Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach

Vineeth Kashyap
UC Santa Barbara
vineeth@cs.ucsb.edu

Ben Wiedermann
Virginia Tech
benalan@cs.vt.edu

Ben Hardekopf
UC Santa Barbara
benh@cs.ucsb.edu

***Abstract*—Secure information flow guarantees the secrecy and integrity of data, preventing an attacker from learning secret information (secrecy) or injecting untrusted information (integrity). Covert channels can be used to subvert these security guarantees; for example, timing and termination channels can, either intentionally or inadvertently, violate these guarantees by modifying the timing or termination behavior of a program based on secret or untrusted data. Attacks using these covert channels have been published and are known to work in practice—as techniques to prevent non-covert channels are becoming increasingly practical, covert channels are likely to become even more attractive for attackers to exploit.**

The goal of this paper is to understand the subtleties of timing- and termination-sensitive noninterference, explore the space of possible strategies for enforcing noninterference guarantees, and formalize the exact guarantees that these strategies can enforce. As a result of this effort we create a novel strategy that provides stronger security guarantees than existing work, and we clarify claims in existing work about what guarantees can be made.

I. Introduction

Secure information flow guarantees the secrecy and integrity of data, preventing an attacker from learning secret information (secrecy) or injecting untrusted information (integrity). A commonly desired guarantee is *noninterference*: for secrecy, noninterference states that the behavior of publically observable events cannot be influenced by secret data; for integrity, it states that the behavior of trusted events cannot be influenced by untrusted data. The task of secure information flow is to identify *information channels*—mechanisms that are able to transmit information—and prohibit *leaks* (information flow along those channels that violates noninterference).

The most prevalent information channels addressed by existing work are *explicit* and *implicit* channels, corresponding to data and control dependencies in a program [32]. However, the most difficult types of information channels to control are *covert channels*; these channels are not intended by their nature to transmit information, but they can be subverted for this task. This paper focuses on *timing* and *termination* covert channels. These channels leak information (intentionally or inadvertently) by modifying the timing or termination behavior of a program based on secret or untrusted data[1]. Timing and termination channels have been used in practical attacks against secrecy (e.g., in web browsers [8], [16] and cryptographic applications [10], [15], [20], [25], [39]) and can also be a concern for integrity (e.g., preventing denial-of-service attacks by isolating the timing and termination behavior of a program from untrusted inputs). The goal of this paper is to understand the subtleties of timing- and termination-sensitive noninterference, explore the space of possible strategies for enforcing noninterference guarantees, and formalize the exact guarantees that these strategies can enforce. As a result of this effort we create a novel strategy that provides stronger guarantees than existing work, and we clarify claims in existing work about what guarantees can be made.

A. Noninterference Guarantees

While the basic concept of timing- and termination-sensitivity is clear, there are different formal definitions of these terms in the literature (e.g., Agat [2], Devriese and Piessens [13], Giacobazzi and Mastroeni [17], and Hedin and Sands [21]). These different formal definitions imply different security guarantees for noninterfering programs; the security implications of these different guarantees are not always clear. These guarantees are partially ordered by their strength, such that a stronger guarantee implies a weaker guarantee. Many of the definitions of timing- and termination-sensitive noninterference used to formally prove claims about security employ weaker guarantees [2], [7], [13], [17], [33], while published attacks against timing- and termination-sensitive noninterference can only be prevented using the stronger guarantees [8], [16].

In this paper we discuss different notions of timing- and termination-sensitivity and give simple examples that illustrate their different guarantees. We then formalize these notions and prove for a variety of enforcement strategies the specific guarantees that they can enforce. In the process, we clarify existing work by Devriese and Piessens [13], one of

[1]Both of these channels can leak an arbitrary number of bits [5], [36]. Timing channels have a higher bandwidth than termination channels (which need time exponential in the size of the leaked information), but termination channels can also be problematic, e.g., if a few bits contain dangerous information, or if an attacker can observe multiple runs of a program.

the most advanced works to date on enforcing timing- and termination-sensitivity, by (1) revealing a mistaken formal claim of security; (2) showing that an enforcement strategy proven to provide a weak guarantee of security actually provides a stronger guarantee; and (3) answering open questions in their paper about what guarantees can be provided.

B. Enforcing Noninterference

There are three basic approaches for enforcing timing- and termination-sensitive noninterference: *mitigation*, *restricted computation*, and *scheduling*. Section II briefly discusses the mitigation and restricted computation approaches; Section IV discusses the scheduling approach in detail. This paper focuses on the scheduling approach, which enforces complete noninterference (unlike mitigation) and allows arbitrary computation (unlike restricted computation).

The idea behind the scheduling approach is to partition a program (statically or dynamically) into a set of sub-programs, one for each element of the security lattice. These sub-programs are then executed concurrently using a scheduling strategy that guarantees the absence of timing and termination leaks.

In this paper we describe a space of possible scheduling strategies to enforce noninterference. We first informally describe a number of different strategies along with the guarantees they can provide. For the most interesting scheduling strategies we formalize the strategies and prove their security guarantees. Some of the described strategies are found in existing work; inspired by our descriptions of the strategies and their guarantees, we also describe a novel scheduling strategy that provides stronger guarantees than any strategy in existing work.

C. Our Contributions

The specific contributions of this paper are:

- We explore different notions of timing- and termination-sensitive noninterference in the literature by describing possible interpretations and how they differ.
- We explore a number of scheduler strategies for enforcing timing- and termination-sensitive noninterference and describe their respective security guarantees.
- We introduce a new scheduling strategy that provides stronger guarantees than any existing scheduler.
- For the most interesting schedulers (including our new scheduler) we give formal proofs of the security guarantees they can provide.
- We use our framework and formal proofs to clarify claims made in existing work [13]—contradicting some claims, strengthening other claims beyond what the prior work showed was possible, and giving results for some open questions asked in prior work.

The rest of the paper is laid out as follows: Section II discusses related work. Section III informally describes different notions of timing- and termination-sensitivity. Section IV describes the scheduling approach to enforcing noninterference. Section V informally describes a number of possible scheduling strategies and their guarantees. Section VI formalizes our notions of timing- and termination-sensitivity as well as the strongest scheduling strategies and proves their security guarantees. Section VII uses our results to clarify existing work and discusses other related issues. Section VIII concludes.

II. Related Work

Secure information flow has benefited from extensive study, beginning with the seminal work by Denning and Denning [11], [12], Goguen and Meseguer [18], and Volpano et al. [37]. Most existing work on enforcing secure information flow has focused on timing- and termination-insensitive security; we refer the reader to the comprehensive survey by Sabelfeld and Myers for general background on secure information flow research [32]. There has been extensive work on enforcing noninterference for concurrent programs; this prior work deals with *internal timing leaks*—violations of noninterference due to race conditions among concurrently executing threads [9], [29], [34], [38], [41]. While some of this prior work deals with thread scheduling (e.g., Russo and Sabelfeld [30]), their goal is to prevent only these internal leaks. In contrast, our work focuses on *external timing leaks*: leaks due to differences in execution time observable by an external attacker—these leaks are relevant for both sequential and concurrent programs. The three basic approaches for enforcing timing- and termination-sensitive noninterference are:

Mitigation: The mitigation approach accepts the fact that leaks will occur but attempts to either degrade the bandwidth of the leaks [22] or to bound the amount of information that can be leaked [6]. Mitigation does not actually enforce timing- and termination-sensitive noninterference, instead it attempts to make violations less harmful.

Restricted Computation: The restricted computation approach guarantees noninterference by constraining programs to disallow dangerous (potentially leaky) computation—this is most often accomplished via a type-system. Some systems disallow all loops and conditionals predicated on restricted (i.e., secret or untrusted) data, e.g., Volpano and Smith [38], or allow loops and conditionals on restricted data, but disallow any non-secret/trusted computation after any such loop or conditional, e.g., Boudol and Castellani [9] and Smith and Volpano [34]. Other systems enable conditionals predicated on restricted data by padding the branches of the conditional to take equal time, but still disallow loops on restricted data, e.g., Agat [2] and Barthe et al. [7]. This technique can be extended to include loops, but only by again restricting low computation after loops (Hedin and Sands [21]) or potentially making the computation unsound (Shroff and Smith [33]).

Scheduling: The scheduling approach for enforcing noninterference is our focus in this paper—unlike mitigation this approach *does* enforce noninterference, and unlike restricted computation it does *not* place any constraints on loops and conditionals. While there is related work in the literature, we are (to our knowledge) the first to recognize this strategy as a general approach for preventing timing and termination leaks. The most directly related work on using scheduling to eliminate timing and termination leaks is by Devriese and Piessens [13], called Secure Multi-Execution (SME). SME executes multiple instances of a sequential program concurrently, with one instance per security level, scheduling the threads to prevent timing and termination leaks. Devriese and Piessens formalize the SME model and prove its security properties, and they also evaluate a practical implementation of the technique for JavaScript. Our family of scheduling strategies encompasses and subsumes those discussed in the SME paper, and in Section VII we discuss the SME paper further. Other related research does not directly target timing- and termination-sensitive noninterference, but is relevant to the general scheduling strategy. The VAX security kernel eliminates timing leaks via cache memory by flushing the cache whenever a process is switched out for another process at a lower security level; Hu [23] proposes a process scheduler based on the security lattice designed to minimize the number of times the cache is flushed. Russo and Sabelfeld [30] consider a multi-threaded language and propose a thread scheduler designed to eliminate internal timing leaks.

III. Security Models and Noninterference

In this section we informally describe and discuss different interpretations of timing- and termination-sensitive noninterference. The information-flow policy of a program is defined using a lattice $(\mathcal{L}, \sqsubseteq)$ where $\mathcal{L}$ is a set of security classes and $\sqsubseteq$ is a partial order indicating relative trust or secrecy among those classes. The terms *high* and *low* indicate relative position in the lattice. With regards to secrecy higher means "more secret", and with regards to integrity higher means "less trusted"—the higher the data is in the lattice, the more its flow should be restricted. Figure 1 shows an example lattice that specifies a policy for compartmentalizing classified information. There are 4 security levels: an Unclassified level (`U`), a secret level for the Army (`S:army`), a secret level for the Navy (`S:navy`), and the highest level that has access to all information. This lattice specifies that Unclassified information can be seen by anyone, but that only someone with Army clearance can see Secret Army information (and similarly for the Navy). In particular, since the `S:army` and `S:navy` levels are non-comparable, Army personnel with Secret clearance cannot see the Navy's Secret information and vice-versa.

