig. 9. Semantics of SME$_{\mathtt{D}}$

This approach, and F, only have the desired effect if s adheres to our declassification interface, by first performing an output on a $c_{\mathtt{D}}$, and subsequently performing input on $c_{\mathtt{D}}$. We define this class of s now.

Definition 5.16. For all s_0,

1. s_0 *is declassification-compliant* iff $\forall \bar{a}_0, s \centerdot s_0 \xrightarrow{\bar{a}_0} s \implies$
   (a) $\forall \bar{a}, c_{\mathtt{D}}, v \centerdot s \xrightarrow{\bar{a}.c_{\mathtt{D}}!v} \implies \exists v' \centerdot s \xrightarrow{\bar{a}.c_{\mathtt{D}}!v.c_{\mathtt{D}}?v'}$, and
   (b) $\forall \bar{a}, c_{\mathtt{D}}, v \centerdot s \xrightarrow{\bar{a}.c_{\mathtt{D}}?v} \implies \exists v', \bar{a}' \centerdot \bar{a} = \bar{a}'.c_{\mathtt{D}}!v'$.
2. s_0 *is declassifying* ($s_0 \in \mathrm{LTS}^{\mathtt{D}}_{\mathrm{IO}}$) iff $\exists \bar{a}, a_{\mathtt{D}} \centerdot s_0 \xrightarrow{\bar{a}.a_{\mathtt{D}}}$.

Unless stated otherwise, any s we consider in this paper is declassification-compliant. We parameterise SME$_{\mathtt{D}}$ with a set $\delta \subseteq \mathbb{C}_{\mathtt{D}}$ of *desired declassifications*, which SME$_{\mathtt{D}}$ consults upon encountering a declassification announcement; if $c_{\mathtt{D}} \notin \delta$, the declassification is turned into a feedback. Let $\delta_\ell = \{c_{\mathtt{D}} \in \delta \mid \pi(c_{\mathtt{D}}) \sqsubseteq \ell \wedge \varphi(c_{\mathtt{D}}) \not\sqsubseteq \ell\}$ be the set of desired declassification channels which an ℓ-observer can observe the target, but not the source, of.

The semantics of SME$_{\mathtt{D}}$ is given in Figure 9. A SME$_{\mathtt{D}}$ state is the same as a SME state; the only new component, δ, is stateless, and used to instantiate the semantics of SME$_{\mathtt{D}}$. Note that $\mathsf{SME}_{\mathtt{D}}(\sigma, s, \delta)$ interacts *only on communication channels*, that is, $\mathsf{SME}_{\mathtt{D}}(\sigma, s, \delta) \notin \mathrm{LTS}^{\mathtt{D}}_{\mathrm{IO}}$; all declassification channel interaction is handled internally. By (D-no), any undesired declassification is a feedback. Furthermore, when an ℓ-run requests a declassification on $c_{\mathtt{D}}$ which is desirable, but which ℓ is not a valid target of, the request is a feedback. By (D-yes$_{\mathtt{d}}$), when a valid target ℓ-run of a declassification $c_{\mathtt{D}}$ needs the declassified value before the source $\varphi(c_{\mathtt{D}})$-run produces it, $\mathtt{d}$ is declassified instead. Otherwise, by (D-yes), if a valid target of a declassification has already received $\mathtt{d}$ for the declassification in question, so does the ℓ-run (either every valid target of a declassification that requests it gets the value produced by $\varphi(c_{\mathtt{D}})$ (if one was produced), or none of them do). This is to prevent the presence or timing of a $c_{\mathtt{D}}$-declassification from leaking through the declassification action, analogous to our treatment of $\mathtt{M}$-output in SME. Otherwise, $\varphi(c_{\mathtt{D}})$ has already produced the value to be declassified into the ℓ-run, and no other valid target has requested the value before it got produced, so a declassification is made by transferring the declassified value from the $\varphi(c_{\mathtt{D}})$-run to the ℓ-run.

5.4. Soundness

We now show how SME$_{\mathtt{D}}$ fits into our declassification models. Recall that SME$_{\mathtt{D}}$ takes as parameter the set δ of desired declassifications defining *what* can be leaked, and that SME$_{\mathtt{D}}$ utilizes rule (D-yes$_{\mathtt{d}}$) or (D-yes) in Figure 9b *where* a declassification occurs. To intentionally leak information, a system utilizes the declassification interface of SME$_{\mathtt{D}}$. This provides SME$_{\mathtt{D}}$ with the information it needs to circumvent ℓ-run separation and support intentional release as a system feature, while at the same time only releasing information between security levels as desired by δ, which is our desired *what* goal. Furthermore, SME$_{\mathtt{D}}$ only declassifies information using rules (D-yes$_{\mathtt{d}}$) or (D-yes). This provides *us* with the information we need to prove that SME$_{\mathtt{D}}$ only releases information through declassification actions, which is our desired *where* goal. We do this by making actions derived with (D-yes$_{\mathtt{d}}$) and (D-yes) *release actions*, to interface SME$_{\mathtt{D}}$ with gradual release.

We start with the *what* results. Let $\rho(\delta) = \{(\varphi(c_{\mathtt{D}}), \pi(c_{\mathtt{D}})) \mid c_{\mathtt{D}} \in \delta\}$ be the *reconciled releases* of δ. As per the discussion on the coarseness of full release at the end of Section 5.2, since systems are (possibly malicious) black boxes, and since there therefore is no way to know or control what $\varphi(c_{\mathtt{D}})$-information a system leaks through $c_{\mathtt{D}}$, any leakage of $\varphi(c_{\mathtt{D}})$-information to $\pi(c_{\mathtt{D}})$ the system makes must be reconciled. Fortunately, $\rho(\delta)$ are the only leaks SME$_{\mathtt{D}}$ can make. Let $\mathrm{FR}_{\rho} = \mathrm{FR}^{\simeq}_{\rho}$.

Theorem 5.17. $\forall \delta, \sigma, s \,.\, \mathsf{SME}_{\mathtt{D}}(\sigma, s, \delta) \in \mathrm{FR}_{\rho(\delta)}$.

The proof of this theorem follows a similar pattern as the proof of Theorem 4.2, utilizing a lemma which is nearly identical to Lemma 4.3.

If a system does not utilize the declassification interface, running the system in SME$_{\mathtt{D}}$ causes no information to be released; the resulting execution is noninterfering. This is a corollary of Theorem 4.2, since no step in a trace from $\mathsf{SME}_{\mathtt{D}}(\sigma, s, \rho)$ is derived using any of the new rules from Figure 9.

Corollary 5.18. $\forall \delta, \sigma, s \notin \mathrm{LTS}^{\mathtt{D}}_{\mathrm{IO}} \,.\, \mathsf{SME}_{\mathtt{D}}(\sigma, s, \delta) \in \mathrm{TSNI}$.

To provide our *where* results, we need to interface SME$_{\mathtt{D}}$ with gradual release. Before we do that, we show how to interface the systems SME$_{\mathtt{D}}$ operates on (i.e. declassification-compliant ones) with gradual release. We do this my creating a mapping from declassification-channels to release channels, as follows: Let $\varrho : \mathbb{C}_{\mathtt{D}} \to \mathbb{C}_{\mathtt{R}}$ be an injective function associating a release channel with each declassification channel, in such a way that the from- and to-levels match, that is, $\varphi(\varrho(c_{\mathtt{D}})) = \varphi(c_{\mathtt{D}})$ and $\pi(\varrho(c_{\mathtt{D}})) = \pi(c_{\mathtt{D}})$ (assume $\mathbb{C}_{\mathtt{D}}$ and $\mathbb{C}_{\mathtt{R}}$ are disjoint). Then, for the declassification semantics in Section 5.3, wrapping a declassifying system in the following declassification feedback $\mathsf{F}_{\mathtt{R}}$ will likewise perform a release action iff the system performs a declassification.

$$\frac{s \xrightarrow{c_{\mathtt{D}}!v} s'}{\mathsf{F}_{\mathtt{R}}(s) \xrightarrow{\bullet} \mathsf{F}_{\mathtt{R}}(c_{\mathtt{D}}?v, s')} \qquad \frac{\varrho(c_{\mathtt{D}}) = c_{\mathtt{R}} \quad s \xrightarrow{c_{\mathtt{D}}?v} s'}{\mathsf{F}_{\mathtt{R}}(c_{\mathtt{D}}?v, s) \xrightarrow{c_{\mathtt{R}}!v} \mathsf{F}_{\mathtt{R}}(s')} \qquad \frac{s \xrightarrow{a} s' \quad a \notin \mathbb{A}_{\mathtt{D}}}{\mathsf{F}_{\mathtt{R}}(s) \xrightarrow{a} \mathsf{F}_{\mathtt{R}}(s')}$$

In SME_{D}, there are, for each declassification, (potentially) multiple ℓ-runs receiving it. Since not all receiving runs are necessarily ready to receive it at the time the value being declassified is produced by the from-level-run, the value cannot be fed to all of them at once. Indeed, the presence of a particular declassification action in those receiving ℓ-runs can vary, and each such (non)reception can leak different information to different observers. Furthermore, performing a release action when the from-level creates the value to be declassified leaks the presence of the declassification action in the from-level. Therefore, we make the reception of a declassified value, by each valid target of said declassification, a separate release action. We overload $\varrho : \mathbb{C}_{\mathtt{D}} \to \mathcal{L} \nrightarrow \mathbb{C}_{\mathtt{R}}$ to a partial injective function ($\nrightarrow$ denotes partial function) such that $\varrho(c_{\mathtt{D}}, \ell)$ is defined iff $c_{\mathtt{D}} \in \delta_\ell$, and such that $\varphi(\varrho(c_{\mathtt{D}}, \ell)) = \varphi(c_{\mathtt{D}})$ and $\pi(\varrho(c_{\mathtt{D}}, \ell)) = \ell$. Rules (D-yes$_{\text{d}}$) and (D-yes) in Figure 9b are responsible for providing a declassified value to a valid target. By modifying the conclusion of these rules from the form

$$\bar{a} \models (\ell.\sigma, S) \xrightarrow{\bullet} (\sigma, S[\ell \mapsto (\bar{a}_\ell.c_{\mathtt{D}}?v, s)]),$$

to the form

$$\bar{a} \models (\ell.\sigma, S) \xrightarrow{\varrho(c_{\mathtt{D}},\ell)!v} (\sigma, S[\ell \mapsto (\bar{a}_\ell.c_{\mathtt{D}}?v, s)]),$$

we obtain a variant SME_{R} of SME_{D} which performs a release action iff a declassified value is provided to a valid target run.

We are now ready to present the *where* result. The following technical lemma is in the style of Lemma 4.3. It states that $\mathsf{SME}_{\mathtt{R}}(\sigma, s, \delta)$ can, under all the environments which an ℓ-observer knows could have caused $\bar{a}$, not only match $\bar{a}$, but do so in such a way that all ℓ'-runs, for $\ell' \sqsubseteq \ell$, are in exactly the same state. Let $k_\ell = k_\ell^{\simeq}$.

Lemma 5.19. $\forall \delta, \sigma, s.$
$\forall e_1, \bar{a}_1, \sigma_1, S_1 \,\mathbf{.}\, e_1 \models \mathsf{SME}_{\mathtt{R}}(\sigma, s, \delta) \to (\bar{a}_1, \sigma_1, S_1) \implies$
$\forall \ell, e_2 \,\mathbf{.}\, e_2 \in k_\ell(\mathsf{SME}_{\mathtt{R}}(\sigma, s, \delta), e_1, \bar{a}_1) \implies$
$\exists \quad \bar{a}_2, \sigma_2, S_2 \,\mathbf{.}\, e_2 \models \mathsf{SME}_{\mathtt{R}}(\sigma, s, \delta) \to (\bar{a}_2, \sigma_2, S_2) \wedge$
$\bar{a}_1 =_\ell \bar{a}_2 \wedge \sigma_1 = \sigma_2 \wedge S_1 =_\ell S_2.$

We now have that SME_{R} enforces gradual release, for any desired declassifications. Let $\text{GR} = \text{GR}^{\simeq}$.

Theorem 5.20. $\forall \delta, \sigma, s \,\mathbf{.}\, \mathsf{SME}_{\mathtt{R}}(\sigma, s, \delta) \in \text{GR}.$

Thus, SME_{D} only releases information through declassifications. Our instrumentation of SME_{D} with release actions was done to provide the *where* guarantee we needed. In practice, however, release actions will not be present in the SME_{D} of any system. To show that our instrumentation is nothing more than a tool to reason about *where* information leaks occur, we show that SME_{D} leaks at most as much as SME_{R}. First, for any list of actions SME_{R} can perform, SME_{D} can perform that same sequence of actions with release actions withheld. This follows from the definition of SME_{R}.

Corollary 5.21. $\forall \delta, \sigma, s, e, \bar{a} \,\mathbf{.}\, e \models \mathsf{SME}_{\mathtt{D}}(\sigma, s, \delta) \xrightarrow{\bar{a}} \implies \exists! \bar{a}' \,\mathbf{.}\, e \models \mathsf{SME}_{\mathtt{R}}(\sigma, s, \delta) \xrightarrow{\bar{a}'} \wedge \mathsf{W}(\bar{a}') = \bar{a}.$

We refer to the $\bar{a}'$ in the above corollary as $\mathsf{R}(\bar{a})$. The following result is what we are after. It follows from the above and Corollary 5.7.

Corollary 5.22. $\forall \delta, \sigma, s, e, \bar{a} \,\mathbf{.}\, e \models \mathsf{SME}_{\mathtt{D}}(\sigma, s, \delta) \xrightarrow{\bar{a}} \implies$
$k_\ell(\mathsf{SME}_{\mathtt{D}}(\sigma, s, \delta), e, \bar{a}) \subseteq k_\ell(\mathsf{SME}_{\mathtt{R}}(\sigma, s, \delta), e, \mathsf{R}(\bar{a})).$

In summary, our full release soundness result states that the only leaks $\mathrm{SME}_\mathtt{D}$ can have are those declared as desired during instantiation of $\mathrm{SME}_\mathtt{D}$, and our gradual release soundness result states that $\mathrm{SME}_\mathtt{D}$ only leaks through desired declassification actions. This, plus the way $\mathrm{SME}_\mathtt{D}$ otherwise separates information like SME, ensures that ℓ-runs only leak information that has been declassified to them, and thus cannot leak arbitrary contextual information. For instance, in program p_{d1}, the $\mathrm{SME}_\mathtt{D}$ of p_{d1} allows the declassification but prevents the implicit flow of a at the same time! This is a fruitful byproduct of our *what* model of declassification and the separation of computation into ℓ-runs; the B-run never obtains A-information, and thus cannot leak it (not even implicitly). At last, $\mathrm{SME}_\mathtt{D}$ of the following program, with $\delta = \{c_\mathtt{D}\}$, $\pi(c_\mathtt{D}) = \mathtt{L}$ and $\varphi(c_\mathtt{D}) = \mathtt{H}$, allows the announced declassification, but stops the explicit flow. This follows from our *where* guarantee and SME separation; the L-run receives dummy values as input on M, and only receives the right $\mathtt{h}_1$ value where the declassification occurs.

