2020 IEEE 33rd Computer Security Foundations Symposium (CSF)

Transparent IFC Enforcement: Possibility and (In)Efficiency Results

Maximilian Algehed
Computer Science and Engineering
Chalmers
Göteborg, Sweden
algehed@chalmers.se

Cormac Flanagan
Computer Science and Engineering
University of California Santa Cruz
Santa Cruz, USA
cormac@ucsc.edu

***Abstract*—Information Flow Control (IFC) is a collection of techniques for ensuring a no-write-down no-read-up style security policy known as *noninterference*. Traditional methods for both static (e.g. type systems) and dynamic (e.g. runtime monitors) IFC suffer from untenable numbers of false alarms on real-world programs. Secure Multi-Execution (SME) promises to provide secure information flow control without modifying the behaviour of already secure programs, a property commonly referred to as *transparency*. Implementations of SME exist for the web in the form of the FlowFox browser and as plug-ins to several programming languages. Furthermore, SME can in theory work in a black-box manner, meaning that it can be programming language agnostic, making it perfect for securing legacy or third-party systems. As such SME, and its variants like Multiple Facets (MF) and Faceted Secure Multi-Execution (FSME), appear to be a family of panaceas for the security engineer. The question is, how come, given all these advantages, that these techniques are not ubiquitous in practice?**

The answer lies, partially, in the issue of runtime and memory overhead. SME and its variants are prohibitively expensive to deploy in many non-trivial situations. The natural question is *why* is this the case? On the surface, the reason is simple. The techniques in the SME family all rely on the idea of *multi-execution*, running all or parts of a program multiple times to achieve noninterference. Naturally, this causes some overhead. However, the predominant thinking in the IFC community has been that these overheads can be overcome. In this paper we argue that there are fundamental reasons to expect this not to be the case and prove two key theorems:

- **All transparent enforcement is polynomial time equivalent to multi-execution.**
- **All black-box enforcement takes time exponential in the number of principals in the security lattice.**

Our methods also allow us to answer, in the affirmative, an open question about the possibility of secure and transparent enforcement of a security condition known as Termination Insensitive Noninterference.

***Index Terms*—Secure Multi-Execution, Information Flow Control, Noninterference, Transparency, Black-Box, White-Box, Efficiency**

I. Introduction

Language-Based Information Flow Control (IFC) [1–4] is a promising technology for securing systems against malicious third-party code. Approaches to IFC typically appear in the form of either a type system [3–5] or a custom programming language semantics [6, 7]. Traditionally, these approaches either statically or dynamically detect behaviour that violates the security criteria of *Noninterference* [1, 8–10] (secrets cannot influence attacker-observable behaviour) and either raise a type or runtime error. However, in order to be sound, even in the state-of-the-art implementations, these techniques need to be conservative and therefore suffer from unmanageable numbers of false alarms [11, 12].

To remedy this issue, techniques have recently been developed for ensuring so called *transparent* IFC [7, 13–17]. This line of work promises to provide security without raising false alarms. Because precisely detecting violations of noninterference is impossible [2] to do both statically and dynamically, methods for transparent IFC instead silently modify programs to ensure noninterference by construction.

All known techniques for providing transparent IFC share one feature, they are all based on multi-execution. This technique was pioneered by Devriese and Piessens [13] in their seminal paper on Secure Multi-Execution (SME). Under SME, the program being made secure is run once for each security level with carefully adapted inputs to ensure noninterference while removing false alarms.

To see an example of how this works, consider the two security levels L (for Low or Public) and H (for High or Secret). Noninterference stipulates that low input is allowed to influence high output, but not the other way around. Under SME, a program p that takes both high and low inputs and produces both high and low output is run twice; one run of p is given only the low input and produces the low part of the output, and the other run of p is given both low and high input and contributes only the high output: see Figure 1.

SME is a black-box enforcement mechanism, it does not need access to the source code of p to work. The ability to secure any program in a black-box manner is powerful and has potential applications in many areas, including databases [18], legacy code [19], and browsers [20].

The black-box property of SME is shared by some [17, 21], but not all [7, 14, 15] transparent enforcement mechanisms in the literature. However, the idea of multi-execution, running the same code multiple times with slightly different inputs, is

The first author was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation and The Osher Endowment's through project Ghost: Exploring the Limits of Invisible Security. The second author was partially supported by NSF grant CCF-1813133

© 2020, Maximilian Algehed. Under license to IEEE.
DOI 10.1109/CSF49147.2020.00013

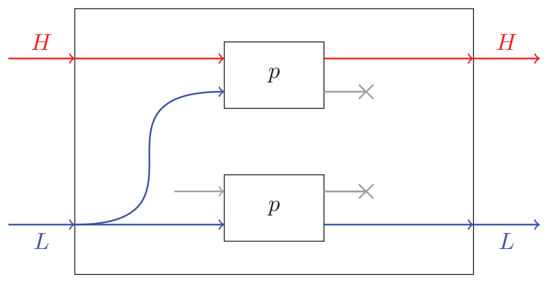

Fig. 1: Multi-Execution of the program p for the two-point lattice.

shared among all known mechanisms in one form or another. Consequently, these mechanisms, no matter where they lie on the scale from black-box to white-box, suffer severe performance penalties as the number of multi-executions grows [7, 14, 16].