Noninterference states that values of variables at a given security level $\ell \in \mathcal{L}$ can only influence the values of variables at any security level that is greater than or equal to ℓ in the security lattice. Different definitions of the word *influence* induce different security models.

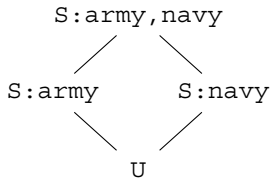

Figure 1. Example security lattice; `S` == Secret, `U` == Unclassified.

The most basic definition of influence considers only the values of the variables: a program is noninterfering if changing the values of variables at level $\ell' \not\sqsubseteq \ell$ cannot affect the values of variables at level ℓ either directly via data dependencies (called *explicit channels*) or indirectly via control dependencies (called *implicit channels*).

More sophisticated definitions of influence also account for *timing channels* (the ability of values at level ℓ' to affect *when* the values at level ℓ are computed) and *termination channels* (the ability of values at level ℓ' to affect *whether* the values at level ℓ are computed, via nontermination or abnormal termination). The remainder of this section discusses different security guarantees that can be made, assuming different properties of these timing and termination channels. Timing and termination models each have a dimension of *sensitivity*, which describes the assumptions that the model makes about the attacker's strength. The terms we use to describe these definitions are specific to this paper—there is no universally agreed-upon terminology.

A. Timing Models

A *timing-insensitive* model (TIME-I) discards all timing information from the program semantics and definition of noninterference, thereby assuming that an attacker has no means to time program execution. This model is common in existing works on secure information flow, and it is the weakest model with respect to timing.

A *timing-sensitive* model assumes that an attacker is able to time program execution, and it accounts for this fact in the program semantics and in the definition of noninterference. The semantics contains a function $\mathcal{T}$ that increments time at each program step. There are two varieties of timing-sensitivity depending on the definition of $\mathcal{T}$:

- A **weakly timing-sensitive** model (TIME-WS) models time abstractly in terms of the number of program execution steps. This model assumes that $\mathcal{T}$ increments time by a fixed, constant amount at each program step. This assumption means that the model abstracts away details of the execution environment that might influence program timing, including architectural features such as caches, pipelining, and branch predictors.

Program **(a)** Program **(b)**

Figure 2. Example programs for timing channels.

Among formal definitions of timing-sensitive noninterference, this is the most prevalent model [2], [7], [13], [17], [33].

- A **strongly timing-sensitive** model (TIME-SS) is the most aggressive security model with respect to timing; it models "real-world" time rather than the number of program execution steps and assumes that $\mathcal{T}$ increments time by some unknown, variable amount at each program step. TIME-SS assumes that an attacker is able to time program execution using "wall-clock" time and this assumption accounts for features in the program's execution environment, such as cache, pipelining, and branch predictors, that may influence program timing. We are aware of only one existing work that formalizes a strongly timing-sensitive model [21].

The programs in Figure 2 illustrate the differences among the three timing security models. For program **(a)** a timing-insensitive security model would not detect any information flow from variable x to the assignment statement, even though the value of x determines *when* the assignment statement executes. A weakly timing-sensitive model would detect this information flow because the program executes a different number of steps depending on which branch it takes. For program **(b)**, a weakly timing-sensitive model would not detect any information flow from variable x to the assignment statement because the program executes an equal number of steps regardless of which branch it takes. However, a strongly timing-sensitive model would detect a potential information flow because architectural features such as caches and branch predictors might affect the wall-clock execution time of the program depending on which branch is taken.

B. Termination Models

A *termination-insensitive* model (TERM-I) ignores the possibility of non-termination or of abnormal termination due to unchecked exceptions such as out-of-memory errors. This model only makes security guarantees under the assumption that a program always terminates normally. This model is common in existing work on secure information flow, and it is the weakest model with respect to termination.

A *termination-sensitive* model assumes that an attacker is able to observe a program's termination behavior, and it accounts for this fact in the program semantics and in the definition of noninterference. The ability to "observe termination behavior" is a bit odd and deserves further explanation. Assume the attacker is at some security level ℓ. We refer to any computation observable at level ℓ or lower as *low computation* and anything else as *high computation*. A termination-sensitive model assumes that, at any step in the program execution, an attacker can determine whether the program *will* or *will not* execute any more low computation in the future. A program is termination-sensitive noninterfering if values at security level $\ell' \not\sqsubseteq \ell$ cannot influence the answer to this question for an attacker at level ℓ.

A program can violate termination-sensitive noninterference in two ways; which of these ways is considered as part of the model defines two varieties of termination-sensitivity:

- A **weakly termination-sensitive** model (TERM-WS) only considers whether the high computation may diverge on some inputs and not others. If the high computation precedes some low computation in the program, then an attacker can infer information about the inputs based on whether that subsequent low computation will happen or not (which depends on whether the high computation diverges or not). Note that the attacker *cannot* observe the termination behavior of the high computation directly, only the *effect* of the high computation's termination behavior on the low computation. Among formal definitions of termination-sensitive noninterference, weak termination-sensitivity is the most prevalent model [2], [13], [31].

- A **strongly termination-sensitive** model (TERM-SS) considers the same factors as weak termination-sensitivity, but also considers whether the high computation may influence the termination behavior of the low computation via abnormal termination, e.g., by applying a partial function (such as division by zero) or exhausting a shared resource[2]. For example, consider a case where the high computation may or may not allocate almost all available memory based on some input. If the low computation then attempts to allocate memory, the low computation will or will not abnormally terminate from an out-of-memory error based on the behavior of the high computation, and an attacker observing the termination behavior of the low computation can infer information about the inputs. This is the strongest attacker model with respect to termination. We are not aware of any existing work that formalizes a strongly-sensitive termination model.

The programs in Figure 3 illustrate the differences among the three termination security models. For program **(a)** a termination-insensitive security model would not detect any information flow from variable x to the assignment statement, even though the value of x determines *whether*

[2]Although some researchers (e.g., [32]) describe resource exhaustion as a covert channel distinct from termination, we choose to include it in the termination channel because termination often is the by-product of resource exhaustion.

```
if x then while 1 do skip      if x then f x
    else skip ;                    else skip ;
y := 0                         y := 0
```

Program **(a)** Program **(b)**

Figure 3. Example programs for termination channels.

the assignment statement executes. A weakly termination-sensitive model would detect this information flow. For program **(b)**, a weakly termination-sensitive model would not detect any information flow from variable x to the assignment statement, despite the fact that the function application's allocation behavior may exhaust available memory and cause abnormal termination. A strongly termination-sensitive model would detect this potential flow.

C. Noninterference Guarantees

It is difficult to separate termination from timing and treat them orthogonally. In fact, termination-sensitivity is a special case of timing-sensitivity since influencing *whether* a low computation happens or not also influences *when* that computation happens (i.e., "sometime" versus "never"). For this reason we consider timing and termination sensitivity together for the remainder of the paper. From the discussion above, we distinguish three types of noninterference guarantees that vary in strength in terms of timing- and termination-sensitivity:

Guarantee	Timing/Termination Model
Insensitive	TIME-I, TERM-I
Weakly-sensitive	TIME-WS, TERM-WS
Strongly-sensitive	TIME-SS, TERM-SS

Insensitive noninterference guarantees noninterference under the TIME-I, TERM-I model, weakly-sensitive noninterference guarantees noninterference under the TIME-WS, TERM-WS model, and strongly-sensitive noninterference guarantees noninterference under the TIME-SS, TERM-SS model. In Section VI we formally define these three guarantees and prove the guarantees enforced by a number of different scheduling strategies.

IV. Enforcement: Scheduling Approach

As explained in Section II, existing work has developed differing approaches to enforcing the different timing- and termination-sensitive noninterference properties described in the previous section, with restricted computation as the most common approach. In this paper we focus on a different approach that we call the *scheduling approach*. This approach encompasses a family of strategies for preventing timing and termination leaks all based on the same basic idea: partitioning a program into sub-programs and concurrently executing those sub-programs in a manner that prevents leaks. Recently a related approach called Secure Multi-Execution [13] (SME) has been proposed; we see SME as a particular instance of this more general family of strategies that we explore in this paper.

The reasoning behind the scheduling approach is as follows. A program is *insensitively* noninterfering if the low computation cannot be either data or control dependent on high computation. Intuitively, this means that the low computation can be sliced out and executed independently of the high computation. Abadi et al. [1] formalize this connection between noninterference and dependency. The scheduling approach is based on the insight that (1) for an insensitively noninterfering program all low computation is independent of any high computation; (2) this program can be partitioned into multiple sub-programs that are independent of one another; and (3) these sub-programs can then be scheduled to execute so as to eliminate timing and termination leaks. The family of solutions representing the scheduling approach is parameterized by two characteristics:

- The *partitioning strategy*: how a solution partitions a program into sub-programs, one per security level.
- The *scheduling strategy*: how a solution schedules sub-program execution.

The partitioning strategy depends heavily on the particular features of the programming language being partitioned. We briefly discuss possible partitioning strategies in this section, however the main part of this paper focuses on the scheduling strategy, which enforces a particular security guarantee independently of the partitioning strategy used. We first outline some possible partitioning strategies. We then define a scheduler semantics that, given a set of sub-programs derived using some (arbitrary) partitioning strategy, models the concurrent execution of the individual sub-programs. This scheduling semantics is parameterized by the select function that chooses which sub-program to execute in each time-slice. In subsequent sections, we discuss a number of possible select functions and the security guarantees they provide (including the strategies described in the SME paper). We then formalize these strategies and prove our claims about their guarantees.