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

5.5. *Transparency*

The new declassification features we added to SME introduce new circumstances under which transparency is broken. If a system attempts a declassification which $\mathrm{SME}_\mathtt{D}$ does not reconcile, then $\mathrm{SME}_\mathtt{D}$ will deny the declassification by feeding d to the to-run, thus breaking transparency. However, $\mathrm{SME}_\mathtt{D}$ can impede transparency even if all releases made by a system are reconciled. By Corollary 5.18, when s attempts to leak information without utilizing the declassification interface, the corresponding control in $\mathrm{SME}_\mathtt{D}(\sigma, s, \delta)$ never receives the declassified value. For instance, with $\delta = \{c_\mathtt{D} \mid \pi(c_\mathtt{D}) = \mathtt{L} \wedge \varphi(c_\mathtt{D}) = \mathtt{H}\}$, consider the system s defined by program p_{d4} in Section 5.2. We have $s \in \mathrm{FR}_{\rho(\delta)}$ (i.e. if we reconcile all releases made by s, s is otherwise secure). However, the L-run in $\mathrm{SME}_\mathtt{D}(\sigma, s, \delta)$ never gets the H-value in the M-input, and thus, $\mathrm{SME}_\mathtt{D}(\sigma, s, \delta)$ cannot be transparent. Thus $s \in \mathrm{FR}_{\rho(\delta)}$ is too weak an assumption to guarantee that $\mathrm{SME}_\mathtt{D}(\sigma, s, \delta)$ is transparent. What is needed is a property stipulating that a system only releases information through its declassification interface. By design, this property is TSNI! Due to the assumption in Section 5.3 that $\pi(c_\mathtt{D}) = \kappa(c_\mathtt{D})$ for all $c_\mathtt{D}$, TSNI assumes that an ℓ-observer who can observe the to-level of a declassification also observes the values being declassified. So for declassifying systems, TSNI says that if we fix declassified values arbitrarily, no information is released. This implies that TSNI systems do not release non-declassified values, i.e., for TSNI systems, release is only possible through the declassification interface. However, even if we assume TSNI, a similar problem arises when the L-run reaches a declassification before the H-run does; then the L-run instead receives d. This occurs in the $\mathrm{SME}_\mathtt{D}$ of p_{d5} from Section 5.2 ($c_\mathtt{R}$ replaced with $c_\mathtt{D}$) if L is scheduled too often before H. Fortunately, we can rule out this scenario by assuming high-lead schedulers.

It turns out that these three scenarios – *1)* $\mathrm{SME}_\mathtt{D}$ preventing undesired declassifications, *2)* systems not utilizing the declassification interface to leak, and *3)* the to-level of a release reaching the declassification action before the from-level does – are the only inhibitors for transparency. So by assuming high-lead scheding, that $s \in \mathrm{TSNI}$, and that all declassifications attempted by s are desired by $\mathrm{SME}_\mathtt{D}$ (that is, $\delta = \{c_\mathtt{D} \mid \exists \bar{a}, v \,.\, s \xrightarrow{\bar{a}.c_\mathtt{D}!v}\}$), we get transparency – that is, that the interaction on ℓ-presence channels, of s under $\mathrm{SME}_\mathtt{D}$ cf. s with declassification channels in feedback, is ℓ-equivalent.

Theorem 5.23. $\forall \sigma \in \text{HLS}, s \in \text{TSNI}, e, \bar{a}\,\textbf{.}\,$, *with* $\delta(s) = \{c_{\mathtt{D}} \mid \exists \bar{a}, v\,\textbf{.}\, s \xrightarrow{\bar{a}.c_{\mathtt{D}}!v}\}$,

a) $e \models \mathsf{F}(s) \xrightarrow{\bar{a}} \implies \exists \bar{a}'\,\textbf{.}\, e \models \mathsf{SME}_{\mathtt{D}}(\delta(s), \sigma, s) \xrightarrow{\bar{a}'} \wedge \forall \ell\,\textbf{.}\, \bar{a} \leq_{\star,\pi^{-1}(\ell),\bullet} \bar{a}'$
b) $e \models \mathsf{SME}_{\mathtt{D}}(\delta(s), \sigma, s) \xrightarrow{\bar{a}} \implies \exists \bar{a}'\,\textbf{.}\, e \models \mathsf{F}(s) \xrightarrow{\bar{a}'} \wedge \forall \ell\,\textbf{.}\, \bar{a} \leq_{\star,\pi^{-1}(\ell),\bullet} \bar{a}'$

Program p_{d5} satisfies TSNI; when $c_{\mathtt{D}}$ input (which has L presence and content) is fixed by the environment, changes in H input do not affect the value declassified. Also, the presence of the declassification action does not depend on L-unobservables. It is easy to see that if $\sigma \in$ HLS and $\delta = \{c_{\mathtt{D}}\}$, then $\text{SME}_{\mathtt{D}}$ of p_{d5} routes the value announced by the H-run (which is the actual value received on M) to the L-run in the desired way, thus yielding a transparent run. When $\delta = \emptyset$, $\text{SME}_{\mathtt{D}}$ of p_{d5} stops the forbidden declassification to preserve soundness. Since $\delta \neq \delta(s)$, our transparency result does not guarantee transparency. Indeed, transparency does not hold, since in $\text{SME}_{\mathtt{D}}$, the declassification is prevented.

To further clarify the scope of the above transparency result, it is worth noting that there are programs which, when declassification is a noninteraction, satisfy TSNI, but when their declassification plumbing is pulled out to adhere to the $\text{SME}_{\mathtt{D}}$ declassification interface, do *not* satisfy TSNI; the following modification p_{d7} of p_{d5} is such a program.

```
in M h; l := declassify(h, cD); if l != h { out L 0 }                    // pd7
```

If $c_{\mathtt{D}}$ is made into feedback, the result is a program which is not in $\text{LTS}^{\mathtt{D}}_{\text{IO}}$ and satisfies TSNI (and thus has no leaks). However, without said feedback, since the program compares the value it received on $c_{\mathtt{D}}$ with the value it sent on $c_{\mathtt{D}}$, the program fails to satisfy TSNI, so $\text{SME}_{\mathtt{D}}$ does not *have* to be transparent for it. Indeed, as it turns out, the $\text{SME}_{\mathtt{D}}$ of this program attempts to leak whether the *actual* M-input received by the $\text{SME}_{\mathtt{D}}$ of p_{d7} equals d or not (which δ then either permits or prevents), so $\text{SME}_{\mathtt{D}}$ of p_{d7}, while sound for any δ, is not transparent. Thus, TSNI for programs in $\text{LTS}^{\mathtt{D}}_{\text{IO}}$ requires that a program can, without causing further leaks, let an *arbitrary* value (provided by the environment, independent of program state) be the result of executing a declassify statement. This rules out programs that behave like p_{d7}. This is needed by $\text{SME}_{\mathtt{D}}$ since $\text{SME}_{\mathtt{D}}$ ensures that the L-run never gets the H input it needs to construct the correct value to be declassified, to ensure that the only H values the L-run gets are ones that have been declassified. Without this assumption on s, $\text{SME}_{\mathtt{D}}$ can only guarantee soundness, not transparency. This explains why we did not assume $\mathsf{F}(s) \in$ TSNI in our transparency result (which is otherwise more natural). On a related note, this approach seems to be one way of resolving the issues with the *scrambling* declassification semantics of *(qualified) robustness* [32] mentioned by Sabelfeld and Sands [47].

5.6. *Blocking on declassification*

We close this section with a justification for the part of the semantics of $\text{SME}_{\mathtt{D}}$ which makes declassification actions performed by ℓ-runs nonblocking operations. Consider a variant $\text{SME}'_{\mathtt{D}}$ of $\text{SME}_{\mathtt{D}}$ which does block the to-level-run of a declassification until the from-level-run has produced the value to be declassified. To keep the justification concrete, we consider only the $\mathcal{L}_{\text{LH}}$ below, and note that the justifications generalize to arbitrary lattices.

It turns out that $\text{SME}'_{\mathtt{D}}$ is both transparent and sound. The transparency proof is the same as for $\text{SME}_{\mathtt{D}}$ (since, as noted at the end of as noted in Section 3.5, no L actions are permitted to follow a H input in secure programs). The soundness proof applies if we have $\text{SME}'_{\mathtt{D}}$ produce a release action each time the L-run is scheduled while blocking on a declassified value. The reason why this change to the soundness proof is needed – and the reason we chose to have $\text{SME}_{\mathtt{D}}$ make declassifications nonblocking – is that the *nonpresence* of the declassification output from the H-run can leak 1 bit about the state of the H-run, *each time* the L-run is scheduled, until the H-run produces a value to be declassified. This bit could be all the information contained in the H-run state, and this could be a different bit every time the L-run is scheduled, as is the case in the below program, for scheduler $(\texttt{H.L})^{\infty}$ (which we assume is known to the attacker).

```
do {                                                    // p_d8
  in H h;
} while ( h != d) ;
l := declassify(h, H->L);
out L l;
```

In constrast, SME_{D} leaks at most the whole state of the H-run once for each declassify action the L-run attempts (in the above example, one bit). We find that with the declassification semantics in SME'_{D}, it is harder to assess what and how much is leaked (e.g. when the program running under SME'_{D} is white-box), as the leak for a single declassify action can continue over time, forever. Hence we settled for SME_{D}.

6. Full transparency

This section shows how to achieve full 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 10. 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 will 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 to noninterference.

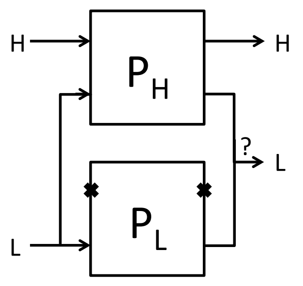

Fig. 10.: SME with barrier synchronization

6.1. Semantics

A counterexample to $s \in \text{NI}^{\mathcal{R}}$ is a proof of the logic negation of $s \in \text{NI}^{\mathcal{R}}$. That is, a level of observation ℓ, two $\mathcal{R}_\ell$-equivalent environments, and a trace which s can perform under one of those environments, but cannot $\mathcal{R}_\ell$-match under the other. We refer to such a counterexample as an *$\mathcal{R}$-attack on s.*

Definition 6.1. An *$\mathcal{R}$-attack* α is a 4-tuple $(\ell, e_1, e_2, \bar{a}_1)$ with $e_1 \; \mathcal{R}_\ell \; e_2$ and $e_1 \models \bar{a}_1$. The attack is an *$\mathcal{R}$-attack on s* iff

1) $e_1 \models s \xrightarrow{\bar{a}_1}$, and
2) $\forall \bar{a}_2 \,.\, e_2 \models s \xrightarrow{\bar{a}_2} \implies \neg(\bar{a}_2 \; \mathcal{R}_\ell \; \bar{a}_1)$

We now formalize our variant of SME, SME_{T}, which can construct attacks on run-time. Its semantics for a two-point lattice is given in Figure 11. SME_{T} adds two new components to the SME-state, $\bar{\alpha}_1$ and $\bar{\alpha}_2$, which contain the list of (potential) attacks discovered so far, making the SME_{T}-state a quintuple $(\bar{a}, \sigma, S, \bar{\alpha}_1, \bar{\alpha}_2)$. The semantics works as follows. While nothing inhibits the H-run from performing

Obtained by adding the SME ℓ-stepper rules (dead) and (silent), to the following four inference rules:

$$\frac{s \xrightarrow{a} s' \qquad \bar{a}_\ell.a \leq_{\star,\ell,\bullet} \bar{a} \qquad \textbf{if } a = c?v \textbf{ then } (\textbf{if } \kappa(c) \not\sqsubseteq \ell \textbf{ then } v = \mathtt{d} \textbf{ else } v \neq \star\,)}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{\bullet} (\bar{a}_\ell.a, s')}\text{old}$$

$$\frac{\bar{a}'.a' \leq \bar{a} \qquad \pi(a') = \mathtt{L} \qquad \bar{a}' \approx_{\mathtt{L}} \bar{a}_{\mathtt{H}} \qquad \forall a\,.\, s \xrightarrow{a} \Longrightarrow a \neq_{\mathtt{L}} a'}{(\bar{a},\mathtt{H}) \models (\bar{a}_{\mathtt{H}}, s) \xrightarrow{\bullet} (\bar{a}_{\mathtt{H}}.\bullet, s)}\text{conflict}$$

$$\frac{s \xrightarrow{c!v_\ell} s' \qquad \bar{a}_\ell =_{\star,\ell,\bullet} \bar{a} \qquad \pi(c) \sqsubseteq \ell \qquad \textbf{if } \kappa(c) \sqsubseteq \ell \textbf{ then } v = v_\ell \textbf{ else } v = \mathtt{d}}{(\bar{a},\ell) \models (\bar{a}_\ell, s) \xrightarrow{c!v} (\bar{a}_\ell.c!v_\ell, s')}\text{new-o}$$

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

(a) SME$_{\mathrm{T}}$ ℓ-stepper

Obtained by adding the ($\bullet$) SME ℓ-stepper inference rule (with $\bar{\alpha}_1$, $\bar{\alpha}_2$ unaffected by the transition) to the following five inference rules:

$$\frac{(\bar{a},\mathtt{H}) \models S(\mathtt{H}) \xrightarrow{a} (\bar{a}_{\mathtt{H}}, s_{\mathtt{H}}) \qquad \pi(a) = \mathtt{H}}{\bar{a} \models (\mathtt{H}.\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2) \xrightarrow{a} (\sigma, S[\ell \mapsto (\bar{a}_{\mathtt{H}}, s_{\mathtt{H}})], \bar{\alpha}_1, \bar{\alpha}_2)}\text{H-a}$$

$$\frac{(\bar{a},\mathtt{H}) \models S(\mathtt{H}) \xrightarrow{a} \qquad \pi(a) = \mathtt{L}}{\bar{a} \models (\mathtt{H}.\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2) \xrightarrow{a} (\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2)}\text{H-block}$$

$$\frac{(\bar{a},\mathtt{L}) \models S(\mathtt{L}) \xrightarrow{a_{\mathtt{L}}} (\bar{a}_{\mathtt{L}}, \_) \qquad \pi(a_{\mathtt{L}}) = \mathtt{L} \qquad S(\mathtt{H}) = (\bar{a}_{\mathtt{H}}, \_) \qquad |\bar{a}_{\mathtt{L}}| + t \geq |\bar{a}_{\mathtt{H}}| \qquad (\bar{a},\mathtt{H}) \models S(\mathtt{H}) \xrightarrow{a_{\mathtt{H}}} \qquad \pi(a_{\mathtt{H}}) \neq \mathtt{L}}{\bar{a} \models (\mathtt{L}.\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2) \xrightarrow{\bullet} (\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2)}\text{L-wait}$$

$$\frac{\begin{array}{c}(\bar{a},\mathtt{L}) \models S(\mathtt{L}) \xrightarrow{a_{\mathtt{L}}} (\bar{a}_{\mathtt{L}}, s_{\mathtt{L}}) \qquad \pi(a_{\mathtt{L}}) = \mathtt{L} \qquad S(\mathtt{H}) = (\bar{a}_{\mathtt{H}}, \_) \qquad |\bar{a}_{\mathtt{L}}| + t \geq |\bar{a}_{\mathtt{H}}| \qquad (\bar{a},\mathtt{H}) \models S(\mathtt{H}) \xrightarrow{a_{\mathtt{H}}} \qquad \pi(a_{\mathtt{H}}) = \mathtt{L} \\ \textbf{if } (\bar{a},\mathtt{H}) \models S(\mathtt{H}) \xrightarrow{a_{\mathtt{L}}} \vee a_{\mathtt{L}} =_{\mathtt{L}} a_{\mathtt{H}} \textbf{ then } \bar{\alpha}'_1 = \bar{\alpha}_1 \textbf{ else } \bar{\alpha}'_1 = \bar{\alpha}_1.(\mathtt{L}, \mathsf{E}(\bar{a}.a_{\mathtt{L}}), \mathsf{E}(\bar{a}_{\mathtt{L}}), \bar{a}_{\mathtt{L}}) \\ \textbf{if } (\bar{a},\mathtt{H}) \models S(\mathtt{H}) \not\xrightarrow{a_{\mathtt{L}}} \wedge a_{\mathtt{L}} =_{\mathtt{L}} a_{\mathtt{H}} \textbf{ then } a = a_{\mathtt{H}} \quad \textbf{else } a = a_{\mathtt{L}}\end{array}}{\bar{a} \models (\mathtt{L}.\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2) \xrightarrow{a} (\sigma, S[\mathtt{L} \mapsto (\bar{a}_{\mathtt{L}}, s_{\mathtt{L}})], \bar{\alpha}'_1, \bar{\alpha}_2)}\text{L-a}$$

$$\frac{(\bar{a},\mathtt{L}) \models S(\mathtt{L}) \xrightarrow{a_{\mathtt{L}}} (\bar{a}_{\mathtt{L}}, s_{\mathtt{L}}) \qquad \pi(a_{\mathtt{L}}) = \mathtt{L} \qquad S(\mathtt{H}) = (\bar{a}_{\mathtt{H}}, \_) \qquad |\bar{a}_{\mathtt{L}}| + t < |\bar{a}_{\mathtt{H}}| \qquad \bar{\alpha}'_2 = \bar{\alpha}_2.(\mathtt{L}, \mathsf{E}(\bar{a}.a_{\mathtt{L}}), \mathsf{E}(\bar{a}_{\mathtt{L}}), \bar{a}_{\mathtt{L}})}{\bar{a} \models (\mathtt{L}.\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2) \xrightarrow{a_{\mathtt{L}}} (\sigma, S[\mathtt{L} \mapsto (\bar{a}_{\mathtt{L}}, s_{\mathtt{L}})], \bar{\alpha}_1, \bar{\alpha}'_2)}\text{L-timeout}$$