In this paper, we provide a unifying formal, extensional, framework for studying multi-execution that is sufficiently expressive to formulate and prove a number of theorems that explain:

1) Why all transparent enforcement mechanisms rely on multi-execution,
2) Why all transparent black-box enforcement mechanisms are inefficient (as seen empirically in previous work, e.g. [7, 14, 16, 22])

Along the way, we use our framework to answer the open question of whether or not Termination Insensitive Noninterference can be transparently enforced. It turns out that it is possible, but inefficient.

Concretely, we provide the following contributions:

- A novel yet simple framework for reasoning about secure programs and enforcement mechanisms (Section II).
- A precise characterisation of the transparency guarantees provided by multi-execution (Section III).
- A black-box version of secure multi-execution for infininte lattices (Section III).
- A novel enforcement mechanism that is sound and transparent for Termination Insensitive Noninterference (Section IV).
- A general framework for optimising multi-execution (Section V) in which we prove a number of results including conditions for safe and transparent optimisation.
- A proof that any secure and transparent enforcement is poly-time equivalent to a multi-execution based enforcement mechanism (Section V).
- A proof that all secure and transparent black-box enforcement mechanisms are inefficient (Section VI).

II. An Extensional Framework for Secure Information Flow

In this section we develop an extensional framework for reasoning about secure information flow. The goal is to create a simple, yet flexible, context in which we can reason both about *possibility* of secure and transparent enforcement as well as *efficiency*.

Programs with Labeled Inputs and Outputs

As is usual in IFC research, we assume a join semi-lattice $\langle \mathcal{L}, \sqsubseteq, \bot, \sqcup \rangle$ that encodes security labels in $\mathcal{L}$ (we use the words label and level interchangeably), the permitted flows between them by the order $\_ \sqsubseteq \_ \subseteq \mathcal{L} \times \mathcal{L}$, the least privileged label with $\bot \in \mathcal{L}$, and a least-upper-bound operation $\_ \sqcup \_ : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$. We let the variables ℓ, ℓ_i, ℓ', $\jmath$, $\jmath_i$, $\jmath'$ etc. range over elements of $\mathcal{L}$ and we write a^ℓ for the pair (a, ℓ).

We work in a non-interactive setting, so a program takes input and eventually produces output (or diverges). The input (and output) is typically a compound data-structure consisting of various "pieces", where each piece can be annotated with its own security label. For example, the input could be a set of files, where each file has a name, contents, and a security label. To model this kind of setting in a general way we assume that both the input and the output is a set of labeled data, and so we formalise the semantics of programs as partial recursive functions $p : \mathcal{P}(A \times \mathcal{L}) \to_p \mathcal{P}(B \times \mathcal{L})$.

Here, $\mathcal{P}(\bullet)$ denotes the power-set constructor and $X \to_p Y$ denotes *partial* functions from X to Y, while $X \to Y$ denotes total functions. One benefit of working with this "sets of labeled data" formalism is that it is simple to remove high-security information from an input or output $x \in \mathcal{P}(A \times \mathcal{L})$ via the following so-called ℓ*-projection* operation:

$$x \downarrow \ell = \{\ a^\jmath \mid a^\jmath \in x, \jmath \sqsubseteq \ell\}$$

Similarly, we can collect the *labels of* x as:

$$\mathcal{L}(x) = \{\ \ell \mid a^\ell \in x\}$$

And finally, we define the ℓ*-selection of* x, i.e. all the elements of x at level ℓ, as:

$$x@\ell = \{\ a^\ell \mid a^\jmath \in x,\ \jmath = \ell\}$$

We refer to the *syntax* of p simply as p and the *semantics* of p as $p(x)$ for some input x. Because the choice of programming language is orthogonal to our purposes, we keep it abstract. We write $p(x)^\checkmark$ when $p(x)$ is defined and write $p(x)^\times$ to mean that $p(x)$ is undefined (i.e. that p diverges on input x).

Next we formalise what we mean by noninterfering programs. Intuitively, a program p is noninterfering if, given inputs x and y that the attacker considers observably equivalent, p produces outputs $p(x)$ and $p(y)$ that the attacker also considers observably equivalent.

Definition 1. We call $x, y \in \mathcal{P}(A \times \mathcal{L})$ ℓ*-equivalent*, written $x \sim_\ell y$, if and only if the ℓ-projection of x and y are the same.

$$x \sim_\ell y \ \triangleq\ x \downarrow \ell = y \downarrow \ell$$

A program p is *noninterfering* if, given any ℓ and two x and y that are ℓ-equivalent such that both $p(x)$ and $p(y)$ are defined, we have that $p(x)$ is ℓ-equivalent to $p(y)$.

$$p \text{ is noninterfering} \triangleq$$

$$\forall \ell. \forall x, y.\ x \sim_\ell y \wedge p(x)^{\checkmark} \wedge p(y)^{\checkmark} \Rightarrow p(x) \sim_\ell p(y)$$

■

Example 1. We present a number of example programs that serve as running examples throughout the paper. First is the program *id*, the identity function, which is noninterfering.

$$\mathit{id}(x) \triangleq x$$

The next program combines information from multiple security labels, but is still noninterfering since the label on the output appropriately reflects the dependencies on the labeled input. We use $|x|$ to denote the size of the set x.

$$\mathit{combine}(x) \triangleq \{(|x@\text{Alice}| + |x@\text{Bob}|)^{\text{Alice}\sqcup\text{Bob}\sqcup\text{Charlie}}\}$$

Labels Alice, Bob, and Charlie are notational shorthand for labels $\{\text{Alice}\}$, $\{\text{Bob}\}$ and $\{\text{Charlie}\}$ in the power-set lattice $\mathcal{P}(P)$ for some set of *principals* P which contains Alice, Bob, and Charlie.