A. Partitioning Strategies

Secure Multi-Execution [13] demonstrates one practical partitioning strategy (implemented for JavaScript) that operates dynamically. If the security lattice has n elements, then n instances of the program will be created, each assigned a security level. A program at security level ℓ can read only from inputs with security levels $\ell' \sqsubseteq \ell$ and can write only to outputs with security level ℓ.

SME is not the only possible partitioning strategy. Other existing work explores methods for partitioning programs based on security levels, though besides SME none of these are intended specifically for enforcing timing- and

termination-sensitive noninterference using the scheduling approach. However, they do demonstrate the practicality of partitioning programs; these methods target real-world languages such as Java [19] and C [24], [35]. Abadi et al. [1] encode both a slicing calculus and a typed lambda calculus with secure information flow in their Dependency Core Calculus, showing the fundamental relation between program slicing and insensitive noninterference. Since then, there have been multiple works that have used program slicing to create sub-programs at different security levels. Hammer et al. [19] provide a program dependence graph based approach to information flow control in Java programs. Their technique can be used both to certify programs for insensitive noninterference as well as to obtain program slices. Amtoft and Banerjee [4] show how a Hoare-like logic based approach to specify information flow analysis can be used to obtain low slices that are free from high variables. Smith and Thober [35] use program slicing to guide programmers in refactoring high code into a separate component.

For the remainder of this paper we assume that an insensitively noninterferent program has been partitioned into a set of independent sub-programs, one per security level, using a strategy such as one of those described above.

B. Scheduling Semantics

In this section, we provide a formal definition for the semantics of schedulers. This definition will be used in the next two sections to informally describe a set of scheduling strategies and to formalize and prove the security of these scheduling strategies.

The scheduler executes each sub-program in its own thread (in the remainder of the paper, we conflate sub-programs with their respective threads and use the terms interchangeably). Informally, a scheduler switches among threads by continually choosing a candidate sub-program to execute for a specified amount of time before control is returned to the scheduler. A scheduler's security properties are determined by the way in which the scheduler chooses sub-programs to execute.

Figure 4 provides formal notation for the scheduling semantics and for the domains over which the semantics operates. We model sub-programs using an unspecified semantics—we leave the sub-program semantics unspecified because we wish to focus on the security properties of schedulers, rather than of the sub-programs themselves. However, these semantics are rich enough to describe the behavior of real-world languages and execution environments.

The sub-program semantics is given by a small-step operational relation $\rightsquigarrow$, which transforms a *sub-program configuration* consisting of a sub-program p, a *store* σ, and a *runtime state* ψ. We use record syntax to denote a configuration, and we use field syntax to denote elements of a configuration. A configuration's store $\kappa.\sigma$ models memory.

Semantic domains:

$p \in \mathit{Sub\text{-}program}$	Sub-programs
$\sigma : \mathit{Variable} \to \mathit{Value}$	Stores
$\psi \in \mathit{State} = \{\mathbf{R}, \mathbf{B}, \mathbf{T}\}$	Runtime States
$\delta \in \mathbb{R}$	Time
$\ell \in \mathcal{L}$	Security Labels

Sub-program semantics:

$$\kappa = \langle p : \mathit{Sub\text{-}program}, \sigma : \mathit{Store}, \psi : \mathit{State} \rangle$$
$$\rightsquigarrow \ \subseteq \kappa \times \kappa \qquad \text{Sub-program Semantics}$$

Scheduler semantics:

$$\mathsf{select} : (\overrightarrow{\mathit{Sub\text{-}program}} \times \overrightarrow{\mathit{State}}) \to (\mathit{Sub\text{-}program} \times \mathit{State} \times \mathcal{L})$$
$$K = \langle \overrightarrow{p} : \overrightarrow{\mathit{Sub\text{-}program}}, \sigma : \mathit{Store}, \overrightarrow{\psi} : \overrightarrow{\mathit{State}}, \delta \in \mathbb{R} \rangle$$
$$\twoheadrightarrow \ \subseteq K \times K \times \mathcal{L} \qquad \text{Scheduler Semantics}$$
$$\mathcal{T} : \mathit{Sub\text{-}program} \to \mathbb{R} \qquad \text{Time Semantics}$$

$$\frac{\exists\, \ell \in \mathcal{L} : \overrightarrow{\psi}[\ell] \neq \mathbf{Terminated} \quad p, \psi, \ell = \mathsf{select}(\overrightarrow{p}, \overrightarrow{\psi}) \quad \langle p, \sigma, \psi \rangle \rightsquigarrow \langle p', \sigma', \psi' \rangle \quad \overrightarrow{p}' = \overrightarrow{p}[\ell \mapsto p'] \quad \overrightarrow{\psi}' = \overrightarrow{\psi}[\ell \mapsto \psi'] \quad \delta' = \delta + \delta_s + \mathcal{T}(p)}{\langle \overrightarrow{p}, \sigma, \overrightarrow{\psi}, \delta \rangle \twoheadrightarrow_\ell \langle \overrightarrow{p}', \sigma', \overrightarrow{\psi}', \delta' \rangle}$$

Figure 4. Sub-program and scheduler semantics. **R** = **Ready**, **B** = **Blocked**, **T** = **Terminated**.

A configuration's runtime state $\kappa.\psi$ indicates whether the sub-program is **Ready**, **Blocked**, or **Terminated**. The semantics handles I/O based on the method described by Devriese and Piessens [13]: a sub-program only blocks on input if (1) it tries to read a value from an input at a lower security level ℓ', and (2) the ℓ' sub-program has not yet read that value from the input. The sub-program's state changes back from **Blocked** to **Ready** once the ℓ' sub-program has read from the low input. A sub-program may also block deterministically based on its own computation (e.g., a sleep system call). In this case, the sub-program changes back from **Blocked** to **Ready** after a deterministic amount of time.

A sub-program can make progress only when it is in the ready state. Given an initial store, a sub-program terminates when the reflexive, transitive closure of the semantic relation (denoted $\rightsquigarrow^*$) yields a configuration whose state is **Terminated**.

The scheduler orchestrates the sub-programs' computations. Its semantics are given by the relation $\twoheadrightarrow$, which transforms *scheduler configurations*. Each scheduler configuration is a four-tuple that consists of a *sub-program vector* $\overrightarrow{p}$, a store σ, a *state vector* $\overrightarrow{\psi}$, and a *time value* δ.

The sub-program vector $\overrightarrow{p}$ is a list of sub-programs. The vector contains a sub-program for each label ℓ in the security

lattice $\mathcal{L}$. A vector is indexed by a security label, so that the expression $\overrightarrow{p}[\ell]$ refers to the sub-program for security label ℓ. The state vector $\overrightarrow{\psi}$ holds the sub-programs' corresponding state values.

The scheduler semantics proceeds in steps as defined at the bottom of Figure 4. At each step, the scheduler checks that there exists at least one sub-program that has not terminated. If so, then the scheduler uses its select function to choose which sub-program will execute next. For the sake of the discussion in this section, we assume that select non-deterministically chooses an available sub-program. In subsequent sections, we describe more concrete strategies.

At each step, the scheduler records the level ℓ of the selected sub-program, as denoted by $\twoheadrightarrow_\ell$. For convenience, we sometimes omit a step's level. We write $\twoheadrightarrow_S$ to mean a scheduling step that selects one level from a set S of possible levels.

Having selected a sub-program p and its corresponding state ψ, the scheduler executes the sub-program for one step using the sub-program semantics given by $\rightsquigarrow$. This results in a transformed sub-program and potentially modifies the sub-program's state and the global store. The scheduler updates the sub-program and state vectors to record these changes. The expression $\overrightarrow{p}[\ell \mapsto p']$ updates the the sub-program vector to contain the reduced sub-program for level ℓ; the expression $\overrightarrow{\psi}[\ell \mapsto \psi']$ updates the the state vector to contain the modified state for level ℓ.

The scheduler also updates the time value δ at each step, according to a parameterized definition of time, $\mathcal{T}$. When a sub-program p takes a step, the scheduler semantics increases time by $\mathcal{T}(p)$ (the time required to execute p for one step) and by δ_s (which corresponds to the time required for the scheduler to select a sub-program and update the vectors).

—Timing and Termination Models—

The scheduler semantics implicitly embeds parameterized timing and termination models. The function $\mathcal{T}$ parameterizes the scheduler semantics by a given timing model. Different definitions for $\mathcal{T}$ correspond to different models of time. Our time parameterization follows in the style of Hedin and Sands [21], who describe how to model various notions of time. A timing-insensitive model can be defined by a function $\mathcal{T}(p) = 0$, i.e., one that ignores time. A weakly timing-sensitive model can be defined by a function $\mathcal{T}(p) = 1$, i.e., one that models every command as taking the same amount of time. A strongly timing-sensitive model can be defined by a deterministic, history-based function $\mathcal{T}$ which provides a value for time at each step and which accounts for all features of the execution environment that affect timing (e.g., caches, branch predictors, etc).

The scheduler semantics also encompasses a range of termination models. As described in Section III-B, an insensitive termination model allows for security guarantees about terminating programs only, i.e., those whose final state is **Terminated**. Under a strongly-sensitive termination model, a low sub-program can detect whether it abnormally terminates due to the behavior of a high sub-program. Under a weakly-sensitive termination model, it cannot. The semantics of the store distinguishes these two cases. If a low sub-program is insensitively noninterferent, then the only way for a high sub-program to cause the low sub-program's abnormal termination is via memory exhaustion. If the store described by the sub-program's semantics is exhaustible (i.e., of finite size), then the semantics describes a strongly-termination sensitive model. If the store is of infinite size, then the semantics describes a weakly-sensitive termination model.

A scheduler's security properties depend on the timing and termination models and on the way in which it selects sub-programs to execute. In the subsequent section, we informally describe possible selection strategies and their security properties under certain timing and termination models. In Section VI, we formally describe the strongest of these strategies and prove their security.