(b) SME$_{\mathrm{T}}$ ℓ-chooser

$$\frac{\bar{a} \models (\sigma, S, \bar{\alpha}_1, \bar{\alpha}_2) \xrightarrow{a} (\sigma', S', \bar{\alpha}'_1, \bar{\alpha}'_2)}{(\bar{a}, \sigma, S, \bar{\alpha}_1, \bar{\alpha}_2) \xrightarrow{a} (\bar{a}.a, \sigma', S', \bar{\alpha}'_1, \bar{\alpha}'_2)}\text{history}$$

(c) SME$_{\mathrm{T}}$ history

Fig. 11. Semantics of SME$_{\mathrm{T}}$

non-L-presence actions (by (H-a) and ($\bullet$)), 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 corresponding action, a barrier forms (by (L-wait)). 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) advanced t steps without reaching a L-observable action (by (L-timeout)). In 1), if the H-run reached a L-observably different action, we note the *discrepancy* in $\bar{\alpha}_1$. The H-run will then be replaced by infinite silence (by (conflict)). In 2), we note the *timeout* in $\bar{\alpha}_2$. Input environments are constructed from traces using E, defined as $(\mathsf{E}(\bar{a}))_c = (\bar{a}\restriction_{?,c}).(c?\star)^\infty$.

Not every attack discovered by $\mathsf{SME}_{\mathrm{T}}(s)$ is an attack on s; we will establish when a discovered attack is an attack on s in Section 6.4.

6.2. Soundness

$\mathsf{SME_T}$ enforces PSNI. This is easily seen by observing that the traces produced by $e \models \mathsf{SME_T}(s)$ and $e\restriction_{\mathtt{L}} \models s$ are $\approx_{\mathtt{L}}$-equivalent. By allowing the L-run to wait up to t steps for the H-run to match an L-observable, $\mathsf{SME_T}(s)$ introduces a timing leak into s when $t > 0$, and thus does not enforce TSNI.

Theorem 6.2. $\forall s, \sigma\,.\, \mathsf{SME_T}(\sigma, s) \in$ PSNI.

We note, however, that $\mathsf{SME_T}$ *does* enforces TSNI when $t = 0$, and that the attacks discovered are $\simeq$-attacks. We find, however, that $\mathsf{SME_T}$ becomes less practical with $t = 0$, since $\mathsf{SME_T}$ is much more likely to detect timeouts or discrepancies; when $\mathsf{SME_T}$ detects these, $\mathsf{SME_T}$ “freezes” the H-run to avoid leaks, making the H-run semantically equivalent to a program producing • infinitely, by (conflict). A more practical version of $\mathsf{SME_T}$ would instead have the H-run behave like it would under $\mathsf{SME}(s)$ henceforth. We hypothesize (but do not prove) that this modification of $\mathsf{SME_T}$ yields a sound enforcement. In any case, this weakening of the soundness guarantee compared to SME is easily controlled by simply wrapping $\mathsf{SME_T}(\sigma, s)$ in a blackbox timing leak mitigator [4].

6.3. Transparency

Modulo •, $\mathsf{SME_T}(s)$ and s produce the same sequence of I/O (up to discrepancy or timeout in $\mathsf{SME_T}(s)$). This holds *even when s is not secure*. Point *a)* below states that $\mathsf{SME_T}$ cannot break transparency without producing an attack on s, and Point *b)* states that any trace that s can perform under $\mathsf{SME_T}$ which does not produce an attack, can be matched by s not running under $\mathsf{SME_T}$.

Theorem 6.3. $\forall \sigma, s, e, \bar{a}.$

$$\begin{array}{ll} a)\ e \models s \xrightarrow{\bar{a}} & \Longrightarrow \nexists \bar{a}'\,.\, e \models \mathsf{SME_T}(\sigma, s) \to (\bar{a}', \_, \_, \epsilon, \epsilon) \wedge \bar{a}' \not\leq_{\star,\bullet} \bar{a} \wedge \bar{a} \not\leq_{\star,\bullet} \bar{a}' \\ b)\ e \models \mathsf{SME_T}(\sigma, s) \to (\bar{a}, \_, \_, \epsilon, \epsilon) & \Longrightarrow \exists \bar{a}'\,.\, e \models s \xrightarrow{\bar{a}'} \wedge \bar{a} =_{\star,\bullet} \bar{a}' \end{array}$$

A corollary of the above is that if $s \in$ TSNI, then s and $\mathsf{SME_T}(\sigma, s)$ have the same (i.e. $=_{\star,\bullet}$-equivalent) I/O behavior! This is because $\mathsf{SME_T}(s)$ never generates attacks for $s \in$ TSNI. In contrast to e.g. Devriese and Piessens [19], who swap the order of outputs in the following two programs (for linearization $\mathtt{B} \sqsubseteq \mathtt{A}$), we have full I/O correspondence between each program and its $\mathsf{SME_T}$.

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

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

However, the same I/O correspondence cannot hold in general if $s \in$ PSNI, since it is impossible (without solving the Turing halting problem) to determine whether the H-run is taking forever, or just a very long time, to match the L action by the L-run. This can be demonstrated by the following PSNI program; for any predetermined value of t, there is an input on M for which $\mathsf{SME_T}$ of the program times out while waiting for the H-run to reach the L output statement.

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

6.4. Attacks

It is important to note that even when s is not secure, running s under SME_T does not guarantee attack discovery; the SME_T of s might take branches of control in such a way that insecure states are never entered, and SME_T only detects attacks along the control flow path that its ℓ-runs take. However, if SME_T *does* detect a discrepancy between the H-run and the L-run on which L-observable comes next (before a timeout has occurred), then SME_T has indeed found a concrete proof that $s \notin \text{PSNI}$.

Theorem 6.4. $\forall s, \sigma, e \,.\, e \models \mathtt{SME}_\mathtt{T}(\sigma, s) \to (\_, \_, \_, \alpha, \epsilon) \implies$ *α is an $\approx$-attack on s.*

This theorem states that if the first attack discovered is a discrepancy, then the discrepancy is an $\approx$-attack on s. Further discrepancies discovered before timeouts are discovered are $\approx$-attack on s as well; we focused on the first attack in the above theorem since attacks discovered later may simply be extensions of the first attack. Unfortunately, as discussed above, we cannot infer in general that a timeout is an attack on s. Likewise, if a timeout is discovered before a discrepancy, we cannot conclude that the discrepancy is an attack on s, since the discrepancy might be the 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 describe the attacks which $\mathtt{SME}_\mathtt{T}$ finds on them. Consider the following program.

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

With $\mathtt{d} = 0$, $t = 100$ and initial input 1, $\mathtt{SME}_\mathtt{T}(s)$ generates an $\approx$-attack $(\mathtt{L}, e_1, e_2, \mathtt{M?d.L!}0)$ on s in $\bar{\alpha}_1$, where $e_1(\mathtt{M}) = \mathtt{M?d.}(\mathtt{M?}\star)^\infty$, $e_2(\mathtt{M}) = \mathtt{M?}1.(\mathtt{M?}\star)^\infty$. Now consider the following program.

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

With $\mathtt{d} = 0$, $t = 100$ and initial input -1, $\mathtt{SME}_\mathtt{T}(s)$ generates an $\approx$-attack $(\mathtt{L}, e_1, e_2, \mathtt{M?d.}\bullet\mathtt{.L!}0)$ on s in $\bar{\alpha}_2$, where $e_1(\mathtt{M}) = \mathtt{M?d.}(\mathtt{M?}\star)^\infty$, $e_2(\mathtt{M}) = \mathtt{M?} - 1.(\mathtt{M?}\star)^\infty$. 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 merely took "too long" for the H-run to match L!0).

7. Related work

Referring the reader for general overviews on language-based information-flow security [44], on dynamic information-flow control [23], and on declassification [47], we focus on related work on multi-execution.

Li and Zdancewic [29] 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. Their focus is on enforcing relaxed noninterference statically, by a security type system.

Russo et al. [41] 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. [16] 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 [19] 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. [12] investigate multi-execution in a reactive setting. Bielova et al.'s model multi-executes Featherweight Firefox [14], a formalization of a web browser as a reactive system. The environments of Bielova et al. are not necessarily total, but the security guarantee is weaker (than Devriese and Piessens' [19]): termination-insensitive noninterference. The I/O model targets the browser setting, with handlers under cooperative scheduling. The full version [13] 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. [26] 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 [25] implement a monadic library for secure multi-execution in Haskell. Austin and Flanagan [7] 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 have been implemented in practical programming languages: for JavaScript by Austin and Flanagan, and for Jeeves [54] by Austin et al. [8]. 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* [56,33]. Robust declassification operates on both confidentiality and integrity labels, requiring the decision to declassify 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 [47]. Compared to this approach, our declassification has aspects of *what* is declassified and *where* in the lattice and in the code information release may take place. 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. [11] 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. [22] 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. [12] discussed above, De Groef et al. multi-execute the actual scripts in web pages rather than the entire browser. The main focus of their experiments is to confirm that the enforcement of simple policies does not modify the behavior of secure pages.

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 [37] 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.

Performance-wise, like Devriese and Piessens [19], the time complexity of our approach is linear in the size of the security lattice, since a process is created for each lattice element. That being said, when the size of the lattice is smaller than the number of CPU cores, the overhead of our approach is negligible, and can even yield an improvement since runs synchronize less with the environment. This is indicated by benchmarks performed by Devriese and Piessens, which are portable to our setting.

Unno et al. [51] 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. [55] 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.

Most recently, Vanhoef et al. [52] have investigated an approach to declassification in secure multi-execution. This is accomplished through *escape hatch* [45] expressions, provided as policies for release, separately from code. Only through these expressions it is possible to declassify information to lower and incomparable security levels. This allows expressing *what* is released in a fine-grained fashion, but does not allow controlling *where* in the program the release may take place. Our approach is more coarse-grained for specifying *what* is released (per-level granularity rather than per-expression), yet it has the benefit of ensuring that information release takes place only at designated declassification points, thus also controlling *where* information can be declassified.

8. 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 and where in the program the release can take place. We prove that declassification only affects the declared security levels and that our declassification mechanism restricts release to program points with declassification annotations. 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.

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 pre-emptive 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, by the Swedish research agencies SSF and VR, the US Naval Research award no. N000141310156 and NSF grant award no. CNS1320470.

References

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

[2] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 207–221, May 2007.

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

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

[5] 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.

[6] 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.

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

[8] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted Execution of Policy-Agnostic Programs. In *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '13, pages 15–26, New York, NY, USA, 2013. ACM.

[9] A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, May 2008.

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

[11] G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. 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.

[12] 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.

[13] 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.

[14] 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.

[15] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *ACM Conference on Computer and Communications Security*, pages 79–90, November 2009.

[16] 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.

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

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

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

[20] C. Dima, C. Enea, and R. Gramatovici. Nondeterministic nointerference and deducible information flow. Technical Report 2006-01, University of Paris 12, LACL, 2006.

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

[22] 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.

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

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

[25] 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.

[26] 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.

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

[28] 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.

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

[30] H. Mantel. Information flow control and applications—Bridging a gap. In *Proc. Formal Methods Europe*, volume 2021 of *LNCS*, pages 153–172. Springer-Verlag, March 2001.

[31] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, November 2004.

[32] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 172–186, June 2004.

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

[34] 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.

[35] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. 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.

[36] 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.

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

[38] 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.

[39] W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *Proc. IEEE Computer Security Foundations Symposium*, pages 3–18, 2013.

[40] Syrian Electronic Army uses Taboola ad to hack Reuters (again). `https://nakedsecurity.sophos.com/2014/06/23/syrian-electronic-army-uses-taboola-ad-to-hack -reuters-again/`.

[41] 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.

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

[43] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.
[44] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
[45] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.
[46] 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.
[47] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 17(5):517–548, January 2009.
[48] 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.
[49] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, July 2003.
[50] 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.
[51] 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.
[52] M. Vanhoef, W. De Groef, D. Devriese, F. Piessens, and T. Rezk. Stateful declassification policies for event-driven programs. In *CSF*, 2014.
[53] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
[54] J. Yang, K. Yessenov, and A. Solar-Lezama. A Language for Automatically Enforcing Privacy Policies. In *POPL*, pages 85–96, 2012.
[55] D. Zanarini, M. Jaskelioff, and A. Russo. Precise enforcement of confidentiality for reactive system. In *Proc. IEEE Computer Security Foundations Symposium*, 2013.
[56] 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 [22] 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. 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 [19] 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 /* Example label conditional function */
12 var cross_origin = function ([url]) { return (url.indexOf("same-origin") == -1); };
13
14 /* Examples */
15 SME.Label("nsIDOMHTMLDocument_GetCookie").as(SME.Labels.HIGH).default("eat=this");
16 SME.Label("nsIDOMHTMLImageElement_SetSrc").if(cross_origin).as(SME.Labels.LOW).else(
       SME.Labels.HIGH);
17 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 sources 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();
```

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

```
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 [42].

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 [19] and at the heart of FlowFox [22]. 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.

B. Projections

A projection applies as far left as possible (unless indicated otherwise with parentheses). For instance, $\bar{a}.\bar{a}' \restriction$ means $(\bar{a}.\bar{a}')\restriction$, not $\bar{a}.(\bar{a}' \restriction)$. For all projections, $\epsilon \restriction = \epsilon$. Let $\$$ range over $?$ and $!$. The projection functions used throughout the paper and appendix are then as given in Figure 12.