The program *combineAll* below is similar to *combine*, but is interfering because it combines everything, rather than a specific subset of the inputs.

$$\mathit{combineAll}(x) \triangleq \{|x|^{\bigsqcup \mathcal{L}(x)}\}$$

To see that *combineAll* is interfering, consider that $\emptyset \sim_L \{0^H\}$ but:

$$\mathit{combineAll}(\emptyset) = \{0^L\} \not\sim_L \{1^H\} = \mathit{combineAll}(\{0^H\})$$

Another example of an interfering program is *leakBit*, which leaks one bit in the H input to L.

$$\mathit{leakBit}(x) \triangleq \text{if } 1^H \in x \text{ then } \{1^L\} \text{ else } \{0^L\}$$

The program *leakAll* below takes x and re-labels everything in the input to $\bot$; it is also interfering.

$$\mathit{leakAll}(x) \triangleq \{\ a^{\bot} \mid a^{\ell} \in x\ \}$$

Finally, the program *termLeak* below leaks information via termination, but is still noninterfering because our definition of noninterference does not consider the termination channel.

$$\mathit{termLeak}(x) \triangleq \text{ if } 1^H \in x \text{ then } \emptyset \text{ else diverge}$$

To summarise, programs *id*, *combine* and *termLeak* are noninterfering, while *combineAll*, *leakBit* and *leakAll* are not. ■

Secure Programs and Termination Criteria

The notion of noninterference introduced above does not prevent leaks through the termination channel, as illustrated by the *termLeak* example, and so captures what is normally called "Termination Insensitive" noninterference [23]. The prevention of termination leaks is a challenging topic for IFC enforcement, so to study it we introduce the following four *termination criteria*.

Definition 2 (Termination Criteria)**.**

- All programs are *Termination Insensitive* (TI).
- Program p is *Monotonically Terminating* (MT) if and only if whenever $p(x)$ is defined, so is $p(x \downarrow \ell)$ for all ℓ.

$$p \text{ is MT } \triangleq \forall x.\ p(x)^{\checkmark} \Rightarrow \forall \ell. p(x \downarrow \ell)^{\checkmark}$$

- Program p is *Termination Sensitive* (TS) if and only if for all ℓ and ℓ-equivalent x and y, $p(x)$ is defined if and only if $p(y)$ is.

$$p \text{ is TS } \triangleq \forall \ell. \forall x, y. x \sim_\ell y \Rightarrow (p(x)^{\checkmark} \Leftrightarrow p(y)^{\checkmark})$$

- p is Total if it always terminates.

$$p \text{ is Total } \triangleq \forall x. p(x)^{\checkmark}$$

■

We let the meta-variable τ range over termination criteria.

$$\tau ::= \text{TI} \mid \text{MT} \mid \text{TS} \mid \text{Total}$$

The termination criteria TS and TI are standard from the IFC literature [23–25], and the notion of total programs is also a natural termination criteria. Our new criteria, MT, is motivated by the termination requirements of SME, as we discuss below. MT is a weaker requirement than TS, but stronger than TI, which results in the following ordering of termination criteria.

Proposition 1. *The following chain of implications is strict.*

$$\begin{aligned} & p \textit{ is Total} \\ \Rightarrow\ & p \textit{ is TS} \\ \Rightarrow\ & p \textit{ is MT} \\ \Rightarrow\ & p \textit{ is TI} \end{aligned}$$

Recall that an implication $P \Rightarrow Q$ is *strict* if it is *not* the case that $P \Leftrightarrow Q$. Using the termination criteria, we define a family of notions of secure program in a way that separates termination and noninterference.

Definition 3 (τ-secure)**.** A program is τ*-secure* if and only if it is τ and noninterfering. ■

The τ-secure criteria are naturally ordered the same way as the termination criteria.

Proposition 2. *The following chain of implications is strict.*

$$\begin{aligned} & p \textit{ is Total-secure} \\ \Rightarrow\ & p \textit{ is TS-secure} \\ \Rightarrow\ & p \textit{ is MT-secure} \\ \Rightarrow\ & p \textit{ is TI-secure} \end{aligned}$$

Program	**NI**	**Termination**	**Security**
id	yes	Total	Total-secure
combine	yes	Total	Total-secure
combineAll	no	Total	×
leakBit	no	Total	×
leakAll	no	Total	×
termLeak	yes	TI	TI-secure
divergeIfLPresent	yes	TS	TS-secure
divergeIfHPresent	yes	MT	MT-secure
divergeIfHAbsent	yes	TI	TI-secure

TABLE I: Noninterference (NI), termination, and security of example programs

We re-visit some programs from Example 1 to illustrate the various termination criteria.

Example 2. The programs *id* and *combine* from Example 1 are noninterfering and Total; hence they are both Total-secure. The program *termLeak* is noninterfering, but its termination is influenced by H information, and so it is only TI-secure. Program *leakBit*, *combineAll*, and *leakAll* meanwhile are interfering and hence are not τ-secure for any τ. ■

Next we present three noninterfering programs that illustrate different termination criteria.