V. Scheduling Strategies

In this section, we explore the space of possible scheduling strategies created by the scheduler semantics in the previous section. We informally examine several strategies, discussing their characteristics and the security guarantees they can provide (either *insensitive*, *weakly-sensitive*, or *strongly-sensitive* noninterference). Figure 5 summarizes these strategies. Some of these strategies correspond to those used in existing work, but the Lattice-Based strategy is novel to this paper and results in stronger security guarantees than any other scheduler. The three strongest strategies—Sequential-2, Multiplex-2, and Lattice-Based—are formalized and their security guarantees proven in Section VI.

Each scheduling strategy is defined by a *filter* function. This function, at each step, examines the set of all threads, whether **Ready**, **Blocked**, or **Terminated**, and returns a subset of threads for the select function to choose from (selecting a **Blocked** or **Terminated** thread to run corresponds to executing a noop instruction). We assume that the select function chooses threads from the resulting pool in a round-robin fashion to guarantee progress for each thread in the pool[3]. In the following discussion we conflate threads with their security levels, e.g., thread `A` is lower than thread `B` if the code in thread `A` is at a lower security level than the code in thread `B`.

A. Scheduling Strategy *Sequential-1*

The sequential strategies execute threads in increasing order of their security level. At each step, Sequential-1

[3]This assumption gives the attacker greater power than with a nondeterministic scheduler because it gives the attacker stronger guarantees about how the scheduler will behave, making it easier to infer information.

Scheduling Strategy	Filter Function	Security Guarantee
Sequential-1†	Lowest **R** thread	**Weakly-sensitive** between comparable levels. **Insensitive** between noncomparable levels.
Sequential-2†	Lowest **R/B** thread	**Strongly-sensitive** between comparable levels. **Insensitive** between noncomparable levels.
Multiplex-1	All **R** threads (or all **R/B** threads)	**Insensitive** between all levels.
Multiplex-2	All **R/B/T** threads	**Weakly-sensitive** between all levels
Lattice-Based†	Fixed number of threads that includes all **R/B** threads for which all lower threads are **T**	**Strongly-sensitive** between comparable levels. **Weakly-sensitive** between noncomparable levels.

Figure 5. Scheduling strategies and their security guarantees (**R** = **Ready**, **B** = **Blocked**, **T** = **Terminated**). The filter function defines a scheduling strategy by returning a pool of threads for the select function to choose from (choosing a **Blocked** or **Terminated** thread corresponds to executing a noop instruction). Strategies marked with † can starve higher-level threads if a lower-level thread diverges.

selects the lowest-level **Ready** thread. Given a partial order, the 'lowest-level' thread may not always be well-defined; to ensure this choice is always well-defined we extend the partial order to a total order (meaning we refine the partial order to a total order that respects the initial ordering relation). Any finite partial order can be extended in this manner.

The immediate consequence of extending the partial order to a total order is that no guarantees can be made between noncomparable threads. Consider the case of the security lattice in Figure 1, with the noncomparable levels `S:army` and `S:navy`. In the new total order one of these levels must necessarily precede the other. If `S:army`'s execution precedes `S:navy`'s then the `S:army` thread's timing and termination behavior can leak information to the `S:navy` thread; similarly if `S:navy` precedes `S:army`.

Among comparable threads, however, Sequential-1 gives a stronger guarantee: *weakly-sensitive* noninterference. The strategy guarantees that the scheduler never executes a higher-level thread when a lower-level thread is **Ready**. Thus, the timing of the lower-level thread, measured using number of execution steps, is independent of the behavior of higher-level threads (including their termination behavior). However, this strategy does allow higher-level threads to execute when no lower-level threads are **Ready**. Higher-level threads can therefore affect the execution environment (e.g., cache, branch predictors, memory, etc), which prevents Sequential-1 from guaranteeing strongly-sensitive noninterference. For example, a higher-level thread may or may not evict a certain memory location from the cache based on secret data. A lower-level thread can observe which action was taken based on the length of time needed for it to access the same memory location and thereby infer information about the higher thread.

An additional difficulty with this strategy is that higher-level threads may starve if a lower-level thread diverges. Consider the following program:

```
while (true) do
   low  := low+1 ; high := high+2
```

Once divided into a sub-program per level, the low thread (updating `low` in an infinite loop) will never terminate, and hence the high thread (updating `high` in an infinite loop) will never get a chance to execute.

*B. Scheduling Strategy **Sequential-2***

The Sequential-2 strategy strengthens Sequential-1 by choosing the lowest level **Ready** *or* **Blocked** thread. The effect is to always execute the lowest non-terminated thread to completion before executing any other thread. Because no higher thread can affect the execution environment in any way while a lower thread executes, this strategy provides *strongly-sensitive* noninterference between comparable threads. However, this strategy, like Sequential-1, can offer only *insensitive* noninterference between noncomparable threads and also may starve higher-level threads if a lower-level thread diverges.

*C. Scheduling Strategy **Multiplex-1***

The multiplexing strategies use a filter function that returns multiple threads as candidates for the select function, as opposed to the sequential strategies' filter function that returns a singleton set. The Multiplex-1 filter function returns all **Ready** threads for the select function to choose from. In effect, the Multiplex-1 scheduler time-multiplexes among all of the **Ready** threads regardless of their security level, executing each thread one step at a time.

The strongest security guarantee that the Multiplex-1 strategy can provide is *insensitive* noninterference. As an example, consider the following program running in a high thread concurrently with some low thread:

```
if (high) then skip else while (true) do skip
```

If `high` is true, then the high thread executes one `skip` instruction and terminates. After the high thread terminates

the low thread continues to execute all by itself. However, if `high` is false, then the high thread executes forever. In this case, the low thread must continually alternate its execution with the high thread, thus leaking information about `high` to a low observer[4]. An advantage of Multiplex-1 over the sequential strategies is that all threads are guaranteed progress, so no high thread will starve regardless of the lower threads' behaviors.

D. Scheduling Strategy *Multiplex-2*

The Multiplex-2 strategy strengthens Multiplex-1 by picking from among *all* threads—**Ready**, **Blocked**, and **Terminated**. In effect, Multiplex-2 time-multiplexes among all threads regardless of status or security level. The key difference from Multiplex-1 is that the number of threads being multiplexed remains constant (i.e., all of them). Thus, Multiplex-2 prevents the termination behavior of one thread from affecting the timing of any other thread and therefore does not leak information in the way Multiplex-1 does.

However, Multiplex-2 can still give only a *weakly-sensitive* noninterference guarantee. Because high and low threads are time-multiplexed, high threads are able to affect the execution environment and leak information in the same manner as that described for Sequential-1.

E. Scheduling Strategy *Lattice-Based*

In this section, we describe a novel strategy inspired by the strengths and weaknesses of the Sequential and Multiplex strategies. The essential idea of this strategy is to schedule based on the the partial order of the security lattice itself. However, doing so naïvely does not quite provide the guarantees we need. We first describe two failed attempts at defining the Lattice-Based scheduling strategy, then we describe our correct version of the strategy.

*1) First Attempt at Scheduling Strategy **Lattice-Based**:* In our first attempt at defining the Lattice-Based strategy, we combine the behaviors of Sequential-2 and Multiplex-2 by multiplexing all threads at one tier of the security lattice to completion before executing any threads from a higher tier. To be more precise, the scheduler assigns to thread ℓ a priority that is equal to the length of the longest chain in the lattice from ℓ to $\bot$. The filter function then returns all **Ready/Blocked/Terminated** threads with priority x such that all threads whose priority is lower than x are **Terminated**. For example, given the lattice in Figure 6a: first the $\bot$ thread executes to completion, then the `D` and `E` threads execute concurrently until they both are complete, then `A`, `B`, and `C` execute concurrently until they are all complete, then finally $\top$ executes.

[4]A thread could also leak information via its blocking behavior in the same fashion; we can modify the filter function to return all **Ready/Blocked** threads to prevent leaking information via blocking, but the resulting scheduling strategy still fails the example above and has the same insensitive noninterference guarantee as Multiplex-1.

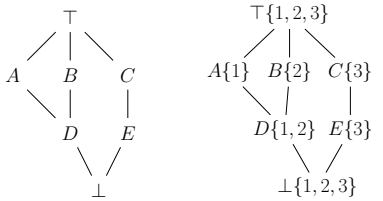

Figure 6. (a) an example security lattice and (b) the example lattice annotated with slots assigned by the donation strategy discussed in Section V-E3.

The intent of this attempt at defining the Lattice-Based strategy is to combine the security guarantees of Sequential-2 and Multiplex-2; however, it is only partially successful. The strongest guarantee this strategy can give is *strongly-sensitive* noninterference among comparable threads and *insensitive* noninterference among noncomparable threads. Consider again the scheduling of the lattice in Figure 6a. As with Sequential-2, no thread executes until all lower threads have completed[5]. But suppose that while `D` and `E` are executing concurrently, `E` terminates before `D` does. Then the `C` thread cannot begin executing until `D` completes, thus `D`'s timing and termination behaviors leak to `C`.

*2) Second Attempt at Scheduling Strategy **Lattice-Based**:* The previous definition of the Lattice-Based strategy leaked information because it could force one thread to wait for another, noncomparable thread to terminate. To prevent this problem, we refine the filter function to return all **Ready/Blocked** threads x such that all threads lower than x are **Terminated**. For the lattice in Figure 6a, `C` would have to wait for `E` to complete, but would not have to wait for `D`.

This strategy still guarantees *strongly-sensitive* noninterference between comparable security levels, but does not improve on the *insensitive* noninterference between noncomparable levels. Suppose, using the same lattice, that $\bot$ has completed and `D` and `E` are concurrently executing. Suppose that `E` then completes, hence `C` immediately starts executing concurrently with `D`, sharing the execution time half and half. Then `D` completes and both `A` and `B` begin executing, thus splitting the execution time with `C` into thirds (i.e., `C` is now executing on every third instruction instead of every other instruction). From this reduction in execution time, `C` can infer timing and termination behavior of `D`.