C. Proofs

Proof of Theorem 3.6. Since s is input-blocking, the number of $\star$ reads s performs while blocking on input does not affect future behavior of s. Let $\hat{e}$ be the result of removing all $\star$ inputs in e (save infinitely trailing $\star$, if any). Then $\forall \bar{a} \,.\, e \models s \xrightarrow{\bar{a}} \implies \exists \bar{a}' \,.\, \hat{e} \models s \xrightarrow{\bar{a}} \wedge \bar{a} \simeq_\ell \bar{a}'$, and $\forall \bar{a} \,.\, \hat{e} \models s \xrightarrow{\bar{a}} \implies \exists \bar{a}' \,.\, e \models s \xrightarrow{\bar{a}} \wedge \bar{a} \simeq_\ell \bar{a}'$. Furthermore, $e \approx_\ell \hat{e}$. We now prove the contrapositive of the stated implication. Assume $s \notin$ PSNI. Then for some ℓ, e_1, e_2 and $\bar{a}_1$, we have $e_1 \approx_\ell e_2$, $e_1 \models s \xrightarrow{\bar{a}_1}$, but no $\bar{a}_2$ for which $e_2 \models s \xrightarrow{\bar{a}_2}$ and $\bar{a}_1 \approx_\ell \bar{a}_2$. By the above, we get for some $\bar{a}_1' \approx_\ell \bar{a}_1$ and for $\hat{e}_1$ and $\hat{e}_2$ that $\hat{e}_1 \models s \xrightarrow{\bar{a}_1'}$, but no $\bar{a}_2$

$$
\begin{array}{rlrl}
o.\bar{a}\restriction_{?} = & \bar{a}\restriction_{?} & o.\bar{a}\restriction_{?} = & \bar{a}\restriction_{?} \\
i.\bar{a}\restriction_{?} = & i.(\bar{a}\restriction_{?}) & i.\bar{a}\restriction_{?} = & i.(\bar{a}\restriction_{?}) \\
i.\bar{a}\restriction_{!} = & \bar{a}\restriction_{!} & \bar{a}\restriction_{\vec{\bullet}} = & \epsilon, \text{ if } \nexists \bar{a}'.a.\bar{a}'' \leq \bar{a} \,.\, a \neq \bullet \\
o.\bar{a}\restriction_{!} = & o.(\bar{a}\restriction_{!}) & \bar{a}'.a.\bar{a}''\restriction_{\vec{\bullet}} = & \bar{a}'.a.(\bar{a}''\restriction_{\vec{\bullet}}), \text{ if } a \neq \bullet \\
\bullet.\bar{a}\restriction_{c} = & \bar{a}\restriction_{c} & \bullet.\bar{a}\restriction_{\bullet} = & \bar{a}\restriction_{\bullet} \\
c'\$v.\bar{a}\restriction_{c} = & \bar{a}\restriction_{c} \text{ , if } c' \neq c & c\$v.\bar{a}\restriction_{\bullet} = & c\$v.(\bar{a}\restriction_{\bullet}) \\
c\$v.\bar{a}\restriction_{c} = & c\$v.(\bar{a}\restriction_{c}) & c?\star.\bar{a}\restriction_{\hat{\star}} = & \bar{a}\restriction_{c} \\
\bullet.\bar{a}\restriction_{\ell} = & \bullet.(\bar{a}\restriction_{\ell}) & c?v.\bar{a}\restriction_{\hat{\star}} = & c?v.(\bar{a}\restriction_{\hat{\star}}), \text{ if } v \neq \star \\
c\$v.\bar{a}\restriction_{\ell} = & \bullet.(\bar{a}\restriction_{\ell}), \text{ if } \pi(c) \not\sqsubseteq \ell & o.\bar{a}\restriction_{\hat{\star}} = & o.(\bar{a}\restriction_{\hat{\star}}) \\
c\$\star.\bar{a}\restriction_{\ell} = & c\$\star.(\bar{a}\restriction_{\ell}), \text{ if } \pi(c) \sqsubseteq \ell & c?\star.\bar{a}\restriction_{\star} = & \bullet.(\bar{a}\restriction_{c}) \\
c\$v.\bar{a}\restriction_{\ell} = & c\$\mathtt{d}.(\bar{a}\restriction_{\ell}), \text{ if } \pi(c) \sqsubseteq \ell, v \neq \star \text{ and } \kappa(c) \not\sqsubseteq \ell & c?v.\bar{a}\restriction_{\star} = & c?v.(\bar{a}\restriction_{\star}), \text{ if } v \neq \star \\
c\$v.\bar{a}\restriction_{\ell} = & c\$v.(\bar{a}\restriction_{\ell}), \text{ if } \pi(c) \sqsubseteq \ell, v \neq \star \text{ and } \kappa(c) \sqsubseteq \ell & o.\bar{a}\restriction_{\star} = & o.(\bar{a}\restriction_{\star})
\end{array}
$$

Fig. 12. Projection functions

for which $\hat{e}_2 \models s \xrightarrow{\bar{a}_2}$ and $\bar{a}_1' \approx_\ell \bar{a}_2$. Since $e_1 \approx_\ell \hat{e}_1$, $e_2 \approx_\ell \hat{e}_2$ and $e_1 \approx_\ell e_2$, we get by transitivity that $\hat{e}_1 \approx_\ell \hat{e}_2$. By construction and by definition of $\approx$ and $\simeq$, $\hat{e}_1 \simeq_\ell \hat{e}_2$. ($\approx_\ell$-equivalent environments which are not $\simeq_\ell$-equivalent are only allowed to differ in the number of $\star$s between the observables they provide). Since $(\simeq_\ell) \subsetneq (\approx_\ell)$, the nonexistence of a $\bar{a}_2$ such that $\hat{e}_2 \models s \xrightarrow{\bar{a}_2}$ and $\bar{a}_1' \approx_\ell \bar{a}_2$ implies nonexistence of a $\bar{a}_2$ such that $\hat{e}_2 \models s \xrightarrow{\bar{a}_2}$ and $\bar{a}_1' \simeq_\ell \bar{a}_2$. Thus $s \notin \mathsf{TSNI}$. $\square$

Proof of Lemma 4.3. Let s, σ, ℓ, e_1 and e_2 such that $e_1 \simeq_\ell e_2$ be given. We prove that

$$
\begin{aligned}
&\forall \bar{a}_1, \sigma_1, S_1 \,.\, e_1 \models \mathsf{SME}(\sigma, s) \twoheadrightarrow (\bar{a}_1, \sigma_1, S_1) \implies \\
&\exists \bar{a}_2, \sigma_2, S_2 \,.\, e_2 \models \mathsf{SME}(\sigma, s) \twoheadrightarrow (\bar{a}_2, \sigma_2, S_2) \wedge \\
&\bar{a}_1 =_\ell \bar{a}_2 \;\wedge\; S_1 =_\ell S_2 \;\wedge \sigma_1 = \sigma_2
\end{aligned}
\tag{1}
$$

by induction in $n = |\bar{a}_1|$. Let σ_1 and S_1 be given such that $e_1 \models \mathsf{SME}(\sigma, s) \twoheadrightarrow (\bar{a}_1, \sigma_1, S_1)$.

$n = 0$**:** Then $\bar{a}_1 = \epsilon$. Set $\bar{a}_2 = \epsilon$. Then for $(\bar{a}_2, \sigma_2, S_2) = \mathsf{SME}(\sigma, s) = (\bar{a}_1, \sigma_1, S_1)$, we get that $\bar{a}_2 = \bar{a}_1 = \epsilon$, $S_2 = S_1 = \lambda\ell \to s$ and $\sigma_2 = \sigma_1 = \sigma$.
Since σ_1 and S_1 such that $e_1 \models \mathsf{SME}(\sigma, s) \twoheadrightarrow (\bar{a}_1, \sigma_1, S_1)$ were arbitrary, (1) holds for $n = 0$.

$n + 1$ **given** n**:** Assume (1) for all $\bar{a}$ for which $|\bar{a}| = n$; this is the induction hypothesis (IH). Let $\bar{a}_1$ be such that $|\bar{a}_1| = n + 1$. Then $\bar{a}_1 = \bar{a}_1'.a_1$ for some $\bar{a}_1'$ and a_1. Let $e_1 \models \mathsf{SME}(\sigma, s) \twoheadrightarrow (\bar{a}_1', \sigma_1', S_1') \twoheadrightarrow (\bar{a}_1, \sigma_1, S_1)$. By (IH), we have for some $\bar{a}_2'$, σ_2' and S_2' for which $\bar{a}_1' =_\ell \bar{a}_2'$, $S_1' =_\ell S_2'$ and $\sigma_1' = \sigma_2'$ that $e_2 \models \mathsf{SME}(\sigma, s) \twoheadrightarrow (\bar{a}_2', \sigma_2', S_2')$. Let a_2, σ_2 and S_2 be chosen such that such that $e_2 \models \mathsf{SME}(\sigma, s) \twoheadrightarrow (\bar{a}_2', \sigma_2', S_2') \twoheadrightarrow (\bar{a}_2, \sigma_2, S_2)$, where we set $\bar{a}_2 = \bar{a}_2'.a_2$. For some ℓ_1, $\sigma_1' \xrightarrow{\ell_1} \sigma_1$. Since $\sigma_2' = \sigma_1'$, $\sigma_2' \xrightarrow{\ell_1} \sigma_2$. So $\sigma_1 = \sigma_2$.
It remains to be shown that $\bar{a}_1 =_\ell \bar{a}_2$ and $S_1 =_\ell S_2$. Case on ℓ_1.

$\ell_1 \not\sqsubseteq \ell$**:** Case on the derivation of the last step in trace $e_1 \models \mathsf{SME}(\sigma, s) \twoheadrightarrow (\bar{a}_1, \sigma_1, S_1)$. By considering all possible derivations of a step in Figure 5 and by observing that $\sigma_1' \xrightarrow{\ell_1} \sigma_1$ and that

$\ell_1 \not\sqsubseteq \ell$, we see that $a_1 \restriction_\ell = \bullet$ and for each $\ell' \sqsubseteq \ell$, $S_1(\ell') = S_1'(\ell')$, and thus $S_1 =_\ell S_1'$. By analogous reasoning, $a_2 \restriction_\ell = \bullet$ and for each $\ell' \sqsubseteq \ell$, $S_2(\ell') = S_2'(\ell')$, and thus $S_2 =_\ell S_2'$. By transitivity of $=_\ell$ on objects ranged by S, $S_1 =_\ell S_2$. By definition of $=_\ell$ on traces, by definition of $\bar{a}_1$ and $\bar{a}_2$, and since $\bar{a}_1' =_\ell \bar{a}_2'$, $\bar{a}_1 =_\ell \bar{a}_2$.

$\ell_1 \sqsubseteq \ell$: Since $S_1' =_\ell S_2'$, we get that $S_1'(\ell_1) = S_2'(\ell_1)$. Case on the Figure 5b-rule used in the derivation of the last step in trace $e_1 \models \mathsf{SME}(\sigma, s) \to (\bar{a}_1, \sigma_1, S_1)$.

($\bullet$): Here, $(\bar{a}_1', \ell_1) \models S_1'(\ell_1) \xrightarrow{\bullet} (\bar{a}_{\ell_1}^1, s_{\ell_1}^1)$ and $S_1 = S_1'[\ell_1 \mapsto (\bar{a}_{\ell_1}^1, s_{\ell_1}^1)]$ for some $(\bar{a}_{\ell_1}^1, s_{\ell_1}^1)$. We show that $(\bar{a}_2', \ell_1) \models S_2'(\ell_1) \xrightarrow{\bullet} (\bar{a}_{\ell_1}^1, s_{\ell_1}^1)$ (†). Thus, by setting $S_2 = S_2'[\ell_1 \mapsto (\bar{a}_{\ell_1}^1, s_{\ell_1}^1)]$, we obtain a derivation of the last step in trace $e_2 \models \mathsf{SME}(\sigma, s) \to (\bar{a}_2, \sigma_2, S_2)$ using Figure 5b-rule ($\bullet$). Since $S_1' =_\ell S_2'$, we will get by definition of S_1, S_2 and of $=_\ell$ on objects ranged by S that $S_1 =_\ell S_2$. Also, since $\bar{a}_1' =_\ell \bar{a}_2'$ and $a_1 = a_2 = \bullet$, we will get by definition of $\bar{a}_1$, $\bar{a}_2$, and of $=_\ell$ on traces that $\bar{a}_1 =_\ell \bar{a}_2$.
Case on the Figure 5a-rule used in the derivation of $(\bar{a}_1', \ell_1) \models S_1'(\ell_1) \xrightarrow{\bullet} (\bar{a}_{\ell_1}^1, s_{\ell_1}^1)$.

(dead), (silent), **or** (o-old)**:** In all these cases, $(\bar{a}_{\ell_1}^1, s_{\ell_1}^1)$ is not a function of $\bar{a}_1'$. Thus, since $\sigma_1' \xrightarrow{\ell_1} \sigma_1$ and $S_1'(\ell_1) = S_2'(\ell_1)$, we have (†), established with the same Figure 5a-rule used to establish $(\bar{a}_1', \ell_1) \models S_1'(\ell_1) \xrightarrow{\bullet} (\bar{a}_{\ell_1}^1, s_{\ell_1}^1)$.

(i-old)**:** Let $S_1(\ell_1) = (\bar{a}_{\ell_1}, s_{\ell_1})$. Then for some c, v and s_{ℓ_1}', $s_{\ell_1} \xrightarrow{c?v} s_{\ell_1}'$ and $\bar{a}_{\ell_1}.c?v \le_{\star,\ell_1,\bullet,?,c} \bar{a}_1'$.
Case on $\pi(c)$ and $\kappa(c)$.

$\pi(c) \not\sqsubseteq \ell_1$**:** Then the conditional in the rule yields $v = \mathtt{d}$ since $\pi(c) \sqsubseteq \kappa(c)$. Also, $\bar{a}_x.c?v \le_{\star,\ell_1,\bullet,?,c} \bar{a}_y$ for any $\bar{a}_x$ and $\bar{a}_y$ since both sides of $\le_{\star,\ell_1,\bullet,?,c}$ project to ϵ. Thus (†) holds.

$\pi(c) \sqsubseteq \ell_1 \wedge \kappa(c) \not\sqsubseteq \ell_1$**:** Then the conditional in the rule yields $v = \mathtt{d}$. For $\bar{a}_{\ell_1}.c?v \le_{\star,\ell_1,\bullet,?,c} \bar{a}_2'$ to hold, $\bar{a}_2'$ must have at least as many c-input actions (excluding blanks) as $\bar{a}_{\ell_1}.c?v$. This follows from $\bar{a}_{\ell_1}.c?v \le_{\star,\ell_1,\bullet,?,c} \bar{a}_1'$, $\bar{a}_1' =_\ell \bar{a}_2'$ and $\ell_1 \sqsubseteq \ell$. Thus (†) holds.

$\kappa(c) \sqsubseteq \ell_1$**:** For $\bar{a}_{\ell_1}.c?v \le_{\star,\ell_1,\bullet,?,c} \bar{a}_2'$ to hold, the c-input actions (excluding blanks) in $\bar{a}_{\ell_1}.c?v$ must prefix the c-input actions (excluding blanks) in $\bar{a}_2'$. By $\bar{a}_1' =_\ell \bar{a}_2'$ and $\ell_1 \sqsubseteq \ell$, we get that $\bar{a}_1'$ and $\bar{a}_2'$ have exactly the same c-input events (excluding blanks). Thus, by $\bar{a}_{\ell_1}.c?v \le_{\star,\ell_1,\bullet,?,c} \bar{a}_1'$, $\bar{a}_{\ell_1}.c?v \le_{\star,\ell_1,\bullet,?,c} \bar{a}_2'$ holds. Thus (†) holds.