Example 3. First, the following program is not Total, as it sometimes diverges, but divergence only depends on public (L-labeled) information, so it is TS.

$$\textit{divergeIfLPresent}(x) \triangleq \text{if } 1^L \in x \text{ then } x \text{ else diverge}$$

In the next program, divergence depends on H information, so it is not TS. It is MT as removing H information only improves termination.

$$\textit{divergeIfHPresent}(x) \triangleq \text{if } 1^H \in x \text{ then diverge else } \emptyset$$

In contrast, for the following program, removing H information hurts termination, so this program is TI but not TS.

$$\textit{divergeIfHAbsent}(x) \triangleq \text{if } 1^H \notin x \text{ then diverge else } \emptyset$$

For a brief summary of this example and examples 1 and 2, see Table I. ■

A version of MT-security has been studied in the past by Rafnsson and Sabelfeld [21] and Jaskelioff and Russo [26]. These authors consider programs whose termination is stable under "default" inputs in a statically labeled setting. Specifically, they formulate definitions of security that require that the program preserves ℓ-equivalence and that replacing inputs with default values does not cause non-termination. This is analogous to MT-security, requiring that termination is preserved when the high inputs have a default value, like 0, is identical to saying that termination is preserved when the H-selection of the input is the "default value" $\emptyset$.

Enforcement Mechanisms

Next we turn our attention to mechanisms for enforcing τ-security.

Definition 4 (Enforcement Mechanism)**.** An enforcement mechanisms E is a polynomial-time total recursive function that takes any program $p : \mathcal{P}(A \times \mathcal{L}) \to_p \mathcal{P}(B \times \mathcal{L})$ to a program $E[p] : \mathcal{P}(A \times \mathcal{L}) \to_p \mathcal{P}(B \times \mathcal{L})$. ■

Normally, we talk about two properties of enforcement mechanisms: security and transparency [7, 13, 14, 17, 24, 27]. Security means that applying a secure enforcement mechanism E to any program p always yields a secure program $E[p]$. Transparency, on the other hand, means that given a secure program p, applying E does not change the observable behaviour of p.

Definition 5 (Security and Transparency)**.** An enforcement mechanism E is (1) τ-secure and (2) τ'-transparent if:

1) For all p, $E[p]$ is τ-secure
2) For all τ'-secure p and inputs x, $E[p](x) = p(x)$.

We say that E is τ-τ' if it is τ-secure and τ'-transparent. ■

The chains of implications from propositions 1 and 2 are reflected in the definitions of security and transparency for enforcement mechanisms.

Proposition 3. *The following two chains of implications are strict.*

$$\begin{array}{rl|rl} & E \textit{ is Total-secure} & & E \textit{ is TI-transparent} \\ \Rightarrow & E \textit{ is TS-secure} & \Rightarrow & E \textit{ is MT-transparent} \\ \Rightarrow & E \textit{ is MT-secure} & \Rightarrow & E \textit{ is TS-transparent} \\ \Rightarrow & E \textit{ is TI-secure} & \Rightarrow & E \textit{ is Total-transparent} \end{array}$$

Note that the transparency ordering is the converse of the security ordering. This is because security requires the mechanism to *provide* τ-security, while transparency allows the mechanism to *rely* on τ-security. Bearing these orderings in mind, if we write that E is not secure, we mean that it is not TI-secure, and if we write that E is not transparent, we mean that it is not Total-transparent.

Example 4. We give a number of examples of secure and transparent enforcement mechanisms:

- $E[p](x) \triangleq \emptyset$ is Total-secure, but is not transparent.
- $E[p](x) \triangleq p(x)$ is TI-transparent, but not secure.
- No Sensitive Upgrade (NSU) [28] is TI-secure but not transparent.
- Permissive Upgrade (PU) [29] is TI-secure and transparent for more programs than NSU [30], but still not transparent.
- The Hybrid Monitoring (HM) technique is TI-secure [30].
- Zanarini et al. [17] give an enforcement mechanism that is TI-secure and MT-transparent.
- SME [13] provides MT-security and MT-transparency (see below).

■

Multi-Execution

Inspired by Secure Multi-Execution (SME) [13], we define the enforcement mechanism ME (for "Multi-Execution") in our framework. $\text{ME}[p](x)$ runs p multiple times, once for each security label ℓ in the lattice on appropriately censored input $x \downarrow \ell$ that only contains information visible to ℓ, and uses the output of $p(x \downarrow \ell)$ to construct the ℓ-labeled part of final output of $\text{ME}[p](x)$.

Definition 6. $\text{ME}[p](x) \triangleq \bigcup\{\ p(x \downarrow \ell)@\ell \mid \ell \in \mathcal{L}\ \}$ ■

This mathematical definition of ME can naturally be implemented by iterating over all labels ℓ in $\mathcal{L}$, provided $\mathcal{L}$ is finite. Section III deals with the implementation in the case where $\mathcal{L}$ is non-finite.

Figure 1 shows a graphical rendition of $\text{ME}[p]$ for the two-point lattice $\{L, H\}$. The outer box represents $\text{ME}[p](x)$, it takes both H and L input from the arrows on the left, and produces both H and L output in the arrows to the right. Both the H and L parts of the input are used in the top-most run of p, and only the L part of the input is used in the bottom-most run. The top-most run, the one given both the H and L input, then produces only the H output, whereas the bottom-most run contributes only the final L output. Noninterference for this construction is easy to show, the L output can only be influenced by the L input, as that run of p never sees the actual H input.