*3) Correct Version of Scheduling Strategy **Lattice-Based**:* The previous strategy failed because the number of threads concurrently executing could vary, thus leaking information to noncomparable threads. The correct version of Lattice-Based strategy is similar to the previous attempt, but avoids its deficiencies by ensuring that a constant number of threads are concurrently executing at all times, thus re-capturing the benefits of Multiplex-2. Lattice-Based has the strongest

[5]Also like Sequential-2, high threads may starve if lower threads diverge.

security guarantees of all the schedulers, enforcing *strongly-sensitive* noninterference between comparable threads and *weakly-sensitive* noninterference between noncomparable threads. However, it still suffers from the weakness of the *Sequential* strategies by potentially starving higher threads if lower threads diverge.

The Lattice-Based filter function ensures that the scheduler always multiplexes a constant number of threads. To do so, the scheduler creates k *slots* and pre-assigns each thread to one or more slots. The scheduler's filter function then selects a particular slot to execute at each step (as opposed to a particular thread) in a round-robin fashion. If a step's selected slot corresponds to a **Blocked** or **Terminated** thread, then the scheduler executes a noop for that step, effectively padding the execution.

The number of slots k must be large enough to accommodate the largest possible set of qualifying threads (otherwise some qualifying thread would not get to run and this would leak information). Making k larger than necessary does not impact the security guarantee, but does affect performance. Computing a value for k reduces to the MAX CLIQUE graph problem. Given a security lattice $(\mathcal{L}, \sqsubseteq)$, the scheduler constructs an undirected graph $G = (V, E)$, where V contains the elements of the security lattice and E consists of edges between any two non-comparable security levels. Let MC be the maximum clique in G, i.e., the largest set of elements in $\mathcal{L}$ that are non-comparable to one another. Then $k = |MC|$. For the lattice in Figure 6a, $k = 3$. The MAX CLIQUE problem is NP-complete. For small security lattices (fewer than 1,000 elements) the scheduler can employ a fast exact algorithm [27], for large security lattices it can employ an approximation algorithm [28].

*Optimizing Performance of **Lattice-Based**:* Performance is a potential issue with the Lattice-Based scheduling strategy—if the number of qualifying threads is less than k, then the scheduler wastes time executing noops instead of doing useful work. For example, for the lattice in Figure 6a, the $\top$ thread executes when all other threads have terminated. Since $k = 3$, the scheduler will multiplex the $\top$ thread with two dummy threads running noops. Intuitively, however, it is safe for the $\top$ thread to execute in all three scheduling slots. This optimization allows the $\top$ thread to infer information about the timing and termination of A, B, and C, but since they are all lower in the lattice than $\top$, the optimization preserves noninterference.

We use this intuition to develop an optimization for the Lattice-Based strategy called *Slot Donation* that allows lower-level threads to "donate" their scheduling slots to higher-level threads. This optimization would allow A, B, and C to donate their slots to $\top$, so that $\top$ can use all three scheduling slots when it executes. This optimization is safe (i.e., it does not violate noninterference) because lower-level threads are only allowed to donate slots to comparable, higher-level threads.

The scheduler statically computes a *donation strategy*, which assigns to each thread the set of scheduling slots in which it can safely execute. When the filter function returns a thread that can go in multiple scheduling slots, the scheduler executes the thread in each slot given by the donation strategy (instead of filling the extra slots with noops). The donation strategy must guarantee that no threads that may execute concurrently will be assigned the same slot in which to execute.

Computing a donation strategy reduces to a particular type of network flow problem: a circulation problem with lower bounds. The scheduler constructs a flow graph $F = (V, E)$ whose nodes are computed from the elements of the security lattice $\mathcal{L}$ as follows. For every $l \in \mathcal{L}$ the scheduler creates two nodes l and l' and creates an edge $l \to l'$. The scheduler also creates an edge $l'_1 \to l_2$ for all elements $l_1, l_2 \in \mathcal{L}$ such that $l_1 \sqsubset l_2$ and $\nexists l_3 \in \mathcal{L} : l_1 \sqsubset l_3 \sqsubset l_2$. The $\top'$ node is the *sink* node, and the $\bot$ node is the *source* node. Figure 7 gives the flow-graph corresponding to the lattice in Figure 6a.

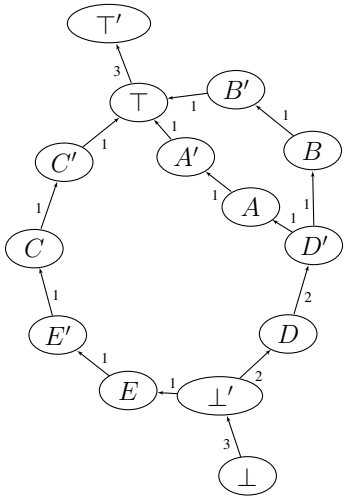

Figure 7. Flow-graph corresponding to Figure 6a, edges are annotated with flow values that satisfy the circulation problem described in Section V-E3.

A circulation problem gives each node a *demand* D and each edge a *capacity* C and *lower bound* L. The solution assigns each edge a flow value that simultaneously satisfies the demands of all nodes and the lower bound and capacity constraints of all the edges. The scheduler assigns demands, capacities, and lower bounds as follows:

1) Set $D(\bot) = -k$ and $D(\top') = k$. For each $v \in V - \{\top', \bot\}$, set $D(v) = 0$. [These demands set up the flow problem such that the source node $\bot$ can supply k slots while the sink node $\top$ must consume k slots. The remaining nodes act as transit nodes].
2) For each $l \in MC$, set $L(l \to l') = 1$ and $C(l \to l') = 1$. [These lower bounds and capacities state that each $l \in MC$ should be assigned exactly one slot].
3) For each $l \notin MC$, set $L(l \to l') = 1$ and $C(l \to$

$l') = k$. [These lower bounds and capacities state that every other $l' \notin MC$ can receive between 1 and k slots. Because every l' must either reach a node in *MC* or be reached by such a node, they are guaranteed to receive at least one slot].

4) For every edge $e \in E$ without an assigned lower bound or capacity, set $L(e) = 0$ and $C(e) = k$. [These lower bounds and capacities state that flow values in edges between comparable levels are not important].

The scheduler can solve the circulation problem using any of the well-known solution strategies [3]. Given a solution, the flow value assigned to the edge $l \rightarrow l'$ determines the number of slots in which the thread at level l may execute. The scheduler assigns each thread a set of slots by initializing the source node to the set of slots $\{1..k\}$, then forwarding them through the graph based on the number of slots in which a node may execute. For example, Figure 7 depicts a flow graph corresponding to the security lattice in Figure 6a. The edges of the graph are annotated with flow values that satisfy the circulation problem setup as described above. Figure 6b gives a possible slot assignment for each node based on the computed donation strategy.

VI. Formal Security Guarantees

We now present a formal framework that encapsulates the scheduling strategies described in the previous section. We first provide formal definitions for various kinds of noninterference. We then formally define strategies for three schedulers: Sequential-2, Multiplex-2, and Lattice-Based.

A. Noninterference

Noninterference is defined with respect to a security lattice $\mathcal{L}$ and a label $\ell \in \mathcal{L}$ that partitions the security lattice into low labels and high labels. Given ℓ, the low labels are $L_\ell = \{\ell' \in \mathcal{L} \mid \ell' \sqsubseteq \ell\}$. The high labels are then $H_\ell = \mathcal{L} - L_\ell$ (which includes all labels noncomparable to ℓ). We omit the subscripts for L and H when the specific label is unimportant. Variables have specified security levels and can be high or low, written $x : L$ or $x : H$. Sub-programs can be labeled as high or low based on whether they modify any low variables ($p : L$) or not ($p : H$).

—Low Equivalence Definitions—

We give a series of definitions that formalize our intuitive notion of security. We begin with the definition of *low equivalence* for the semantic domains given in Figure 4. Low equivalence captures the notion that an attacker can only observe things at low security levels, and hence differences at high security levels are irrelevant.

Definition 1 (Low Equivalence of Sub-Programs)**.** *Two sub-programs p_1 and p_2 are low-equivalent ($p_1 \sim_L p_2$) if both sub-programs are syntactically equivalent (denoted $p_1 = p_2$) or if both sub-programs are high:*

$$p_1 \sim_L p_2 \Leftrightarrow (p_1 = p_2) \vee (p_1 : H \wedge p_2 : H)$$

Definition 2 (Low Equivalence of Stores)**.** *Two stores σ_1 and σ_2 are low-equivalent ($\sigma_1 \sim_L \sigma_2$) if their low values agree:*

$$\sigma_1 \sim_L \sigma_2 \Leftrightarrow \forall x \in dom(\sigma_i)\ .\ x{:}L \Rightarrow \sigma_1[x] = \sigma_2[x]$$

Proving that a semantic relation exhibits insensitive noninterference relies on proving the following lemma, which states that high commands cannot modify low values.

Lemma 1 (Confinement)**.** *For every security level ℓ, high sub-program $p : H$, store σ, and state ψ:*

$$\langle p, \sigma, \psi\rangle \rightsquigarrow \langle p', \sigma', \psi'\rangle \textbf{ \textit{implies} } \sigma \sim_L \sigma'.$$

In our subsequent proofs for the scheduler semantics, we assume that the Confinement Lemma holds for any given sub-program semantics. We also assume that the sub-program semantics is deterministic and that low sub-programs never block due to I/O, as described in Section IV-B.

We now lift these definitions of low equivalence to operate on the semantic domains of the scheduler.

Definition 3 (Low Equivalence of Sub-Program Vectors)**.** *Two sub-program vectors $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$ are low-equivalent ($\overrightarrow{p_1} \sim_L \overrightarrow{p_2}$) if their corresponding sub-programs are low-equivalent:*

$$\overrightarrow{p_1} \sim_L \overrightarrow{p_2} \Leftrightarrow \forall \ell' \in \mathcal{L} : \overrightarrow{p_1}[\ell'] \sim_L \overrightarrow{p_2}[\ell']$$

Definition 4 (Low Equivalence of State Vectors)**.** *Two state vectors $\overrightarrow{\psi_1}$ and $\overrightarrow{\psi_2}$ are low-equivalent ($\overrightarrow{\psi_1} \sim_L \overrightarrow{\psi_2}$) if their corresponding low states are the same:*

$$\overrightarrow{\psi_1} \sim_L \overrightarrow{\psi_2} \Leftrightarrow \forall \ell' \in L : \overrightarrow{\psi_1}[\ell'] = \overrightarrow{\psi_2}[\ell']$$

Given the Confinement Lemma for the sub-program semantics (Lemma 1), we can establish a corresponding corollary for the scheduler semantics.