(i-block)**:** Here, $(\bar{a}_1', \ell_1) \models S_1'(\ell_1) \xrightarrow{c?\star} (\bar{a}_{\ell_1}, s_{\ell_1})$ for some c for which $\pi(c) \sqsubset \ell_1$, $S_1 = S_1'[\ell_1 \mapsto (\bar{a}_{\ell_1}, s_{\ell_1})]$, and $a_1 = \bullet$. The only Figure 5a-rule which can constitute this derivation is (i-new). By this rule, $\bar{a}_{\ell_1}' =_{\star,\ell_1,\bullet,?,c} \bar{a}_1'$ and $s_{\ell_1}' \xrightarrow{c?\star}$, where $S_1'(\ell_1) = (\bar{a}_{\ell_1}', s_{\ell_1}')$. Since $S_1' =_\ell S_2'$, $S_2'(\ell_1) = (\bar{a}_{\ell_1}', s_{\ell_1}')$.
Since $\bar{a}_1' =_\ell \bar{a}_2'$, $\bar{a}_1' =_{\star,\ell_1,\bullet,?,c} \bar{a}_2'$. By transitivity of $=_{\star,\ell_1,\bullet,?,c}$, $\bar{a}_{\ell_1}' =_{\star,\ell_1,\bullet,?,c} \bar{a}_2'$. Thus, by setting $S_2 = S_2'[\ell_1 \mapsto (\bar{a}_{\ell_1}, s_{\ell_1})]$, we obtain a derivation of $(\bar{a}_2', \ell_1) \models S_2'(\ell_1) \xrightarrow{c?\star} (\bar{a}_{\ell_1}, s_{\ell_1})$, and thus of the last step in trace $e_2 \models \mathsf{SME}(\sigma, s) \to (\bar{a}_2, \sigma_2, S_2)$ using Figure 5b-rule (i-block). Since $S_1' =_\ell S_2'$, we will get by definition of S_1, S_2 and of $=_\ell$ on objects ranged by S that $S_1 =_\ell S_2$. Also, since $\bar{a}_1' =_\ell \bar{a}_2'$ and $a_1 = a_2 = \bullet$, we will get by definition of $\bar{a}_1$, $\bar{a}_2$, and of $=_\ell$ on traces that $\bar{a}_1 =_\ell \bar{a}_2$.

(i): Here, $a_1 = c?v$ for some c and v, and $\pi(c) = \ell_1$. Let L_1 be the set of security levels $\ell_1 \sqsubseteq \ell'$ for which $(\bar{a}'_1, \ell') \models S'_1(\ell') \xrightarrow{c?v} (\bar{a}^{\ell'}_1, s^{\ell'}_1)$ for some $(\bar{a}^{\ell'}_1, s^{\ell'}_1)$. We then have $\ell_1 \in L_1$, and S_1 defined as $S_1(\ell') = (\bar{a}^{\ell'}_1, s^{\ell'}_1)$ if $\ell' \in L$, and $S_1(\ell') = S'_1(\ell')$ otherwise. For each $\ell' \in L_1$, the only Figure 5a-rule which can constitute the $(\bar{a}'_1, \ell') \models S'_1(\ell') \xrightarrow{c?v} (\bar{a}^{\ell'}_1, s^{\ell'}_1)$ derivation is (i-new). By this rule, $s^{\ell_1}_{1'} \xrightarrow{c?v^{\ell'}_1} s^{\ell_1}_1$ and $\bar{a}^{\ell'}_{1'} =_{\star,\ell_1,\bullet,?,c} \bar{a}'_1$, where $S'_1(\ell_1) = (\bar{a}^{\ell_1}_{1'}, s^{\ell_1}_{1'})$ and $v^{\ell'}_1 = v$ if $\kappa(c) \sqsubseteq \ell'$ or $v = \star$, and $v^{\ell'}_1 = \mathtt{d}$ otherwise.
Since $\bar{a}'_1 =_\ell \bar{a}'_2$, $e_1 \simeq_\ell e_2$ and $\pi(c) = \ell_1 \sqsubseteq \ell$, setting $a_2 = c?v'$ such that $e_2 \models \bar{a}_2$ gives us that $a_1 =_\ell a_2$, and thus by definition of $\bar{a}_1$, $\bar{a}_2$, and of $=_\ell$ on traces that $\bar{a}_1 =_\ell \bar{a}_2$.
Let L_2 be the set of security levels $\ell_1 \sqsubseteq \ell'$ such that $(\bar{a}'_2, \ell') \models S'_2(\ell') \xrightarrow{c?v'} (\bar{a}^{\ell'}_2, s^{\ell'}_2)$ for some $(\bar{a}^{\ell'}_2, s^{\ell'}_2)$. For each $\ell' \in L_2$, the only Figure 5a-rule which can constitute the $(\bar{a}'_2, \ell') \models S'_2(\ell') \xrightarrow{c?v'} (\bar{a}^{\ell'}_2, s^{\ell'}_2)$ derivation is (i-new). By this rule, $s^{\ell_1}_{2'} \xrightarrow{c?v^{\ell'}_2} s^{\ell_1}_2$ and $\bar{a}^{\ell_1}_{2'} =_{?,c,\ell',\bullet} \bar{a}'_2$, where $S'_2(\ell_1) = (\bar{a}^{\ell_1}_{2'}, s^{\ell_1}_{2'})$ and $v^{\ell'}_2 = v'$ if $\kappa(c) \sqsubseteq \ell'$ or $v = \star$, and $v^{\ell'}_2 = \mathtt{d}$ otherwise.
When $v^{\ell'}_1$ and $v^{\ell'}_2$ are both defined and when $\ell' \sqsubseteq \ell$, $v^{\ell'}_1 = v^{\ell'}_2 \overset{\text{def}}{=} v^{\ell'}$; this is seen by regarding the condition placed on $v^{\ell_1}_1$ and $v^{\ell_1}_2$ by (i-new), and that $v' = v$ when $\kappa(c) \sqsubseteq \ell$.
Let S_2 be defined as $S_2(\ell') = (\bar{a}^{\ell'}_2, s^{\ell'}_2)$ if $\ell' \in L_2$, and $S_2(\ell') = S'_2(\ell')$ otherwise. Let $L = \{\ell' \mid \ell_1 \sqsubseteq \ell' \sqsubseteq \ell\}$. Since $(\bar{a}^{\ell'}_{1'}, s^{\ell'}_{1'}) = S'_1(\ell') = S'_2(\ell') = (\bar{a}^{\ell'}_{2'}, s^{\ell'}_{2'})$ for each $\ell' \in L$, $s^{\ell'}_{1'} \xrightarrow{c?v''} s' \iff s^{\ell'}_{2'} \xrightarrow{c?v''} s'$ for each v'' and s'.
We show that $\forall \ell' \,.\, \ell' \in L_1 \cap L \iff \ell' \in L_2 \cap L$. That is, that for each $\ell' \in L$,
$s^{\ell'}_{1'} \xrightarrow{c?v''} \wedge \bar{a}^{\ell'}_1 =_{\star,\ell',\bullet,?,c} \bar{a}'_1$ iff
$s^{\ell'}_{2'} \xrightarrow{c?v''} \wedge \bar{a}^{\ell'}_2 =_{\star,\ell',\bullet,?,c} \bar{a}'_2$.
For the cases of ℓ' for which $s^{\ell'}_{1'} \not\xrightarrow{c?v''}$ for all v'', we have $\ell' \notin L \cap L_1$, and since $s^{\ell'}_{1'} = s^{\ell'}_{2'}$ and thus $s^{\ell'}_{1'} \not\xrightarrow{c?v''}$ for all v'', we get $\ell' \notin L \cap L_2$.
For the cases of ℓ' for which $s^{\ell'}_{1'} \xrightarrow{c?v''} \wedge \bar{a}^{\ell'}_1 \neq_{\star,\ell',\bullet,?,c} \bar{a}'_1$ where $v'' = v$ if $\kappa(c) \sqsubseteq \ell'$ or $v = \star$, and $v'' = \mathtt{d}$ otherwise, we have $\ell' \notin L \cap L_1$ since $(\bar{a}'_1, \ell') \models S'_1(\ell') \xrightarrow{c?v}$ cannot hold. Since $S'_1(\ell') = S'_2(\ell')$, by similar reasoning, $\ell' \ni L \cap L_2$.
For the cases of ℓ' for which $s^{\ell'}_{1'} \xrightarrow{c?v''}$ and $\bar{a}^{\ell'}_1 =_{\star,\ell',\bullet,?,c} \bar{a}'_1$, where $v'' = v$ if $\kappa(c) \sqsubseteq \ell'$ or $v = \star$, and $v'' = \mathtt{d}$ otherwise, we can, by the same casing on $\kappa(c)$ as the one performed in the proof of the (i-block)-case above, establish $\bar{a}^{\ell'}_2 =_{\star,\ell',\bullet,?,c} \bar{a}'_2$.
Thus, $L_1 \cap L = L_2 \cap L = L'$. Since $\ell_1 \in L_1 \cap L$, $\ell_1 \in L'$. Thus, $e_2 \models \mathsf{SME}(\sigma, s) \to (\bar{a}_2, \sigma_2, S_2)$ is derivable with the above definition of $\bar{a}_2$ and S_2 using Figure 5b-rule (i).
It remains to be shown that $S_1 =_\ell S_2$. For each $\ell' \in L'$, $(\bar{a}^{\ell'}_1, s^{\ell'}_1) = (\bar{a}^{\ell'}_2, s^{\ell'}_2)$. Thus, for each $\ell' \in L'$, $S_1(\ell') = S_2(\ell')$. Since $S_1(\ell') = S'_1(\ell')$ and $S_2(\ell') = S'_2(\ell')$ for each $\ell' \sqsubseteq \ell$ for which $\ell' \notin L'$, we get by definition of S_1, S_2 and of $=_\ell$ on objects ranged by S, $S_1 =_\ell S_2$.

(o): Here, $(\bar{a}'_1, \ell_1) \models S'_1(\ell_1) \xrightarrow{c!v_{\ell_1}} (\bar{a}'_{\ell_1}, s'_{\ell_1})$, $S_1 = S'_1[\ell_1 \mapsto (\bar{a}_{\ell_1}, s_{\ell_1})]$, and either $\exists \bar{a}', v^\kappa_1 \,.\, \bar{a}'_1.c!v^{\ell_1}_1 =_{\star,\ell_1,\bullet,!,c} \bar{a}'.c!v^\kappa_1 \leq_{!,c} \bar{a}^\kappa_1$, in which case $v_1 = v^\kappa_1$, or not, in which case $v_1 = v_{\ell_1}$, where $S'_1(\kappa(c)) = (\bar{a}^\kappa_1, \_)$, and $a_1 = c!v_1$. The only Figure 5a-rule which can constitute this derivation is (o-new). By this rule, $\pi(c) = \ell_1$, $s_{\ell_1} \xrightarrow{c!v'_{\ell_1}} s'_{\ell_1}$, and either

$\kappa(c) = \ell_1$, in which case $v'_{\ell_1} = v_{\ell_1}$, or not, in which case $v'_{\ell_1} = \mathtt{d}$, where $S'_1(\ell_1) = (\_, s_{\ell_1})$.

We show that $(\bar{a}'_2, \ell_1) \models S'_2(\ell_1) \xrightarrow{c!v_{\ell_1}} (\bar{a}'_{\ell_1}, s'_{\ell_1})$. This follows immediately from $S'_1(\ell_1) = S'_2(\ell_1)$. Set $S_2 = S'_2[\ell_1 \mapsto (\bar{a}_{\ell_1}, s_{\ell_1})]$. By definition of S_1, S_2 and of $=_\ell$ on objects ranged by S, $S_1 =_\ell S_2$.

Let $v_2 = v_2^\kappa$ if $\exists \bar{a}', v_2^\kappa \,\centerdot\, \bar{a}'_2.c!v_{\ell_1} =_{\star,\ell_1,\bullet,!,c} \bar{a}'.c!v_2^\kappa \leq_{!,c} \bar{a}_2^\kappa$, and $v_2 = v_{\ell_1}$ otherwise, where $S'_2(\kappa(c)) = (\bar{a}_2^\kappa, \_)$. Set $a_2 = c!v_2$. We thus obtain a derivation of the last step in trace $e_2 \models \mathsf{SME}(\sigma, s) \rightarrow (\bar{a}_2, \sigma_2, S_2)$ using Figure 5a-rule (o-new) and Figure 5b-rule (o).

It remains to show $a_1 =_\ell a_2$. Case on $\kappa(c)$.

$\kappa(c) \not\sqsubseteq \ell$: Regardless of the value of v_1 and v_2, $a_1 =_\ell a_2$.

$\kappa(c) \sqsubseteq \ell$: Then since $S'_1 =_\ell S'_2$, $S'_1(\kappa(c)) = S'_2(\kappa(c))$, and thus $\bar{a}_1^\kappa = \bar{a}_2^\kappa$. We thus have $\exists \bar{a}', v_1^\kappa \centerdot \bar{a}'_1.c!v_1^{\ell_1} =_{\star,\ell_1,\bullet,!,c} \bar{a}'.c!v_1^\kappa \leq_{!,c} \bar{a}_1^\kappa$ iff $\exists \bar{a}', v_2^\kappa \centerdot \bar{a}'_2.c!v_{\ell_1} =_{\star,\ell_1,\bullet,!,c} \bar{a}'.c!v_2^\kappa \leq_{!,c} \bar{a}_2^\kappa$ (since $\bar{a}'_1 =_\ell \bar{a}'_2$), with $v_1^\kappa = v_2^\kappa$ when defined. Thus $v_1 = v_2$, and thus $a_1 =_\ell a_2$.

Thus, $\bar{a}_1 =_\ell \bar{a}_2$, as desired.

Since σ_1 and S_1 such that $e_1 \models \mathsf{SME}(\sigma, s) \rightarrow (\bar{a}_1, \sigma_1, S_1)$ were arbitrary, (1) holds for all $\bar{a}_1$ with $|\bar{a}_1| = n + 1$ assuming (1) holds for all $\bar{a}_1$ with $|\bar{a}_1| = n$.

Since (1) holds for arbitrary s, σ, ℓ, e_1 and e_2 such that $e_1 =_\ell e_2$, Lemma 4.3 follows. □

Proof of Theorem 4.2. Follows from Lemma 4.3 since $\bar{a}_1 =_\ell \bar{a}_2 \implies \bar{a}_1 \simeq_\ell \bar{a}_2$. □

Lemma C.1. $\forall e, s, \ell, \bar{a} \centerdot$
$e{\restriction_\ell} \models s \xrightarrow{\bar{a}} \implies \forall \sigma \centerdot \exists S, \bar{a}' \centerdot$
$e \models \mathsf{SME}(\sigma, s) \rightarrow (\_, \_, S) \wedge S(\ell) = (\bar{a}', \_) \wedge \bar{a} =_{\hat{\star}} \bar{a}'$.

Proof. Follows from the definition of $e{\restriction_\ell}$, (i-new), (i-old), (o-old) and (o-new), and from the assumption that s is input-blocking. □

Lemma C.2. $\forall e, s, \sigma, S \centerdot$
$e \models \mathsf{SME}(\sigma, s) \rightarrow (\_, \_, S) \implies \forall \ell, \bar{a} \centerdot$
$S(\ell) = (\bar{a}, \_) \implies \exists \bar{a}' \centerdot$
$e{\restriction_\ell} \models s \xrightarrow{\bar{a}'} \wedge \bar{a} =_{\hat{\star}} \bar{a}'$.