For example, consider how ME works with the program *leakBit* from Example 1 on inputs $\emptyset$ and $\{1^H\}$.

$$\textit{leakBit}(x) \triangleq \text{ if } 1^H \in x \text{ then } \{1^L\} \text{ else } \{0^L\}$$

Note that $\emptyset \sim_L \{1^H\}$ and so we expect that $\text{ME}[\textit{leakBit}](\emptyset) \sim_L \text{ME}[\textit{leakBit}](\{1^H\})$. As *leakBit* is Total we can simply compute to see that this does indeed hold.

$$\begin{array}{ll|ll}
 & \text{ME}[\textit{leakBit}](\emptyset) \downarrow L & & \text{ME}[\textit{leakBit}](\{1^H\}) \downarrow L \\
= & \textit{leakBit}(\emptyset \downarrow L) \ \downarrow L & = & \textit{leakBit}(\{1^H\} \downarrow L) \ \downarrow L \\
= & \textit{leakBit}(\emptyset) \quad \downarrow L & = & \textit{leakBit}(\emptyset) \quad \downarrow L \\
= & \{0^L\} \quad \downarrow L & = & \{0^L\} \quad \downarrow L \\
= & \{0^L\} & = & \{0^L\}
\end{array}$$

Furthermore, when p is TI-secure and both $\text{ME}[p](x)$ and $p(x)$ terminate, the output of $\text{ME}[p](x)$ has to equal that of $p(x)$. The H part of $\text{ME}[p](x)$ is clearly the same as the H part of $p(x)$, and because p is noninterfering it follows that the L output of $p(x)$ doesn't depend on the H input, and so the L output of $p(x)$ must be the same as the L part of $p(x \downarrow L)$. Putting these two facts together allows us to conclude that $\text{ME}[p](x)$ is the same as $p(x)$ when both terminate.

To see how ME deals with termination, consider program *divergeIfHAbsent* from Example 3:

$$\textit{divergeIfHAbsent}(x) \triangleq \text{ if } 1^H \notin x \text{ then diverge else } \emptyset$$

It diverges if $1^H \notin x$ and terminates with result $\emptyset$ otherwise. $\text{ME}[\textit{divergeIfHAbsent}]$ meanwhile runs both $\textit{divergeIfHAbsent}(x)$ and $\textit{divergeIfHAbsent}(x \downarrow L)$, which means that $\text{ME}[\textit{divergeIfHAbsent}]$ always diverges, as $1^H \notin x \downarrow L$ for all x.

However, if we consider $\text{ME}[\textit{divergeIfHPresent}]$ we get a more interesting result. Unlike *divergeIfHAbsent*, *divergeIfHPresent* diverges when $1^H \in x$ and so if $1^H \notin x$, then we have $\text{ME}[\textit{divergeIfHPresent}](x)^{\checkmark}$ with output $\emptyset$. As termination of $\text{ME}[\textit{divergeIfHPresent}](x)$ *increases* as we remove H elements from x and the output is constant, and $\text{ME}[\textit{divergeIfHAbsent}](x)$ always diverges, we see that ME behaves in an MT-secure manner for both programs.

Monotonic Termination of $\text{ME}[p]$ follows from the fact that $\text{ME}[p](x)$ diverges whenever $p(x \downarrow \ell)$ diverges for any ℓ, and so if $\text{ME}[p](x)$ is defined, then so is $\text{ME}[p](x \downarrow \ell)$.

Theorem 1. *ME is MT-secure.*

Proof. To show noninterference, pick any two x, y, and label ℓ such that $x \sim_\ell y$ and assume that $\text{ME}[p](x)^{\checkmark}$ and $\text{ME}[p](y)^{\checkmark}$. For all $\jmath \sqsubseteq \ell$ we have that $x \sim_\jmath y$ and so:

$$\begin{array}{ll}
\text{ME}[p](x) & @\jmath \\
= \bigcup\{\ p(x \downarrow \ell')@\ell' \mid \ell' \in \mathcal{L}\}@\jmath & \\
= p(x \downarrow \jmath) & @\jmath \\
= p(y \downarrow \jmath) & @\jmath \\
= \bigcup\{\ p(y \downarrow \ell')@\ell' \mid \ell' \in \mathcal{L}\}@\jmath & \\
= \text{ME}[p](y) & @\jmath
\end{array}$$

Because $x \downarrow \ell = \bigcup\{x@\jmath \mid \jmath \sqsubseteq \ell\}$ we now have that $\text{ME}[p](x) \sim_\ell \text{ME}[p](y)$.

For MT-termination, if $\text{ME}[p](x)^{\checkmark}$ then $p(x \downarrow \ell)^{\checkmark}$ for all ℓ and so $\text{ME}[p](x \downarrow \ell)^{\checkmark}$ for all ℓ. Consequently, ME is MT-secure. □

Naturally, a corollary of this theorem is that ME is also TI-secure.

Corollary 1. *ME is TI-secure.*

Next we address MT-transparency. Preservation of termination follows immediately from the definition of the MT termination criteria. This is a feature of the definition of MT termination, it is precisely the definition we need in order to prove MT-transparency. While the observation that conditions such as MT are natural pre-requisites for transparency (see e.g. [21, 26]), we believe that this framework provides a clean explanation for why MT exists. Because ME runs a number of $p(x \downarrow \ell)$ for different ℓ, it is natural to require that these runs terminate if $p(x)$ terminates in order to preserve termination. We note that MT-security is more useful as a correctness criteria for ME than as a notion of security. In other words, being MT-transparent is more useful than being MT-secure.