Corollary 1 (Scheduler Confinement)**.** *For every security level ℓ, every sub-program vector $\overrightarrow{p}$, every store σ, every state vector $\overrightarrow{\psi}$, and every time value δ:*

$$\langle \overrightarrow{p}, \sigma, \overrightarrow{\psi}, \delta\rangle \twoheadrightarrow^*_H \langle \overrightarrow{p'}, \sigma', \overrightarrow{\psi'}, \delta'\rangle$$

implies $\overrightarrow{p} \sim_L \overrightarrow{p'}$ ***and*** $\sigma \sim_L \sigma'$ ***and*** $\overrightarrow{\psi} \sim_L \overrightarrow{\psi'}$.

The scheduler semantics operates over configurations, for which we define a notion of low-equivalence:

Definition 5 (Low Equivalence of Scheduler Configurations)**.** *Two scheduler configurations K_1 and K_2 are low-equivalent ($K_1 \sim_L K_2$) if their sub-program vectors, stores, and state vectors are low-equivalent and if their times are equal:*

$$\begin{aligned} K_1 \sim_L K_2 \Leftrightarrow \\ &(K_1.\overrightarrow{p} \sim_L K_2.\overrightarrow{p}) \wedge (K_1.\sigma \sim_L K_2.\sigma) \wedge \\ &(K_1.\overrightarrow{\psi} \sim_L K_2.\overrightarrow{\psi}) \wedge (K_1.\delta = K_2.\delta) \end{aligned}$$

Given a deterministic sub-program semantics we can establish the following lemma for the scheduler semantics, which says that, under a weakly-sensitive termination model, the scheduler preserves low-equivalence when it takes a low step.

Lemma 2 (Weak Low-step Preservation)**.** *Under a weakly-sensitive termination model, for every security level ℓ and every pair of low-equivalent configurations $K_1 \sim_L K_2$:*

$K_1 \twoheadrightarrow_\ell \langle \overrightarrow{p_1'}, \sigma_1', \overrightarrow{\psi_1'}, \delta_1' \rangle$ ***and*** $K_2 \twoheadrightarrow_\ell \langle \overrightarrow{p_2'}, \sigma_2', \overrightarrow{\psi_2'}, \delta_2' \rangle$

implies $\overrightarrow{p_1'} \sim_L \overrightarrow{p_2'}$ ***and*** $\sigma_1' \sim_L \sigma_2'$ ***and*** $\overrightarrow{\psi_1'} \sim_L \overrightarrow{\psi_2'}$.

Proof: By Definitions 3 and 4, $\overrightarrow{p_1}[\ell] = \overrightarrow{p_2}[\ell]$ and $\overrightarrow{\psi_1} \sim_L \overrightarrow{\psi_2}$. Therefore, under a weakly-sensitive termination model, the scheduler makes the same decision under both configurations: either the scheduler executes $\overrightarrow{p_i}[\ell]$ for one step or the scheduler executes a noop. If the scheduler executes a noop, then $\overrightarrow{p_1'} = \overrightarrow{p_2'}$, and $\sigma_1' = \sigma_2'$, and $\overrightarrow{\psi_1'} \sim_L \overrightarrow{\psi_2'}$. The same result is achieved if the scheduler executes $\overrightarrow{p_i}[\ell]$, where $i \in \{1, 2\}$, because $K_1.\sigma \sim_L K_2.\sigma$ and the sub-program semantics are deterministic. ■

Under a strongly-sensitive termination model, the scheduler does not preserve low-equivalence. Recall from Section IV-B that, under a strongly-sensitive termination model, the store is of finite size. Consider the case when σ_1 and σ_2 are low-equivalent and σ_1 has free memory but σ_2 does not. If p_i allocates memory, then K_1 executes normally but K_2 terminates abnormally due to resource exhaustion. Assuming $\psi_1' \neq \mathbf{Terminated}$, we have $\psi_1' \neq \psi_2'$, so the scheduler semantics do not preserve low-equivalence.

—Noninterference Definitions—

We now formally define different notions of noninterference. A terminating sub-program exhibits *insensitive noninterference* if the sub-program's execution preserves low equivalence.

Definition 6 (Insensitive Noninterference)**.** *A semantic relation $\rightsquigarrow$ exhibits* insensitive noninterference *if, for every security level ℓ and every pair of low-equivalent configurations $\kappa_1 \sim_L \kappa_2$:*

$\kappa_1 \rightsquigarrow^* \langle p_1', \sigma_1', \mathbf{Terminated} \rangle$ ***and***

$\kappa_2 \rightsquigarrow^* \langle p_2', \sigma_2', \mathbf{Terminated} \rangle$

implies $p_1' \sim_L p_2'$ ***and*** $\sigma_1' \sim_L \sigma_2'$.

A sub-program exhibits *sensitive noninterference* if execution time cannot be used to differentiate among low-equivalent values.

Definition 7 (Sensitive Noninterference)**.** *A semantic relation $\twoheadrightarrow$ exhibits* sensitive noninterference *if, for every security level ℓ and every pair of low-equivalent configurations $K_1 \sim_L K_2$:*

$K_1 \twoheadrightarrow^* \langle \overrightarrow{p_1'}, \sigma_1', \overrightarrow{\psi_1'}, \delta' \rangle$ ***and*** $K_2 \twoheadrightarrow^* \langle \overrightarrow{p_2'}, \sigma_2', \overrightarrow{\psi_2'}, \delta' \rangle$

implies $\overrightarrow{p_1'} \sim_L \overrightarrow{p_2'}$ ***and*** $\sigma_1' \sim_L \sigma_2'$ ***and*** $\overrightarrow{\psi_1'} \sim_L \overrightarrow{\psi_2'}$.

This definition says that a semantic relation is sensitively noninterferent if, after executing two low-equivalent configurations for an arbitrary amount of time, low-equivalence is preserved. If the definition holds under a weak timing and termination model then the semantic relation exhibits *weakly-sensitive noninterference*. If the definition holds under a strong timing and termination model then the semantic relation exhibits *strongly-sensitive noninterference*.

Section IV claims that some scheduling strategies are "strongly-sensitive among comparable levels," a notion we now formally define. This definition states that if an ℓ_2-level sub-program executes, then time cannot be used to differentiate values at levels strictly lower than ℓ_2.

Definition 8 (Strong Sensitivity Among Comparable Levels)**.** *A semantic relation $\twoheadrightarrow$ is* strongly-sensitive among comparable levels *if, under a strong timing and termination model, for every pair of comparable security labels $\ell_1 \sqsubset \ell_2$, if $L = L_{\ell_1}$, then for every pair of low-equivalent configurations $K_1 \sim_L K_2$:*

$K_1 \twoheadrightarrow_{\ell_2} K_1' \twoheadrightarrow^* \langle \overrightarrow{p_1''}, \sigma_1'', \overrightarrow{\psi_1''}, \delta'' \rangle$ ***and***

$K_2 \twoheadrightarrow_{\ell_2} K_2' \twoheadrightarrow^* \langle \overrightarrow{p_2''}, \sigma_2'', \overrightarrow{\psi_2''}, \delta'' \rangle$

implies $\overrightarrow{p_1''} \sim_L \overrightarrow{p_2''}$ *and* $\sigma_1'' \sim_L \sigma_2''$ *and* $\overrightarrow{\psi_1''} \sim_L \overrightarrow{\psi_2''}$.

Definitions 7 and 8 are sufficient to describe the guarantees made by the most secure schedulers from Section IV. We next formally define these schedulers, then prove that the schedulers provide the security guarantees that we have ascribed to them.

B. Scheduler Definitions

We now give formal semantics for the three schedulers from Section V that can make the strongest security guarantees: Sequential-2, Multiplex-2, and Lattice-Based. Each scheduler is defined as the composition of a series of filters, and each filter whittles down the set of threads that are allowed to execute in each step. The filters' job is to provide at most one thread that is allowed to execute. If the filter does not provide a thread, then the scheduler executes a noop.

The scheduler semantics from Section IV-B defines a select function which returns a thread, its state, and its level. To define this select function as the composition of filters requires some formal machinery, which we present in Figure 8. The select function performs three operations: configs, a series of $filters$, and noop?. The configs operation maps each thread to the set of slots in which it can run. This mapping is parameterized by the function slots, which assigns threads to one or more slots. For each thread-slot pair, configs generates a *thread configuration* t that consists

Selection Semantics

$$\mathsf{select} : (\overrightarrow{\mathit{Sub\text{-}program}} \times \overrightarrow{\mathit{State}}) \to (\mathit{Sub\text{-}program} \times \mathit{State} \times \mathcal{L})$$
$$\equiv \mathsf{noop?} \circ \mathit{filter}^* \circ \mathsf{configs}$$
$$T = \overline{\langle p : \mathit{Sub\text{-}program}, \psi : \mathit{State}, \ell : \mathcal{L}, \mathit{slot} : \mathbb{Z} \rangle}$$
$$\mathsf{configs} : (\overrightarrow{\mathit{Sub\text{-}program}} \times \overrightarrow{\mathit{State}}) \to T$$
$$\mathit{filter} : T \to T$$
$$\mathsf{noop?} : T \to (\mathit{Sub\text{-}program} \times \mathit{State} \times \mathcal{L})$$

Thread Configurations

$$\mathsf{configs}(\overrightarrow{p}, \overrightarrow{\psi}) = \bigcup_{\ell \in \mathcal{L}} \{ \langle \overrightarrow{p}[\ell], \overrightarrow{\psi}[\ell], \ell, s \rangle \mid s \in \boxed{\mathsf{slots}}(\ell) \}$$