Proof. Follows from the definition of $e{\restriction_\ell}$, (i-new), (i-old), (o-old) and (o-new), and from the assumption that s is input-blocking. □

Lemma C.3. $\forall e, \sigma, s, \bar{a} \centerdot$
$e \models \mathsf{SME}(\sigma, s) \rightarrow (\bar{a}, \_, S) \implies$
$\forall \ell, \bar{a}_\ell \centerdot S(\ell) = (\bar{a}_\ell, \_) \implies \bar{a} =_{\pi^{-1}(\ell),\ell,\bullet} \bar{a}_\ell$

Proof. Follows from (i) and (o). □

Proof of Theorem 4.4. We prove a); the proof of b) is similar. Assume $s \in \text{PSNI}$. Let e and $\bar{a}$ be arbitrary such that $e \models s \xrightarrow{\bar{a}}$. Since $s \in \text{PSNI}$ and $e \approx_\ell e\lceil_\ell$ for all ℓ, we have for all ℓ that $e\lceil_\ell \models s \xrightarrow{\bar{a}_\ell}$ and $\bar{a} \approx_\ell \bar{a}_\ell$ for some $\bar{a}_\ell$. By Lemma C.1, for each ℓ, we have for any σ some S_ℓ for which $e \models \mathsf{SME}(\sigma, s) \to (\_, \_, S_\ell)$ that $\bar{a}_\ell =_{\hat{\star}} \bar{a}'_\ell$ for some $\bar{a}'_\ell$ for which $S_\ell(\ell) = (\bar{a}'_\ell, \_)$. Since $(=_{\hat{\star}}) \subseteq (\approx)$, $\bar{a}_\ell \approx_\ell \bar{a}'_\ell$. By transitivity, $\bar{a} \approx_\ell \bar{a}'_\ell$. Since $\mathsf{SME}(\sigma, s)$ is deterministic, we have for some S'_ℓ that $\bar{a}'_\ell \leq \bar{a}''_\ell$ for all ℓ, where $S'_\ell(\ell) = (\bar{a}''_\ell, \_)$. Let $\bar{a}'$ be such that $e \models \mathsf{SME}(\sigma, s) \to (\bar{a}', \_, S'_\ell)$. By Lemma C.3, $\bar{a}''_\ell =_{\pi^{-1}(\ell),\ell,\bullet} \bar{a}'$. From $\bar{a}'_\ell \leq \bar{a}''_\ell$ we get $\bar{a}'_\ell \leq_{\star,\ell,\pi^{-1}(\ell),\bullet} \bar{a}''_\ell$. Since $\bar{a} \approx_\ell \bar{a}'_\ell$, we get $\bar{a} =_{\star,\ell,\pi^{-1}(\ell),\bullet} \bar{a}'_\ell$. Together this yields $\bar{a} \leq_{\star,\ell,\pi^{-1}(\ell),\bullet} \bar{a}'$. $\square$

Proof of Theorem 4.6. We prove a); the proof of b) is similar.

When no output is produced on channels c for which $\pi(c) \sqsubset \kappa(c)$, the result follows from Theorem 4.4 since both s and $\mathsf{SME}(\sigma, s)$ are run under the same input stream (and thus will read the same nth input (for all n) on all channels, including ones with $\pi(c) \sqsubset \kappa(c)$). So we need only show that the nth output (for any n) on any channel c with $\pi(c) \sqsubset \kappa(c)$ in $\bar{a}$ has the same value as the nth c-output in $\bar{a}'$. This follows from Definition 4.5, (o), and the observation that, in any trace of s, after the first occurrence of an input action on a channel c, the remainder of the trace is silent to all observers at levels ℓ' which do not satisfy $\pi(c) \sqsubseteq \ell'$ (thus, the $\kappa(c)$-run cannot be delayed to produce a c output by reading blanks on channels c' for which $\pi(c) \sqsubset \pi(c') \sqsubset \kappa(c)$). $\square$

Proof of Lemma 5.2. Let $\mathcal{R} \in \{\simeq, \approx\}$, ℓ, s, e, $\bar{a}$, and a be given. Assume $e' \in k^{\mathcal{R}}_\ell(s, e, \bar{a}.a)$. We must show that $e' \in k^{\mathcal{R}}_\ell(s, e, \bar{a})$. Since $e' \in k^{\mathcal{R}}_\ell(s, e, \bar{a}.a)$, there must be some $\bar{a}'$ for which $e' \models s \xrightarrow{\bar{a}'}$ and $\bar{a}.a\, \mathcal{R}_\ell\, \bar{a}'$. By definition of $\mathcal{R}$, for every prefix of $\bar{a}.a$, there is a prefix of $\bar{a}'$ which is $\mathcal{R}_\ell$-equivalent to it. Thus, there is some prefix $\bar{a}''$ of $\bar{a}'$ for which $\bar{a}\, \mathcal{R}_\ell\, \bar{a}''$. Thus $e' \in k^{\mathcal{R}}_\ell(s, e, \bar{a})$. $\square$

Proof of Theorem 5.4. Let $\mathcal{R} \in \{\simeq, \approx\}$ and s be given. We prove each direction of the biimplication.

$\Longleftarrow$ **:** Prove contrapositive. Assume $s \notin \text{NI}^{\mathcal{R}}_k$. Must show that $s \notin \text{NI}^{\mathcal{R}}$. By Lemma 5.2, there must be a $\ell, e, \bar{a}, a, e'$ for which $e\, \mathcal{R}_\ell\, e'$, $e \models s \xrightarrow{\bar{a}}$, $e \models s \xrightarrow{\bar{a}.a}$, $e' \models s \xrightarrow{\bar{a}}$ and $e' \models s \not\xrightarrow{\bar{a}.a}$. By setting $e_1 = e$, $e_2 = e'$ and $\bar{a}_1 = \bar{a}.a$, we get a counterexample to $s \in \text{NI}^{\mathcal{R}}$.

$\Longrightarrow$ **:** Prove contrapositive. Assume $s \notin \text{NI}^{\mathcal{R}}$. Must show that $s \notin \text{NI}^{\mathcal{R}}_k$. Let $\bar{a}_1$ be shortest such that for some e_1 and e_2 for which $e_1\, \mathcal{R}_\ell\, e_2$, $e_1 \models s \xrightarrow{\bar{a}_1}$ but no $\bar{a}_2$ exists for which $e_2 \models s \xrightarrow{\bar{a}_2}$ and $\bar{a}_1\, \mathcal{R}_\ell\, \bar{a}_2$. Since $e \models s \xrightarrow{\epsilon}$ holds for all e and s, $\bar{a}_1 = \bar{a}'_1.a_1$ for some $\bar{a}'_1$ and a_1. Since $\bar{a}_1$ is shortest, $e_2 \models s \xrightarrow{\bar{a}'_1}$. Thus $e_2 \in k^{\mathcal{R}}_\ell(s, e_1, \bar{a}'_1)$ and $e_2 \notin k^{\mathcal{R}}_\ell(s, e_1, \bar{a}'_1.a_1)$, a counterexample to $s \in \text{NI}^{\mathcal{R}}_k$.

$\square$

In the following lemma, $S_1 =_\ell S_2$ iff $\forall \ell' \sqsubseteq^\rho \ell \,.\, S_1(\ell') = S_2(\ell')$.

Lemma C.4. $\forall \delta, \sigma, s, \ell, e_1, e_2 \,.\, e_1 \simeq^{\rho(\delta)}_\ell e_2 \implies$
$\forall \bar{a}_1, \sigma_1, S_1 \,.\, e_1 \models \mathsf{SME_D}(\delta, \sigma, s) \to (\bar{a}_1, \sigma_1, S_1) \implies$
$\exists \bar{a}_2, \sigma_2, S_2 \,.\, e_2 \models \mathsf{SME_D}(\delta, \sigma, s) \to (\bar{a}_2, \sigma_2, S_2) \wedge$
$\bar{a}_1 =_\ell \bar{a}_2 \;\wedge\; S_1 =_\ell S_2 \;\wedge \sigma_1 = \sigma_2$

Proof of Lemma C.4. Let δ, ρ, σ, s, ℓ, e_1 and e_2 such that $\rho = \rho(\delta)$ and $e_1 \simeq^\rho_\ell e_2$ be given. We prove that

$$\begin{aligned}
&\forall \bar{a}_1, \sigma_1, S_1 \,.\, e_1 \models \mathsf{SME_D}(\delta, \sigma, s) \to (\bar{a}_1, \sigma_1, S_1) \implies \\
&\exists \bar{a}_2, \sigma_2, S_2 \,.\, e_2 \models \mathsf{SME_D}(\delta, \sigma, s) \to (\bar{a}_2, \sigma_2, S_2) \wedge \qquad (1)\\
&\bar{a}_1 =_\ell \bar{a}_2 \;\wedge\; S_1 =_\ell S_2 \;\wedge \sigma_1 = \sigma_2
\end{aligned}$$

by induction in $n = |\bar{a}_1|$. Let σ_1 and S_1 be given such that $e_1 \models \mathsf{SME}_{\mathtt{D}}(\delta, \sigma, s) \to (\bar{a}_1, \sigma_1, S_1)$.

Since the semantics of $\mathsf{SME}_{\mathtt{D}}$ consists of inference rules added to the semantics of SME, the remainder of this proof is an addition to the proof of Lemma 4.3, with the outermost casing replaced with $\ell_1 \sqsubseteq^\rho \ell$ contra $\neg(\ell_1 \sqsubseteq^\rho \ell)$. When $\ell_1 \sqsubseteq^\rho \ell$ and $\ell_1 \not\sqsubseteq \ell$, $\bar{a}_1 =_\ell \bar{a}_2$ still holds since each rule in the SME semantics which produces I/O produces I/O on channels with presence level ℓ_1; therefore, $a_1 =_\ell a_2 =_\ell \bullet$.

Add the following cases to the $\ell_1 \sqsubseteq^\rho \ell$ case. In all these cases, $a_1 = \bullet$, and thus $\bar{a}_1 = \bar{a}_1'.\bullet$.

($\bullet$): Add the following case to the Figure 9a rule casing:

(D-o): Since $S_1'(\ell_1) = S_2'(\ell_1) = (\bar{a}_{\ell_1}^{1'}, s_{\ell_1}^{1'})$ and since (D-o) only conditions on $s_{\ell_1}^{1'}$, we get (†).

($\mathtt{D}$-no): Here, $(\bar{a}_1, \ell_1) \models S_1'(\ell_1) \xrightarrow{c_{\mathtt{D}}?v} (\bar{a}_1^{\ell_1}, s_1^{\ell_1})$ for some $(\bar{a}_1^{\ell_1}, s_1^{\ell_1})$, $c_{\mathtt{D}} \notin \delta_{\ell_1}$, and $S_1 = S_1'[\ell_1 \mapsto (\bar{a}_1^{\ell_1}, s_1^{\ell_1})]$. The only Figure 9a rule which can constitute this derivation is (D-i). Since (D-i) only conditions on states, and since $S_1'(\ell_1) = S_2'(\ell_1)$, we get that $(\bar{a}_1, \ell_1) \models S_2'(\ell_1) \xrightarrow{c_{\mathtt{D}}?v} (\bar{a}_1^{\ell_1}, s_1^{\ell_1})$. By setting $S_2 = S_2'[\ell_1 \mapsto (\bar{a}_1^{\ell_1}, s_1^{\ell_1})]$, we obtain a derivation of the last step in $e_2 \models \mathsf{SME}_{\mathtt{D}}(\delta, \sigma, s) \to (\bar{a}_2, \sigma_2, S_2)$ through rules ($\mathtt{D}$-no) and (D-i). Thus, $a_2 = \bullet = a_1$, and since $\bar{a}_1' =_\ell \bar{a}_2'$, we get $\bar{a}_1 =_\ell \bar{a}_2$. Since $S_1' =_\ell S_2'$, we get by definition of S_1 and S_2 that $S_1 =_\ell S_2$.

($\mathtt{D}$-yes$_{\mathtt{d}}$): Here, $(\bar{a}_1, \ell_1) \models S_1'(\ell_1) \xrightarrow{c_{\mathtt{D}}?\mathtt{d}} (\bar{a}_1^{\ell_1}, s_1^{\ell_1})$ for some $(\bar{a}_1^{\ell_1}, s_1^{\ell_1})$, $c_{\mathtt{D}} \in \delta_{\ell_1}$, $S_1 = S_1'[\ell_1 \mapsto (\bar{a}_1^{\ell_1}, s_1^{\ell_1})]$ and for $S_1'(\varphi(c_{\mathtt{D}})) = (\bar{a}_1^{\varphi}, \_)$ and $S_1'(\ell_1) = (\bar{a}_{1'}^{\ell_1}, \_)$, $|\bar{a}_1^{\varphi} \upharpoonright_{!, c_{\mathtt{D}}, \bullet}| < |\bar{a}_{1'}^{\ell_1} \upharpoonright_{!, c_{\mathtt{D}}, \bullet}|$. The only Figure 9a rule which can constitute this derivation is (D-i). Since (D-i) only conditions on states, and since $S_1'(\ell_1) = S_2'(\ell_1)$, we get that $(\bar{a}_1, \ell_1) \models S_2'(\ell_1) \xrightarrow{c_{\mathtt{D}}?\mathtt{d}} (\bar{a}_1^{\ell_1}, s_1^{\ell_1})$. Since $\ell_1 \sqsubseteq^\rho \ell$ and $c_{\mathtt{D}} \in \delta_{\ell_1}$, we get by definition of ρ that $\varphi(c_{\mathtt{D}}) \sqsubseteq^\rho \ell$. Thus, since $S_1' =_\ell S_2'$, $S_1'(\varphi(c_{\mathtt{D}})) = S_2'(\varphi(c_{\mathtt{D}}))$. By setting $S_2 = S_2'[\ell_1 \mapsto (\bar{a}_1^{\ell_1}, s_1^{\ell_1})]$, we obtain a derivation of the last step in $e_2 \models \mathsf{SME}_{\mathtt{D}}(\delta, \sigma, s) \to (\bar{a}_2, \sigma_2, S_2)$ through rules ($\mathtt{D}$-yes$_{\mathtt{d}}$) and (D-i). Thus, $a_2 = \bullet = a_1$, and since $\bar{a}_1' =_\ell \bar{a}_2'$, we get $\bar{a}_1 =_\ell \bar{a}_2$. Since $S_1' =_\ell S_2'$, we get by definition of S_1 and S_2 that $S_1 =_\ell S_2$.