Theorem 2. *ME is MT-transparent.*

Proof. Pick any MT-secure program p and input x. We first prove that $\text{ME}[p](x)^{\checkmark} \Leftrightarrow p(x)^{\checkmark}$. If $p(x)^{\times}$ then because $x =$

		Transparency			
		Total	**TS**	**MT**	**TI**
Security	**TI**	✓	✓[16]	✓[17]	MEST
	MT	✓	✓	ME	
	TS	× ([25])	× [25]		
	Total	×			

TABLE II: Possible and impossible combinations of security and transparency.

$x\downarrow\bigsqcup\mathcal{L}(x)$ we have that $p(x\downarrow\bigsqcup\mathcal{L}(x))^{\times}$ and so $\text{ME}[p](x)^{\times}$. If $p(x)^{\checkmark}$ then $p(x\downarrow\ell)^{\checkmark}$ for all ℓ, and so clearly $\text{ME}[p](x)^{\checkmark}$.

Next we show that when $\text{ME}[p](x)^{\checkmark}$ and $p(x)^{\checkmark}$ we have that $\text{ME}[p](x) = p(x)$. For all $a^{\ell} \in p(x)$ is equivalent to $a^{\ell} \in p(x\downarrow\ell)@\ell$ (by p MT-secure), which in turn is equivalent to $a^{\ell} \in \text{ME}[p](x)$ (by definition). Consequently, $p(x) = \text{ME}[p](x)$. □

Again, a simple corollary follows from the hierarchy of notions of security.

Corollary 2. *ME is TS- and Total-transparent.*

Note that MT-secure and MT-transparent is the strongest classification of the security and transparency of the ME enforcement mechanism. For example, ME could not be MT-secure and TI-transparent, as this would require preserving TI-termination, which would in turn prohibit ME from being MT-secure. Likewise, ME could not be TS-secure, as this would prohibit it from preserving the semantics of merely MT-secure programs. In general, ME could not be τ-τ' for any τ stronger than, or τ' weaker, than MT.

In the following few sections, we establish a number of results about the possibility and impossibility of secure, transparent, and efficient information flow control. Before doing so, we review the state of transparent enforcement of noninterference in the literature.

The State of Transparent Enforcement

Table II presents the combinations of secure and transparent enforcement that are possible or impossible to enforce. The table is upper-triangular as anything below the diagonal is impossible. For example, it is impossible to simultaneously enforce a strong property such as TS-security, while preserving the semantics of all programs that obey a weaker property, such as TI-security. Put differently, because some programs are TI-secure but not TS-secure, these programs must have their semantics altered by a TS-secure enforcement mechanism.

As shown above, it is possible to construct an MT-MT enforcement mechanism ME. Because possibility propagates "up and to the left" in our table, any MT-MT enforcement mechanism is also TI-MT and TI-TS, for example, we also mark all boxes to the left and above of MT-MT as possible.

Ngo et al. [25] show that TS-TS enforcement is impossible. Their proof also works to show that the weaker TS-Total condition is impossible (we mark this by putting the citation in brackets in Table II), and consequently Total-Total enforcement is also impossible.

One open question in the literature is whether or not TI-TI enforcement is possible. In Section IV we answer this question by presenting MEST, a TI-TI enforcement mechanism.

One issue faced by ME is that a naive implementation fails to converge when $\mathcal{L}$ is non-finite, and takes infeasibly long to execute when $\mathcal{L}$ is very large. Running p once for each label in an infinite lattice is, clearly, impossible. In Section III we construct MEF, a version of ME which lifts these restrictions and makes ME work for non-finite lattices.

III. Multi-Execution for Decentralised Lattices

In this section, we instantiate the framework of Section II in the form of a novel formulation of the ME enforcement mechanism. Our new enforcement mechanism is inspired by MF [7], FSME [14], and OGMF [15]. Unlike SME, these schemes do not run one parallel version of the program for each level in the lattice, instead multi-execution under these schemes is determined by the behaviour of the program and the levels in the input. These methods are "adaptively" multi-executing in a "white-box" manner. This allows them to avoid running one copy of p for each security level and thereby also allows them to work for infinite $\mathcal{L}$, something which SME is unable to do.

We develop an enforcement mechanism MEF to provide a black-box account of these methods. MEF is, to the best of our knowledge, the first black-box enforcement mechanism that is MT-secure, MT-transparent, and works for non-finite (so called "decentralised" [31]) lattices.

We start by observing that the definition of ME requires running p on $x\downarrow\ell$ for each $\ell \in \mathcal{L}$. Assuming x is finite, then there is only a finite number of such downward projections of x, even for non-finite $\mathcal{L}$. Thus, regardless of the size of $\mathcal{L}$ we only need to run p a finite number of times. The tricky part is to appropriately compose the result of these runs to produce the right output.

To accomplish this, we need to introduce some additional machinery. If S is a finite subset of $\mathcal{L}$ we call

$$C(S) \triangleq \{ \bigsqcup S' \mid S' \subseteq S \}$$

the *closure-set of* S. Note that $C(S)$ simply encodes all the ways of combining zero or more labels in S, including both the lower bound of $\bot$ and the upper bound of $\bigsqcup S$. For example, if $S = \mathcal{L}(x)$ for some x, then $C(S)$ is all the ways to label data arising by combining zero or more $a^{\ell} \in x$.