Filters

$$\mathsf{running}(T) = \{ t \in T \mid t.\psi \neq \mathbf{T} \}$$
$$\mathsf{min}(T) = \{ t \in T \mid \forall t' \in T : t \neq t' \Rightarrow t.\ell \boxed{<} t'.\ell \}$$
$$\mathsf{round\text{-}robin}(T) = \{ t \in T \mid \boxed{\mathsf{next}} = t.\mathit{slot} \}$$
$$\mathsf{min\text{-}depth}(T) = \{ t \in T \mid \forall t' \in T : t'.\ell \sqsubset t.\ell \Rightarrow t'.\psi = \mathbf{T} \}$$

noop Execution

$$\mathsf{noop?}(T) = \begin{cases} (t.p, t.\psi, t.\ell) & : (T = \{t\}) \wedge (t.\psi = \mathbf{R}) \\ (\mathsf{noop}, \mathbf{R}, t.\ell) & : (T = \{t\}) \wedge (t.\psi \neq \mathbf{R}) \\ (\mathsf{noop}, \mathbf{R}, \bot) & : \text{otherwise} \end{cases}$$

Figure 8. Scheduling strategy components (**R** = **Ready**, **T** = **Terminated**). A scheduler is defined over a set of thread configurations, which combines information about each thread with the thread's assigned slot(s). Each scheduler relies on the composition of one or more *filter*s, which eventually selects at most one thread to execute. If the filters do not choose a running thread to execute, then noop? executes a noop. The boxed functions indicate parameters to the scheduler whose behavior can affect the scheduler's security guarantees.

of the thread's program, state, level, and slot number. These configurations provide enough information to enable filters to compose.

Each *filter* operation whittles down the full set of thread configurations T by applying selection criteria to the set. Filters can be composed so that, eventually, the *filter* operation chooses at most one thread configuration. If the filter chooses a configuration whose thread is in the running state, then noop? returns this configuration. Otherwise, noop? returns a noop thread that performs no work.

Figure 8 lists several possible filters, which can be composed to specify the Sequential-2, Multiplex-2, and Lattice-Based schedulers. Sequential-2 is defined by the filter composition min∘running. The scheduler's slots function assigns the same slot to every security level. The $<$ relation extends the security lattice's partial order to a total order. This extension ensures that the filter always returns a singleton set, though the set's element may correspond to a **Blocked** thread.

Multiplex-2 is defined by the filter round-robin. The scheduler's slots function assigns a unique slot to each security level. The scheduler's next function iterates through the slots using modular arithmetic. This function ensures that the filter always returns a singleton set, though the set's element may correspond to a **Blocked** or **Terminated** thread.

Lattice-Based is defined by the filter composition round-robin ∘ min-depth ∘ running. The scheduler's slots function is the donation strategy defined in Section V-E3. This strategy may map a single thread to multiple slots and multiple threads to the same slot. However, it guarantees that any two non-comparable threads are never mapped to the same slot. The scheduler's next function iterates through the security level's slots using modular arithmetic.

The boxed operations slots, $<$, and next in Figure 8 indicate parameters to the scheduler whose behavior can affect the scheduler's security guarantees.

C. Scheduler Security Guarantees

We define two classes of schedulers, *ordered* and *fixed-step*, and prove their properties. We use these properties to prove the security guarantees provided by the scheduling strategies Sequential-2, Multiplex-2, and Lattice-Based.

—Ordered Schedulers—

An *ordered scheduler* is one that permits an ℓ-level sub-program to execute only when all sub-programs at lower levels have terminated.

Definition 9 (Ordered Scheduler)**.** *The semantic relation $\twoheadrightarrow$ defines an ordered scheduler if it maintains the following invariant for all security levels ℓ:*

$$\langle \overrightarrow{p}, \sigma, \overrightarrow{\psi}, \delta \rangle \twoheadrightarrow_\ell \langle \overrightarrow{p}', \sigma', \overrightarrow{\psi}', \delta' \rangle$$

implies $\forall \ell' \in \mathcal{L} \,.\, \ell' \sqsubset \ell \Rightarrow \overrightarrow{\psi}[\ell'] = \mathbf{Terminated}$

Lemma 3. *Every ordered scheduler enforces strong sensitivity among the comparable levels of $\mathcal{L}$.*

Proof: Let $\ell_1, \ell_2 \in \mathcal{L}$ such that $\ell_1 \sqsubset \ell_2$ and define $L = L_{\ell_1}$. Let $K_1 \sim_L K_2$ and

$$K_1 \twoheadrightarrow_{\ell_2} K_1' \twoheadrightarrow^* \langle \overrightarrow{p_1''}, \sigma_1'', \overrightarrow{\psi_1''}, \delta'' \rangle$$
$$K_2 \twoheadrightarrow_{\ell_2} K_2' \twoheadrightarrow^* \langle \overrightarrow{p_2''}, \sigma_2'', \overrightarrow{\psi_2''}, \delta'' \rangle$$

Because $\ell_2 \in H$, Corollary 1 gives $\overrightarrow{p_i} \sim_L \overrightarrow{p_i'}$, and $\sigma_i \sim_L \sigma_i'$, and $\overrightarrow{\psi_i} \sim_L \overrightarrow{\psi_i'}$, where $i \in \{1, 2\}$. By Definition 9, all subsequent transitions are also in H because all sub-programs at level ℓ_1 and lower must have terminated before the sub-program at level ℓ_2 can execute. Corollary 1 applies, inductively, to give $\overrightarrow{p_i'} \sim_L \overrightarrow{p_i''}$, and $\sigma_i' \sim_L \sigma_i''$, and $\overrightarrow{\psi_i'} \sim_L \overrightarrow{\psi_i''}$, where $i \in \{1, 2\}$. By transitivity, $\overrightarrow{p_i} \sim_L \overrightarrow{p_i''}$, and $\sigma_i \sim_L \sigma_i''$, and $\overrightarrow{\psi_i} \sim_L \overrightarrow{\psi_i''}$, where $i \in \{1, 2\}$. By transitivity, $\overrightarrow{p_1''} \sim_L \overrightarrow{p_2''}$, and $\sigma_1'' \sim_L \sigma_2''$, and $\overrightarrow{\psi_1''} \sim_L \overrightarrow{\psi_2''}$. ∎

—Fixed-Step Schedulers—

A *fixed-step* scheduler is one that enforces a constant time interval between the selection of every slot.

Definition 10 (Fixed-step Scheduler)**.** *A* fixed-step scheduler *is one that (a) defines a deterministic* slots *function that is fixed for every run of every sub-program, (b) defines a* slots *function that assigns distinct slots to any set of security levels whose sub-programs may be multiplexed, and (c) defines a deterministic, history-based* next *function that depends only on the number of steps the scheduler has executed.*

Lemma 4. *Every fixed-step scheduler exhibits weakly-sensitive noninterference.*

Proof: For a given security level ℓ and a pair of low-equivalent configurations $K_1 \sim_L K_2$, let:

$$K_1 \twoheadrightarrow^{n_1} \langle \overrightarrow{p_1'}, \sigma_1', \overrightarrow{\psi_1'}, \delta' \rangle \textbf{ and } K_2 \twoheadrightarrow^{n_2} \langle \overrightarrow{p_2'}, \sigma_2', \overrightarrow{\psi_2'}, \delta' \rangle$$

Under a weak timing model:

$$K_1.\delta + \sum_{j=1}^{n_1} (\delta_s + \delta_p) = \delta' = K_2.\delta + \sum_{j=1}^{n_2} (\delta_s + \delta_p)$$

where δ_p is the constant-time value that the weak timing model gives to each sub-program step. By Definition 5, $K_1.\delta = K_2.\delta$. Because δ_s is constant, it must be that $n_1 = n_2$. Let k represent this number. Then:

$$K_i \twoheadrightarrow_{\ell_i^1} \cdots \twoheadrightarrow_{\ell_i^k} \langle \overrightarrow{p_i'}, \sigma_i', \overrightarrow{\psi_i'}, \delta' \rangle$$

where $i \in \{1, 2\}$. Because the scheduler is fixed-step, $\ell_1^j = \ell_2^j$, for $j = 1 \ldots k$. Either $\ell_i^j : L$ or $\ell_i^j : H$. If $\ell_i^j : L$ then Lemma 2 and weak sensitivity give that the sub-program vectors, stores, and state vectors of K_i^{j-1} and K_i^j are low-equivalent for $j = 2 \ldots k$. The same result is achieved by Corollary 1 and weak timing, if $\ell_i^k : H$. By induction over k, the sub-program vectors, stores, and state vectors of K_i are low-equivalent to $\overrightarrow{p_i'}$, σ_i', and $\overrightarrow{\psi_i'}$, respectively. By transitivity, $\overrightarrow{p_1'} \sim_L \overrightarrow{p_2'}$, and $\sigma_1' \sim_L \sigma_2'$, and $\overrightarrow{\psi_1'} \sim_L \overrightarrow{\psi_2'}$. ■

A fixed-step scheduler does not exhibit strongly-sensitive noninterference among all security levels. Under strong termination, differences in the high portion of stores $K_1.\sigma$ and $K_2.\sigma$ may, for example, cause K_1 to terminate abnormally but not cause K_2 to do so. If the abnormal termination prevents a low sub-program from updating the store, then low-equivalence is not preserved.