($\mathtt{D}$-yes): Here, $(\bar{a}_1, \ell_1) \models S_1'(\ell_1) \xrightarrow{c_{\mathtt{D}}?v_1} (\bar{a}_1^{\ell_1}, s_1^{\ell_1})$ for some $c_{\mathtt{D}}$, v_1, $(\bar{a}_1^{\ell_1}, s_1^{\ell_1})$, $c_{\mathtt{D}} \in \delta_{\ell_1}$, $S_1 = S_1'[\ell_1 \mapsto (\bar{a}_1^{\ell_1}, s_1^{\ell_1})]$ and for $S_1'(\varphi(c_{\mathtt{D}})) = (\bar{a}_1^{\varphi}.c_{\mathtt{D}}!v_{\varphi}.\_, \_)$ and $S_1'(\ell_1) = (\bar{a}_{1'}^{\ell_1}.c_{\mathtt{D}}!v_{\ell_1}, \_)$, $|\bar{a}_1^{\varphi} \upharpoonright_{!, c_{\mathtt{D}}, \bullet}| = |\bar{a}_{1'}^{\ell_1} \upharpoonright_{!, c_{\mathtt{D}}, \bullet}|$. The only Figure 9a rule which can constitute this derivation is (D-i). Since (D-i) only conditions on states, and since $S_1'(\ell_1) = S_2'(\ell_1)$, we get that $(\bar{a}_1, \ell_1) \models S_2'(\ell_1) \xrightarrow{c_{\mathtt{D}}?v_2} (\bar{a}_1^{\ell_1}, s_1^{\ell_1})$. Since $\ell_1 \sqsubseteq^\rho \ell$ and $c_{\mathtt{D}} \in \delta_{\ell_1}$, we get by definition of ρ that $\varphi(c_{\mathtt{D}}) \sqsubseteq^\rho \ell$. Thus, since $S_1' =_\ell S_2'$, $S_1'(\varphi(c_{\mathtt{D}})) = S_2'(\varphi(c_{\mathtt{D}}))$. Depending on the "if"-statement in ($\mathtt{D}$-yes), then either $v_1 = \mathtt{d}$ or $v_1 = v_\varphi$. If the "if"-statement is true for some $\ell_{\mathtt{d}}$ in the derivation with state S_1', the "if"-statement will be true for the same $\ell_{\mathtt{d}}$ in the derivation with state S_2', since $S_1' =_\ell S_2'$ and since $c_{\mathtt{D}} \in \delta_{\ell_{\mathtt{d}}}$ implies that $\ell_{\mathtt{d}} \sqsubseteq^\rho \ell$. Thus, regardless of which value v_1 has, by setting $v_2 = v_1$, and by setting $S_2 = S_2'[\ell_1 \mapsto (\bar{a}_1^{\ell_1}, s_1^{\ell_1})]$, we obtain a derivation of the last step in $e_2 \models \mathsf{SME}_{\mathtt{D}}(\delta, \sigma, s) \to (\bar{a}_2, \sigma_2, S_2)$ through rules ($\mathtt{D}$-yes) and (D-i). Thus, $a_2 = \bullet = a_1$, and since $\bar{a}_1' =_\ell \bar{a}_2'$, we get $\bar{a}_1 =_\ell \bar{a}_2$. Since $S_1' =_\ell S_2'$, we get by definition of S_1 and S_2 that $S_1 =_\ell S_2$.

□

Proof of Theorem 5.17. Follows from Lemma C.4 since $\bar{a}_1 =_\ell \bar{a}_2 \implies \bar{a}_1 \simeq_\ell \bar{a}_2$. □

Proof of Corollary 5.13. Follows by comparison of Definitions 5.12 and 3.4 with $\rho = \emptyset$. □

Proof of Corollary 5.18. Since $s \notin \mathtt{LTS}_{\mathtt{IO}}^{\mathtt{D}}$, derivations of any action in any trace of $\mathsf{SME}_{\mathtt{D}}(\rho, \sigma, s)$ never use rules from Figure 9. Thus, $\mathsf{SME}_{\mathtt{D}}(\rho, \sigma, s)$ behaves like $\mathsf{SME}(\sigma, s)$. By Theorem 4.2, $\mathsf{SME}(\sigma, s) \in \mathsf{TSNI}$. Therefore, $\mathsf{SME}_{\mathtt{D}}(\rho, \sigma, s) \in \mathsf{TSNI}$. □

Proof of Lemma 5.19. Let δ, σ, s, e_1, $\bar{a}_1$, σ_1 and S_1 such that $e_1 \models \mathsf{SME}_\mathtt{R}(\sigma, s, \delta) \rightarrow (\bar{a}_1, \sigma_1, S_1)$ be arbitrary. Let ℓ, e_2 such that $e_2 \in k_\ell(\mathsf{SME}_\mathtt{R}(\sigma, s, \delta), e_1, \bar{a}_1)$ be arbitrary. We show there exist $\bar{a}_2$ and S_2 such that $e_2 \models \mathsf{SME}_\mathtt{R}(\sigma, s, \delta) \rightarrow (\bar{a}_2, \sigma_1, S_2)$, $\bar{a}_1 =_\ell \bar{a}_2$ and $S_1 =_\ell S_2$. We do this by induction in $n = |\bar{a}_1|$.

$n = 0$**:** Then $\bar{a}_2 = \epsilon$. For any s' and e', it follows by definition of $e' \models s' \rightarrow$ that $e' \models s' \xrightarrow{\epsilon} s'$. Let $\mathsf{SME}_\mathtt{R}(\sigma, s, \delta) = (\epsilon, \sigma, S)$. Then $\sigma_1 = \sigma$ and $S_1 = S$, and for $S_2 = S$ and $\bar{a}_2 = \epsilon$, we have $e_2 \models \mathsf{SME}_\mathtt{R}(\sigma, s, \delta) \rightarrow (\bar{a}_2, \sigma_1, S_2)$, $\bar{a}_1 = \bar{a}_2$ and $S_1 = S_2$. Since $(=) \subseteq (=_\ell)$, we have $\bar{a}_1 =_\ell \bar{a}_2$ and $S_1 =_\ell S_2$.

$n + 1$ **given** n**:** Assume Lemma 5.19 holds for all $\bar{a}_1'$ for which $|\bar{a}_1'| = n$; this is the induction hypothesis (IH). Let $\bar{a}_1$ be such that $|\bar{a}_1| = n + 1$. Then $\bar{a}_1 = \bar{a}_1'.a_1$ for some $\bar{a}_1'$ and a_1. By Lemma 5.2, we have that $e_1, e_2 \in k_\ell(\mathsf{SME}_\mathtt{R}(\sigma, s, \delta), e_1, \bar{a}_1')$. Since $e_1 \models \mathsf{SME}_\mathtt{R}(\sigma, s, \delta) \rightarrow (\bar{a}_1, \sigma_1, S_1)$ and $\bar{a}_1 = \bar{a}_1'.a_1$, we have $e_1 \models \mathsf{SME}_\mathtt{R}(\sigma, s, \delta) \rightarrow (\bar{a}_1', \sigma_1', S_1')$ for some σ_1' for which $\sigma_1' = \ell_1.\sigma_1$ for some ℓ_1. By (IH), we have for some $\bar{a}_2'$ and S_2' that $e_2 \models \mathsf{SME}_\mathtt{R}(\sigma, s, \delta) \rightarrow (\bar{a}_2', \sigma_1', S_2')$, $\bar{a}_1' =_\ell \bar{a}_2'$ and $S_1' =_\ell S_2'$. Let a_2 and S_2 be defined such that $e_2 \models \mathsf{SME}_\mathtt{R}(\sigma, s, \delta) \rightarrow (\bar{a}_2'.a_2, \sigma_1, S_2) \rightarrow$ (they are unique), and set $\bar{a}_2 = \bar{a}_2'.a_2$. The proof reduces to establishing $a_1 =_\ell a_2$ and $S_1 =_\ell S_2$. Case on ℓ_1.

$\ell_1 \not\sqsubseteq \ell$**:** Then neither a_1 nor a_2 are observable actions, since, by $\mathsf{SME}_\mathtt{R}$, all non-$\bullet$ actions $\hat{a}$ an ℓ_1-run can cause $\mathsf{SME}_\mathtt{R}$ to perform have $\pi(\hat{a}) = \ell_1$. So $\pi(a_1) \not\sqsubseteq \ell$ and $\pi(a_2) \not\sqsubseteq \ell$, so $a_1 =_\ell a_2$. For each inference rule of $\mathsf{SME}_\mathtt{R}$, only the state of the ℓ_1-run is modified, except for rule (i), which modifies only states of $(\ell_1 \sqsubseteq)$-runs. Thus $S_1' =_\ell S_1$ and $S_2' =_\ell S_2$, and by transitivity, $S_1 =_\ell S_2$.

$\ell_1 \sqsubseteq \ell$**:** Then since $S_1' =_\ell S_2'$, we have $S_1'(\ell_1) = S_2'(\ell_1) =: (\hat{a}, \hat{s})$. Since $\hat{s}$ is deterministic and input blocking, if $\hat{s}$ wants to do a production as its next action, then only that action is enabled in $\hat{s}$. If $\hat{s}$ wants to input on a channel, then only input on that channel are enabled in $\hat{s}$. Case on rule used to derive action a_1.

(D-yes$_\mathtt{d}$) **or** (D-yes)**:** Then $a_1 = c_\mathtt{R}!v_1$ for some $c_\mathtt{R}$ and v_1. Then $\hat{s} \xrightarrow{c_\mathtt{D}?v_1}$ for $c_\mathtt{D}$ for which $\varrho(c_\mathtt{D}, \ell_1) = c_\mathtt{R}$. Then (D-yes$_\mathtt{d}$) or (D-yes) is used to derive a_2. So $a_2 = c_\mathtt{R}!v_2$ for some v_2. Since $\pi(c_\mathtt{R}) = \kappa(c_\mathtt{R}) = \ell_1$ and since $\mathsf{SME}_\mathtt{R}(\sigma, s, \delta)$ is deterministic, $v_1 = v_2$ must hold; otherwise, there is no $\bar{a}'$ for which $e_2 \models \mathsf{SME}_\mathtt{R}(\sigma, s, \delta) \xrightarrow{\bar{a}'}$ and $\bar{a}_1 \simeq_\ell \bar{a}'$, contradicting the assumption that $e_2 \in k_\ell(\mathsf{SME}_\mathtt{R}(\sigma, s, \delta), e_1, \bar{a}_1)$. Thus $v_1 = v_2$, and thus $a_1 = a_2$. Since $\hat{s}$ is deterministic, $S_1 = S_1'[\ell_1 \mapsto (\hat{a}.c_\mathtt{D}?v, \hat{s}')]$ and $S_2 = S_2'[\ell_1 \mapsto (\hat{a}.c_\mathtt{D}?v', \hat{s}')]$. By this, since $S_1' =_\ell S_2'$ and since $S_1(\ell_1) = S_2(\ell_2)$, $S_1 =_\ell S_2$.

(D-no)**:** Then $a_1 = \bullet$. Also, since $S_1'(\ell_1) = S_2'(\ell_1)$, then by the same rule, $a_2 = \bullet$. We have $S_1 = S_1'[\ell_1 \mapsto (\hat{a}.a_1', \hat{s}_1')]$ and $S_2 = S_2'[\ell_1 \mapsto (\hat{a}.a_2', \hat{s}_2')]$ for some a_1', a_2', $\hat{s}_1'$ and $\hat{s}_2'$. Since $(\hat{a}.a_1', \hat{s}_1')]$, resp. $(\hat{a}.a_2', \hat{s}_2')]$, is a function of $S_1'(\ell_1)$, resp. $S_2'(\ell_2)$, and $S_1'(\ell_1) = S_1'(\ell_1)$, we have that $S_1(\ell_1) = S_1(\ell_1)$. By definition of S_1 and S_2 and since $S_1' =_\ell S_2'$, we get $S_1 =_\ell S_2$.

(i-block)**:** Then $a_1 = \bullet$ and some c with $\pi(c) \sqsubset \ell_a$, $\hat{s} \xrightarrow{c?\star}$. Since $\bar{a}_1' \simeq_\ell \bar{a}_2'$, both have the same number of c input. Thus, since (i-new) was used to derive a_1, and since (i-old) and (i-new) are mutually exclusive, only (i-new) and (i-block) can be used to derive a_2. Thus $a_2 = \bullet$, so $a_1 \simeq_\ell a_2$. Also, since $S_1'(\ell_1) = S_1'(\ell_1)$ and since $c?\star$ is provided as input to the ℓ_1-run both when deriving a_1 and a_2, $S_1(\ell_1) = S_1(\ell_1)$. By definition of S_1 and S_2 and since $S_1' =_\ell S_2'$, we get $S_1 =_\ell S_2$.

(silent)**:** Then $a_1 = \bullet$. If either (dead), (silent) or (o-old) was used to derive a_1, then (silent) and the same rule can be used to derive a_2. We show that if (i-old) was used to derive a_1, then (silent) and that same rule is used to derive a_2. If the c input on is unobservable, then this clearly holds. If the c input on is observable, then since $\bar{a}'_1$ and $\bar{a}'_2$ have the same number of c input since $\bar{a}'_1 \simeq_\ell \bar{a}'_2$, this holds. Thus $a_2 = \bullet$, so $a_1 \simeq_\ell a_2$. Since $S'_1(\ell_1) = S'_1(\ell_1)$ and since the same rules were used to derive a_1 and a_2, $S_1(\ell_1) = S_1(\ell_1)$. By definition of S_1 and S_2 and since $S'_1 =_\ell S'_2$, we get $S_1 =_\ell S_2$.

(i)**:** Then $a_1 = c?v$ and $\pi(c) = \ell_1$. Since $\bar{a}'_1 =_\ell \bar{a}'_2$ and $e_1 \simeq_\ell e_2$, $e' \models \bar{a}'_2.c?v'$ for some v', and $v = \star \iff v' = \star$. If $v = \star$, then $v' = \star$, thus $a_1 \simeq_\ell a_2$. Since $S'_1(\ell_1) = S'_1(\ell_1)$ and since the same rules were used to derive a_1 and a_2, $S_1(\ell_1) = S_1(\ell_1)$. If instead $v \neq \star$, then if content is unobservable, $a_1 \simeq_\ell a_2$ regardless of what value v' has. Since $S'_1(\ell_1) = S'_1(\ell_1)$ and since dummy values are fed to the ℓ_1-runs, $S_1(\ell_1) = S_1(\ell_1)$. By definition of S_1 and S_2 and since $S'_1 =_\ell S'_2$, we get $S_1 =_\ell S_2$. If instead content is observable, then $v = v'$ since $\bar{a}'_1 =_\ell \bar{a}$ $e_1 \simeq_\ell e_2$. So $a \simeq_\ell a'$. Since $S'_1(\ell_1) = S'_1(\ell_1)$ and since the same value is fed to the ℓ_1-runs, $S_1(\ell_1) = S_1(\ell_1)$. In all these cases, for each ℓ' for which $\ell_1 \sqsubseteq \ell'_1 \sqsubseteq \ell$, an argument similar to the above proves that $S'_1(\ell'_1)$ is waiting for input on c iff $S'_2(\ell'_1)$ is waiting for input on c, and for each of these ℓ'_1, the ℓ'_1-runs get the same input. So by definition of S_1 and S_2 and since $S'_1 =_\ell S'_2$, we get $S_1 =_\ell S_2$.

(o)**:** Then $a_1 = o$ and $\pi(a_1) = \ell_1$. Then (o-old) is also used to derive a_2. Both a_1 and a_2 are outputs on the same channel. If content is unobservable, $a_1 \simeq_\ell a_2$. If content is observable, the content is the same by $S'_1(\ell_1) = S'_2(\ell_1)$ (and the content provider has produced the value in $S'_1(\ell_1)$ iff it has in $S'_2(\ell_1)$). Thus $a \simeq_\ell a'$. Also, for $S_1(\ell_1) = (\hat{\bar{a}}.a'_1, \hat{s}'_1)$ and $S_2(\ell_1) = (\hat{\bar{a}}.a'_2, \hat{s}'_2)$, since $(\hat{\bar{a}}.a'_1, \hat{s}'_1)$, resp. $(\hat{\bar{a}}.a'_2, \hat{s}'_2)$, is a function of $S'_1(\ell_1)$, resp. $S'_2(\ell_2)$, and $S'_1(\ell_1) = S'_1(\ell_1)$, we have that $S_1(\ell_1) = S_1(\ell_1)$. By definition of S_1 and S_2 and since $S'_1 =_\ell S'_2$, we get $S_1 =_\ell S_2$.