For any set $S \subseteq \mathcal{L}$ and label $\ell \in S$ we define the *upward neighbourhood of* ℓ *in* S, written $\ell \uparrow S$, as:

$$\ell \uparrow S \triangleq \{ \jmath \in \mathcal{L} \mid \ell \sqsubseteq \jmath, \forall \jmath' \in S.\ \jmath' \sqsubseteq \jmath \Rightarrow \jmath' \sqsubseteq \ell \}$$

That is, $\ell \uparrow S$ is the set of all $\jmath \in \mathcal{L}$ such that if $\ell \sqsubseteq \jmath$, then there is no other label in S which can flow to $\jmath$ but can not flow to ℓ. Put differently, if $\jmath \in \ell \uparrow S$, then ℓ is the greatest label in S such that $\ell \sqsubseteq \jmath$. From the point of view of information flow, this means that ℓ captures all the labels in s whose information may flow to $\jmath$.

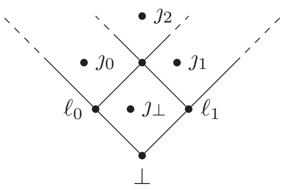

Fig. 2: The upwards neighbourhoods of $C(\{\ell_0, \ell_1\})$

For an illustration of how these upwards neighbourhoods work, see Figure 2. Each area in the diagram corresponds to the upwards neigbourhood of one of the levels in $C(\{\ell_0, \ell_1\})$. The level $\jmath_\perp$ in the middle square, for example, is in the upward neighbourhood of level $\perp$ in $C(\{\ell_0, \ell_1\})$ while $\jmath_0$ is in the upward neighbourhood of ℓ_0. Likewise, $\jmath_1$ is in $\ell_1 \uparrow C(\{\ell_0, \ell_1\})$ and $\jmath_2$ is in $\ell_0 \sqcup \ell_1 \uparrow C(\{\ell_0, \ell_1\})$.

Note that in the case where $S = \{\ell_0, \ell_1\}$ where ℓ_0 and ℓ_1 are incomparable, neither $\ell_0 \uparrow S$ nor $\ell_1 \uparrow S$ contain $\ell_0 \sqcup \ell_1$. For this reason, it is important to always consider upwards neighbourhoods in $C(S)$ rather than S itself. The upward neighbourhood of ℓ in $C(S)$ precisely characterises the levels for which the computation $p(x \downarrow \ell)$ can produce output if there is one $p(x \downarrow \ell)$ for each $\ell \in C(S)$. This is captured by the following proposition together with the fact that ℓ can flow to all levels in $\ell \uparrow C(S)$.

Proposition 4. *Given any $S \subseteq \mathcal{L}$, the family which maps each $\ell \in C(S)$ to $\ell \uparrow C(S)$ partitions $\mathcal{L}$.*

Proof. See Appendix A. □

For the next definition we overload the @ operator to work for sets $S \subseteq \mathcal{L}$ of labels as $x@S = \{ a^\ell \mid a^\ell \in x, \ell \in S \}$. Finally, we define a more sophisticated version of ME that we call MEF (for ME-Fastish).

Definition 7.

$$\text{MEF}[p](x) \triangleq \bigcup \{ p(x \downarrow \ell) @ (\ell \uparrow C(\mathcal{L}(x))) \mid \ell \in C(\mathcal{L}(x)) \}$$

■

Intuitively, this definition computes all the possible combinations of levels which can arise from computing with data in the input (i.e. $C(\mathcal{L}(x))$) and performs multi-execution with only these levels. However, this is not sufficient to capture all the levels the data may be output at. In effect, this is because a computation may combine data labeled ℓ_0 and ℓ_1, but write the output at some level $\ell_2 \sqsupset \ell_0 \sqcup \ell_1$. To get around this issue and ensure that we capture *all* outputs of a computation, we finally do the selection $p(x \downarrow \ell)@(\ell \uparrow C(\mathcal{L}(x)))$. Because the upward neighbourhoods of $C(\mathcal{L}(x))$ are disjoint, the final output of MEF$[p](x)$ is a union of disjoint sets, so each element of the output is added to the final set only once. To see this principle in action, consider the following example.

Example 5. Recall the definition of *combine* from Example 1:

$$\textit{combine}(x) \triangleq \{(|x@\text{Alice}| + |x@\text{Bob}|)^{\text{Alice}\sqcup\text{Bob}\sqcup\text{Charlie}}\}$$

MEF[*combine*]$(\{1^{\text{Alice}}\})$ runs one opy of *combine* for each level in $C(\{\text{Alice}\}) = \{\perp, \text{Alice}\}$. This means that MEF runs *combine*$(\{1^{\text{Alice}}\} \downarrow \perp)$ and *combine*$(\{1^{\text{Alice}}\} \downarrow \text{Alice})$. The first run outputs $\{0^{\text{Alice}\sqcup\text{Bob}\sqcup\text{Charlie}}\}$ and the second one $\{1^{\text{Alice}\sqcup\text{Bob}\sqcup\text{Charlie}}\}$. Because Alice $\sqcup$ Bob $\sqcup$ Charlie is in the upwards closed neighbourhood of Alice in $\{\text{Alice}, \perp\}$ and not in the upwards closed neighbourhood of $\perp$, we have that the output of MEF[*combine*]$(\{1^{\text{Alice}}\})$ is $\{1^{\text{Alice}\sqcup\text{Bob}\sqcup\text{Charlie}}\}$. Note that, had MEF been using the same rule for selecting outputs as ME, i.e. had the definition used $p(x \downarrow \ell)@\ell$ rather than $p(x \downarrow \ell)@(\ell \uparrow C(\mathcal{L}(x)))$, the output in our example would be $\emptyset$, as Alice $\sqcup$ Bob $\sqcup$ Charlie is equal to neither Alice nor $\perp$. ■

While the formulation of MEF looks similar to that of ME, it has the advantage of being implementable even for infinite, "decentralised", lattices like DC-labels [31]. It also has the nice property of being equivalent to ME (provided that both terminate).