—Scheduler Security Guarantees—

Theorem 1 (Sequential-2)**.** *The Sequential-2 scheduler is strongly-sensitive among comparable levels.*

Proof: By Lemma 3, it suffices to show that Sequential-2 is an ordered scheduler. This is true by construction. The scheduler's filter min ∘ ready prevents a sub-program at level ℓ_2 from executing before a sub-program at level $\ell_1 \sqsubset \ell_2$ has terminated. ■

Theorem 2 (Multiplex-2)**.** *The Multiplex-2 scheduler is weakly-sensitive.*

Proof: By Lemma 4, it suffices to show that Multiplex-2 is a fixed-step scheduler. This is true by construction. Every security level may be multiplexed. The scheduler's deterministic slots function assigns a distinct slot to every security level. The scheduler's next function uses modular arithmetic over a finite domain, which means that the slot choice depends only on the number of steps taken. ■

Theorem 3 (Lattice-Based)**.** *The Lattice-Based scheduler is strongly-sensitive among comparable levels and weakly-sensitive among all other levels.*

Proof: By Lemma 4, if Lattice-Based is a fixed-step scheduler than it is weakly-sensitive among all levels. If Lattice-Based also is an ordered scheduler, than it is strongly-sensitive among comparable levels, by Lemma 3. Lattice-Based is an ordered scheduler because the min-depth ∘ running component of the scheduler's filter prevents a sub-program at level ℓ_2 from executing before a sub-program at level $\ell_1 \sqsubset \ell_2$ has terminated. Lattice-Based is a fixed-step scheduler because only non-comparable levels may be multiplexed and the scheduler's deterministic slots function assigns a distinct set of slots to any subset of non-comparable security levels. The scheduler's next function uses modular arithmetic over a finite domain, which means that the slot choice depends only on the number of steps taken. ■

VII. Further Discussion

In this section we first discuss the relation of our work to an earlier paper on timing- and termination-sensitive secure information flow called "Noninterference Through Secure Multi-Execution," by Devriese and Piessens [13]. We then discuss how we might incorporate a form of declassification into our scheme.

A. Secure Multi-Execution

Secure Multi-Execution (SME), described by Devriese and Piessens [13], is the first description and practical implementation of an instance of the scheduling approach to eliminating timing and termination leaks. The SME paper contains formal claims and open questions that are addressed by our work.

1) First SME Scheduling Strategy ($\mathsf{select}_{lowprio}$*):* The $\mathsf{select}_{lowprio}$ strategy described in the SME paper is equivalent to our Sequential-2 strategy. The SME paper claims that this scheduling strategy enforces weakly-sensitive noninterference (what the paper refers to as "strong noninterference") among all security levels [13, § IV-A]. Our paper shows that this claim is not true for non-comparable security

levels and that this claim may be strengthened for comparable security levels. The SME paper's proof of weakly-sensitive noninterference is valid for comparable security levels, and indeed we showed in Section VI-C that it can be strengthened to a proof of strongly-sensitive noninterference. The SME paper's mistaken claim of weakly-sensitive noninterference among non-comparable levels relies on the assumed extension of a partially ordered security lattice to a totally ordered one [13, § III-B]. However, we have shown in Sections V-A and VI-C that, although such an extension does make it possible to employ $\mathsf{select}_{lowprio}$, any proof of security guarantees that relies on a total order does not necessarily apply to the original, partially ordered lattice.

2) Second SME Scheduling Strategy (unnamed): The SME paper proposes an unnamed second scheduling strategy that is not formalized or proven secure, but is implemented and evaluated in the paper [13, § II-B, § V-A]. The description of the strategy is ambiguous.

Assuming that threads are time-multiplexed on the same processor, this unnamed strategy is equivalent to Multiplex-1. Thus, as shown in Section V, it cannot guarantee anything stronger than insensitive noninterference.

Assuming instead that the number of threads is limited to the number of processor cores and each thread is pinned to a separate core[6], this same strategy is equivalent to Multiplex-2 and can guarantee at most weakly-sensitive noninterference.

3) Nonterminating Sub-programs: The SME paper's security guarantees are only made for terminating runs of sub-programs. The paper raises an open question: whether any scheduling strategy can guarantee progress of high sub-programs while also guaranteeing noninterference [13, § IV-A]. We show that this is possible using the Multiplex-2 strategy, but that the strongest guarantee that can be given for this case is weakly-sensitive noninterference.

B. Declassification

Strict noninterference between security levels is sometimes too limiting—some applications need the ability to *declassify* information, allowing a lower security level to observe information that comes from higher in the lattice [26], [40]. Our current framework does not allow declassification, but we believe that it can be accommodated under certain constraints. The main constraint is that declassifying some data D must necessarily declassify the timing and termination behavior of the computation that computes D.

Under this constraint, declassification can be handled as follows. When partitioning a program into sub-programs, create an additional sub-program to compute D, the data that will be declassified to a lower security level L (this extra partitioning can be done using program slicing, multi-execution, or whatever other technique is used for partitioning the original program). Then schedule the execution of that sub-program as if it had security level L (while still keeping the sub-program logically separate from the sub-program that is actually at level L). In this way, D can be declassified while still preserving insensitive noninterference.

[6] According to the authors, this is the intended interpretation [14].

VIII. Conclusion

Secure information flow is a critical requirement for ensuring the privacy and integrity of information. Covert channels such as timing and termination are a particularly difficult source of information leaks to prevent. Attackers can use these covert channels to leak arbitrary amounts of information, even in programs that have been certified as secure using timing- and termination-insensitive noninterference properties. We believe that timing- and termination-sensitive secure information flow will only become more important over time. As techniques are developed and employed to prevent the more standard types of leaks (i.e., explicit and implicit leaks), attackers will have ever more motivation to exploit these covert channels. We believe that our contributions in this paper will help the research community to better understand and address the security challenges of the future.

Existing work in timing- and termination-sensitive secure information flow uses a variety of formal definitions and enforcement strategies. We clarify this research area by (1) describing and discussing in detail different definitions of timing- and termination-sensitivity; (2) exploring a space of scheduling strategies for enforcing timing- and termination-sensitive noninterference; and (3) formally proving the security guarantees implied by a variety of scheduling strategies. As an outgrowth of this clarification, we introduce a new scheduling strategy that provides stronger security guarantees than any in existing work, and we address issues with existing work by demonstrating an error in a formal claim of security, strengthening another formal security guarantee, and answering open questions.

Acknowledgements: We thank Frank Piessens, Dominique Devriese, and the anonymous reviewers for their comments on this paper.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 147–160, 1999.

[2] J. Agat. Transforming out timing leaks. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 40–53, 2000.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice Hall, 1993.

[4] T. Amtoft and A. Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Computer Programming (SCP)*, 64(1):3–28, 2007.

[5] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *European Symposium on Research in Computer Security (ESORICS)*, pages 333–348, 2008.

[6] A. Askarov, A. C. Myers, and D. Zhang. Predictive black-box mitigation of timing channels. In *ACM Conference on Computer and Communications Security (CCS)*, pages 297–307, 2010.

[7] G. Barthe, T. Rezk, and M. Warnier. Preventing Timing Leaks Through Transactional Branching Instructions. *Electronic Notes Theoretical Computer Science*, 153:33–55, May 2006.

[8] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *International Conference on World Wide Web (WWW)*, pages 621–628, 2007.

[9] G. Boudol and I. Castellani. Noninterference for concurrent programs. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 382–395, 2001.

[10] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX Security Symposium*, pages 1–14, 2003.

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

[12] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM (CACM)*, 20(7):504–513, 1977.

[13] D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[14] D. Devriese and F. Piessens. Personal communication, November 2010.

[15] J. F. Dhem, F. Koeune, P. A. Leroux, P. Mestré, J. J. Quisquater, and J. L. Willems. A Practical Implementation of the Timing Attack. In *International Conference on Smart Card Research and Applications (CARDIS)*, pages 167–182, 2000.

[16] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *ACM Conference on Computer and Communications Security (CCS)*, pages 25–32, 2000.

[17] R. Giacobazzi and I. Mastroeni. Timed Abstract Non-interference. In *Formal Modeling and Analysis of Timed Systems*, volume 3829 of *Lecture Notes in Computer Science*, chapter 22, pages 289–303. 2005.

[18] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[19] C. Hammer, J. Krinke, and G. Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, 2006.

[20] H. Handschuh and H. M. Heys. A Timing Attack on RC5. In *SAC '98: Proceedings of the Selected Areas in Cryptography*, pages 306–318, 1999.

[21] D. Hedin and D. Sands. Timing aware information flow security for a javacard-like bytecode. *Electronic Notes in Theoretical Computer Science*, 141(1):163–182, 2005.

[22] W. M. Hu. Reducing timing channels with fuzzy time. In *IEEE Symposium on Security and Privacy*, pages 8–20, 1991.

[23] W. M. Hu. Lattice Scheduling and Covert Channels. In *IEEE Symposium on Security and Privacy*, pages 52–61, 1992.

[24] T. Khatiwala, R. Swaminathan, and V. N. Venkatakrishnan. Data Sandboxing: A Technique for Enforcing Confidentiality Policies. In *Annual Computer Security Applications Conference (ACSAC)*, pages 223–234, 2006.

[25] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 104–113, 1996.

[26] A. C. Myers and B. Liskov. A decentralized model for information flow control. *SIGOPS Operating Systems Review (OSR)*, 31(5):129–142, October 1997.

[27] P. R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1–3):197–207, 2002.

[28] P. M. Pardalos and J. Xue. The maximum clique problem. *Journal of Global Optimization*, 4(3):301–328, 1994.

[29] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Asian Computing Science Conference on Advances in Computer Science (ASIAN)*, pages 120–135, 2007.

[30] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *IEEE Workshop on Computer Security Foundations*, pages 177–189, 2006.

[31] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI)*, pages 474–480, 2007.

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

[33] P. Shroff and S. F. Smith. Securing timing channels at runtime. Technical report, Department of Computer Science, The Johns Hopkins University, July 2008.

[34] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 355–364, 1998.

[35] S. F. Smith and M. Thober. Refactoring programs to secure information flows. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 75–84, 2006.

[36] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 243–254, 2004.

[37] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computing Security*, 4(2-3):167–187, 1996.

[38] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *IEEE Workshop on Computer Security Foundations*, page 156, 1997.

[39] W. H. Wong. Timing attacks on RSA: revealing your secrets through the fourth dimension. *Crossroads*, 11(3):5, May 2005.

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

[41] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *IEEE Workshop on Computer Security Foundations*, pages 29–43, 2003.