□

Proof of Theorem 5.20. Let δ, σ, s be arbitrary. Let ℓ, $a \notin \mathbb{A}^\ell_{\mathrm{R}}$, e and $\bar{a}$ such that $e \models \mathsf{SME}_{\mathrm{R}}(\sigma, s, \delta) \xrightarrow{\bar{a}.a}$ be arbitrary. Let $e' \in k_\ell(\mathsf{SME}_{\mathrm{R}}(\sigma, s, \delta), e, \bar{a})$ be arbitrary. We show that $e' \in k_\ell(\mathsf{SME}_{\mathrm{R}}(\sigma, s, \delta), e, \bar{a}.a)$; then, by Lemma 5.2, $k_\ell(\mathsf{SME}_{\mathrm{R}}(\sigma, s, \delta), e, \bar{a}) = k_\ell(\mathsf{SME}_{\mathrm{R}}(\sigma, s, \delta), e, \bar{a}.a)$. Since $e \models \mathsf{SME}_{\mathrm{R}}(\sigma, s, \delta) \xrightarrow{\bar{a}.a}$, we have for some σ, ℓ_a and S that $e \models \mathsf{SME}_{\mathrm{R}}(\sigma, s, \delta) \rightarrow (\bar{a}, \ell_a.\sigma, S)$. Since $e' \in k_\ell(\mathsf{SME}_{\mathrm{R}}(\sigma, s, \delta), e, \bar{a})$, we have by Lemma 5.19 some $\bar{a}'$ and S' for which $e' \models \mathsf{SME}_{\mathrm{R}}(\sigma, s, \delta) \rightarrow (\bar{a}', \ell_a.\sigma, S')$, $\bar{a}' =_\ell \bar{a}$ and $S' =_\ell S$. Let a' be defined such that $e' \models \mathsf{SME}_{\mathrm{R}}(\sigma, s, \delta) \rightarrow (\bar{a}', \ell_a.\sigma, S') \xrightarrow{a'}$ (it is unique). The proof reduces to establishing $a \simeq_\ell a'$. Case on ℓ_a.

$\ell_a \not\sqsubseteq \ell$**:** Then neither a nor a' are observable actions, since, by $\mathsf{SME}_{\mathrm{R}}$, all non-$\bullet$ actions $\hat{a}$ an ℓ_a-run can cause $\mathsf{SME}_{\mathrm{R}}$ to perform have $\pi(\hat{a}) = \ell_a$. So $\pi(a) \not\sqsubseteq \ell$ and $\pi(a') \not\sqsubseteq \ell$. So $a \simeq_\ell a'$.

$\ell_a \sqsubseteq \ell$**:** Then since $S' =_\ell S$, we have $S(\ell_a) = S'(\ell_a) =: (\hat{\bar{a}}, \hat{s})$. Since $\hat{s}$ is deterministic and input blocking, if $\hat{s}$ wants to do a production as its next action, then only that action is enabled in $\hat{s}$. If $\hat{s}$ wants to input on a channel, then only input on that channel are enabled in $\hat{s}$. Case on rule used to derive action a.

(D-yes$_{\mathtt{d}}$) **or** (D-yes)**:** Impossible since $a \notin \mathbb{A}^\ell_{\mathrm{R}}$.

(D-no)**:** Then $a = \bullet$. Also, by the same rule, $a' = \bullet$, so $a \simeq_\ell a'$.

(i-block)**:** Then $a = \bullet$ and some c with $\pi(c) \sqsubset \ell_a$, $\hat{s} \xrightarrow{c?\star}$. Since $\bar{a} \simeq_\ell \bar{a}'$, both have the same number of c input. Thus, since (i-new) was used to derive a, and since (i-old) and (i-new) are mutually exclusive, only (i-new) and (i-block) can be used to derive a'. Thus $a' = \bullet$, so $a \simeq_\ell a'$.

(silent)**:** Then $a = \bullet$. If either (dead), (silent) or (o-old) was used to derive a, then (silent) and the same rule can be used to derive a'. We show that if (i-old) was used to derive a, then (silent) and that same rule is used to derive a'. If the c input on is unobservable, then this clearly holds. If the c input on is observable, then since $\bar{a}$ and $\bar{a}'$ have the same number of c input since $\bar{a} \simeq_\ell \bar{a}'$, this holds. Thus $a' = \bullet$, so $a \simeq_\ell a'$.

(i)**:** Then $a = c?v$ and $\pi(c) = \ell_a$. Since $\bar{a}' =_\ell \bar{a}$ and $e \simeq_\ell e'$, $e' \models \bar{a}'.c?v'$ for some v', and $v = \star \iff v' = \star$. If $v = \star$, then $v' = \star$, thus $a \simeq_\ell a'$. If $v \neq \star$, then if content is unobservable, $a \simeq_\ell a'$ regardless of what value v' has. If content is observable, then $v = v'$ since $\bar{a}' =_\ell \bar{a}$ and $e \simeq_\ell e'$. So $a \simeq_\ell a'$.

(o)**:** Then $a = o$ and $\pi(a) = \ell_a$. Then (o-old) is also used to derive a'. Both a and a' are outputs on the same channel. If content is unobservable, $a \simeq_\ell a'$. If content is observable, the content is the same by $S' =_\ell S$. Thus $a \simeq_\ell a'$.

□

Let $\mathsf{B}(e, s)$ behave like s except when s performs input on a declassification channel; that input is then drawn from e. In effect, B binds declassification channels internally. Let $\mathsf{B}(e, s) = \mathsf{B}(\epsilon, e, s)$.

$$\frac{s \xrightarrow{o} s' \quad o \in \mathbb{A}_{\mathtt{D}}}{\mathsf{B}(\bar{a}, e, s) \xrightarrow{\bullet} \mathsf{B}(\bar{a}.o, e, s')} \qquad \frac{s \xrightarrow{i} s' \quad i \in \mathbb{A}_{\mathtt{D}} \quad e \models \bar{a}.i}{\mathsf{B}(\bar{a}, e, s) \xrightarrow{\bullet} \mathsf{B}(\bar{a}.i, e, s')} \qquad \frac{s \xrightarrow{a} s' \quad a \notin \mathbb{A}_{\mathtt{D}}}{\mathsf{B}(\bar{a}, e, s) \xrightarrow{a} \mathsf{B}(\bar{a}.a, e, s')}$$

Proof of Theorem 5.23. We prove a); the proof of b) is similar. Assume $s \in \textsc{tsni}$. Let e and $\bar{a}$ be arbitrary such that $e \models \mathsf{F}(s) \xrightarrow{\bar{a}}$. Since s is deterministic, any trace emitted by s prefixes a unique (possibly infinite) sequence of actions. Let e' be e with all $c_{\mathtt{D}}$ inputs, for any $c_{\mathtt{D}}$, replaced by the $c_{\mathtt{D}}$ inputs emitted (fed back) in the unique (possibly infinite) sequence of actions of $\mathsf{F}(s)$. Then $e' \models s \xrightarrow{\bar{a}}$. Then $e' \restriction_\ell \models s \xrightarrow{\bar{a}_\ell}$ and $\bar{a} \approx_\ell \bar{a}_\ell$ for some $\bar{a}_\ell$. Observe that by definition of F, for each $c_{\mathtt{D}}!v \in \mathbb{A}_{\mathtt{D}}$ in $\bar{a}$, $c_{\mathtt{D}}?v$ immediately follows. The same holds true for $\bar{a}_\ell$ by definition of $e' \restriction_\ell$. Thus $e \restriction_\ell \models \mathsf{B}(e' \restriction_\ell, s) \xrightarrow{\bar{a}_\ell}$ and $\bar{a} \approx_\ell \bar{a}_\ell$ for some $\bar{a}_\ell$. The remainder of this proof is obtained by comparing $\mathsf{B}(e' \restriction_\ell, s)$ to the ℓ-run of $\mathsf{SME}_{\mathtt{D}}(\rho, \sigma, s)$ (due to high-lead scheduling, no derivation of any step of $\mathsf{SME}_{\mathtt{D}}(\rho, \sigma, s)$ uses rule ($\mathtt{D}$-yes$_{\mathtt{d}}$), so the $c_{\mathtt{D}}$-inputs are the same as in $e' \restriction_\ell$) and by proceeding as in the proofs of Theorems 4.4 and 4.6, □

Proof of Theorem 6.2. Let e_1, e_2 such that $e_1 \approx_{\mathtt{L}} e_2$ be given. Then $e_1 \restriction_{\mathtt{L}} \approx_{\mathtt{L}} e_2 \restriction_{\mathtt{L}}$. Thus, for each $\bar{a}$ for which $e_1 \restriction_{\mathtt{L}} \models s \xrightarrow{\bar{a}}$, we have a $\bar{a}'$ for which $e_2 \restriction_{\mathtt{L}} \models s \xrightarrow{\bar{a}'}$ and $\bar{a} \approx_{\mathtt{L}} \bar{a}'$. Let σ and e be arbitrary. We show

1. for each $\bar{a}$ for which $e \models \mathsf{SME}_{\mathtt{T}}(\sigma, s) \xrightarrow{\bar{a}}$,
   we have a $\bar{a}'$ for which $e \restriction_{\mathtt{L}} \models s \xrightarrow{\bar{a}'}$ and $\bar{a} \approx_{\mathtt{L}} \bar{a}'$, and
2. for each $\bar{a}$ for which $e \restriction_{\mathtt{L}} \models s \xrightarrow{\bar{a}}$,
   we have a $\bar{a}'$ for which $e \models \mathsf{SME}_{\mathtt{T}}(\sigma, s) \xrightarrow{\bar{a}'}$ and $\bar{a} \approx_{\mathtt{L}} \bar{a}'$.

With the above, this gives, by transitivity of $\approx_{\mathtt{L}}$ that, for each $\bar{a}$ for which $e_1 \models \mathsf{SME}_{\mathtt{T}}(\sigma, s) \xrightarrow{\bar{a}}$, we have a $\bar{a}'$ for which $e_2 \models \mathsf{SME}_{\mathtt{T}}(\sigma, s) \xrightarrow{\bar{a}'}$ and $\bar{a} \approx_{\mathtt{L}} \bar{a}'$. The remainder of this proof establishes 1) and 2).

An L-observable action in an $\textsc{sme}_{\mathtt{T}}$-step is only derivable using (L-a) or (L-timeout). These derivations are only possible when the L-run is in a state where its next action is a L-observable. The L-observable action made by the $\mathtt{SME}_\mathtt{T}$-step is either the same as the one the L-run performed, or, in the case of an output on a $\mathtt{H}^\mathtt{L}$-channel, the outputted value can be replaced. Either way, the L-observable action made by the $\mathtt{SME}_\mathtt{T}$-step is $\approx_\mathtt{L}$-equivalent to the L-observable action made by the L-run. When an $\mathtt{SME}_\mathtt{T}$-step is derived using (L-wait), then a L-observable is forthcoming (derived using (L-a) or (L-timeout)). This follows from fairness of the scheduler; eventually, the H-run gets scheduled often enough to either reach an L-observable (in which case, the next time the scheduler produces L, $\mathtt{SME}_\mathtt{T}$-step is derivable using (L-a)), or timeout (in which case, the next time the scheduler produces L, $\mathtt{SME}_\mathtt{T}$-step is derivable using (L-timeout)).

Thus the sequence of L-observables performed by $\mathtt{SME}_\mathtt{T}(\sigma, s)$ is $\approx_\mathtt{L}$-equivalent to the sequence of L-observables performed by the L-run in $\mathtt{SME}_\mathtt{T}(\sigma, s)$. We establish that the sequence of L-observables performed by the L-run in $\mathtt{SME}_\mathtt{T}(\sigma, s)$ (which is run under environment e) is $\approx_\mathtt{L}$-equivalent to the sequence of L-observables performed by s under environment $e{\restriction}_\mathtt{L}$. The two runs match actions (and thus traverse the same sequence of states) until one performs an input. If one run reads on a $\mathtt{H}^\mathtt{H}$-channel, then so can the other, and both runs read d. This can be seen in the definition of $e{\restriction}_\mathtt{L}$ (for the s-under-$e{\restriction}_\mathtt{L}$-run), and by rule (old) (for the L-run in $\mathtt{SME}_\mathtt{T}(\sigma, s)$). If one run reads on a L-presence channel c, then so can the other run. Since both runs are input blocking, both runs will read a (possibly empty) list of blanks. If there are more c-inputs forthcoming in e, then by definition of $e{\restriction}_\mathtt{L}$, there will also be more c-inputs forthcoming in $e{\restriction}_\mathtt{L}$, and vice versa. So eventually, either both runs will read blanks infinitely, or both runs will read the same c-input, and enter the same state. (both runs read the same c-input in the case of $\kappa(c) = \mathtt{H}$ by (new-i) and definition of $e{\restriction}_\mathtt{L}$ (value becomes d)). □

Proof of Theorem 6.3. The sequence of actions performed by the run of s under e, and the sequence of actions performed by the H-run of $\mathtt{SME}_\mathtt{T}(\sigma, s)$ under e, are $=_{\star,\bullet}$-equivalent (as long as no attack is discovered). The two runs match actions (and thus traverse the same sequence of states) until one performs an input. If one run reads on a channel c, then so can the other run. Since both runs are input blocking, both runs will read a (possibly empty) list of blanks. So, either both runs will read blanks infinitely, or eventually, both runs will read the same c-input, and enter the same state.

We now show that the sequence of actions performed by the $\mathtt{SME}_\mathtt{T}(\sigma, s)$-run and the H-run in $\mathtt{SME}_\mathtt{T}(\sigma, s)$ respectively are $=_{\star,\bullet}$-equivalent (as long as no attack is discovered). While no attacks are discovered, (L-timeout) is not used in derivations of steps. In all other rules for deriving a-steps where $a \neq \bullet$, the rule requires that the H-run can do a (particularly, in the case of rule (L-a), the former **else** is never chosen while no attacks are discovered, and the H-run, will do a the next time the scheduler picks H). The only difference between the sequence of actions performed by the $\mathtt{SME}_\mathtt{T}(\sigma, s)$-run and the H-run in $\mathtt{SME}_\mathtt{T}(\sigma, s)$ respectively is in $\bullet$ and $\star$-read actions, which are insignificant when comparing for $=_{\star,\bullet}$-equivalence. □

Proof of Theorem 6.4. Consider each L-observable a in any trace $\bar{a}$ for which $e \models \mathtt{SME}_\mathtt{T}(\sigma, s) \to (\bar{a}, \sigma', S, \epsilon, \epsilon)$. The a-step was derived using rule (L-a), and the H-run in the state before the a-step can do an $a_\mathtt{H}$-step for some $a_\mathtt{H}$ for which $a =_\mathtt{L} a_\mathtt{H}$. The next time the H-run is scheduled, the H-run does the $a_\mathtt{H}$-step by (old). Since, for $S(\mathtt{L}) = (\bar{a}_\mathtt{L}, \_)$, $\bar{a} \approx_\mathtt{L} \bar{a}_\mathtt{L}$, we get by the above that the H-run can produce a trace $\bar{a}_\mathtt{H}$ for which $\bar{a}_\mathtt{L} \approx_\mathtt{L} \bar{a}_\mathtt{H}$. Now consider $\bar{a}$ and L-observable a for which $e \models \mathtt{SME}_\mathtt{T}(\sigma, s) \to (\bar{a}, \sigma', S, \epsilon, \epsilon) \to (\bar{a}.a, \_, S', \alpha, \epsilon)$ for some S and α. The a-step was derived using rule (L-a), and the H-run in $S(\mathtt{H})$ can do a L-observable $a_\mathtt{H}$-step, but cannot do one such that $a =_\mathtt{L} a_\mathtt{H}$. Since the H-run is deterministic, we get that the H-run cannot match $\bar{a}_\mathtt{L}$ where $S'(\mathtt{L}) = (\bar{a}_\mathtt{L}, \_)$. Thus, by definition of α, α is an $\approx$-attack on s. □