Theorem 3. *For all p, x, if $\mathcal{L}$ is finite we have that:*

$$\textit{MEF}[p]x = \textit{ME}[p]x$$

Proof. See Appendix A. □

The theorem above means that MEF inherits the MT-MT property of ME in the case when $\mathcal{L}$ is finite. In the case when $\mathcal{L}$ is non-finite, we still have that MEF is MT-MT. MEF$[p]$ is MT-terminating because $C(\mathcal{L}(x \downarrow \ell)) \subseteq C(\mathcal{L}(x))$ and if p is MT-secure then termination and output behaviour are preserved as well. The proof of MT-transparency works by establishing that MEF$[p](x)$ implements ME$[p](x)$ when read as a (non-runnable) specification for infinite lattices and so we relegate the proof to Appendix A.

Proposition 5. *MEF is MT-MT.*

IV. Termination Insensitive Noninterference is Transparently Enforceable

In this section we answer an open question in the literature [30] by constructing a TI-TI enforcement mechanism. We start by briefly exploring why ME and MEF are not TI-transparent. The issue lies in preserving the termination of TI-secure programs. Put simply, if p is a TI-secure program and $p(x)$ is defined, there is no guarantee that $p(x \downarrow \ell)$ is defined and so there is no guarantee that ME$[p](x)$ is defined.

Overcoming this limitation of ME boils down to two observations. The first observation is that ME fails to be TI-transparent only because of termination, which boils down to the choice to use $x \downarrow \ell$ as the input to the run of p at level ℓ. Because there is no guarantee that $p(x)^\checkmark$ implies $p(x \downarrow \ell)^\checkmark$, we can't produce the output from the run at ℓ, even though $p(x)$ is defined. If we can choose some other $x' \sim_\ell x$ such that $p(x')^\checkmark$ instead, we may be able to get something done. The second observation is that if $p(x)^{\checkmark}$, then for each ℓ there exists at least one x' such that $x' \sim_\ell x\downarrow\ell$ and $p(x')^{\checkmark}$, namely x. We can unfortunately not simply run $p(x)$ instead of $p(x\downarrow\ell)$ however, as this may not be secure. Instead, we need to find x' such that $x' \sim_\ell x$ and $p(x')^{\checkmark}$ based only on $x\downarrow\ell$, ℓ, and p.

The question now becomes, how do we pick such an x'? One answer is to enumerate all $x' \sim_\ell (x\downarrow\ell)$ in a deterministic order and dove-tail, that is run "in parallel", all $p(x')$ and pick the first x' such that $p(x')$ terminates. Note that this does not mean we pick the first x' *in the enumeration order*, rather we pick the x' such that, during dove-tailing, $p(x')$ is the first run to terminate. In the rest of this section we formalise this simple idea and show that by using this one trick, we obtain a TI-TI enforcement mechanism.

If $p : \mathcal{P}(A\times\mathcal{L}) \to_p \mathcal{P}(B\times\mathcal{L})$ is a program, ℓ a level, x an input to p, and $f : \mathbb{N} \to \mathcal{P}(A\times\mathcal{L})$ an enumeration of $\mathcal{P}(A\times\mathcal{L})$ then the following function is partial recursive:

$$\psi^\ell_{p,x}(i) \triangleq \begin{cases} f(i) \text{ if } f(i) \sim_\ell x \text{ and } p(f(i))^{\checkmark} \\ \text{divergent otherwise} \end{cases}$$

This means that, from Corollary 5.V(a) in [32], as long as there exists an $x' \sim_\ell x$ such that $p(x')^{\checkmark}$, then there exists a total recursive function $\phi^\ell_{p,x}$ with the same range as $\psi^\ell_{p,x}$. This is where the dove-tailing happens, $\phi^\ell_{p,x}$ more or less works by interleaving the executions of $p(f(i))$ for all i until one terminates. There are two facts about ϕ we use when constructing our TI-TI enforcement mechanism:

1) If $x \sim_\ell y$ then $\phi^\ell_{p,x\downarrow\ell}(i) = \phi^\ell_{p,y\downarrow\ell}(i)$.
2) If p is TI-secure, then $p(\phi^\ell_{p,x\downarrow\ell}(i)) \sim_\ell p(x)$ provided both runs terminate.

From these two observations we define our TI-TI enforcement mechanism MEST for "multi-execution search for termination".

Definition 8. Define the enforcement mechanism MEST as:

$$\mathrm{MEST}[p](x) \triangleq \begin{cases} \text{divergent if } p(x)^{\times} \\ \cup\{\ p(\phi^\ell_{p,x\downarrow\ell}(0))@(\ell\uparrow C(\mathcal{L}(x))) \\ \quad \mid \ell \in C(\mathcal{L}(x))\} \text{ otherwise} \end{cases}$$

■

To see how MEST works, consider again the program *divergeIfHPresent* from Example 3.

$$\textit{divergeIfHPresent}(x) \triangleq \text{if } 1^H \in x \text{ then diverge else } \emptyset$$

This program is TI-secure, and so we expect that MEST should preserve its semantics. Running MEST[*divergeIfHPresent*]$(\{1^H\})$ diverges, as *divergeIfHPresent*$(\{1^H\})^{\times}$, and assuming without loss of generality that $\phi^\ell_{\textit{divergeIfHPresent},\emptyset}(0) = \emptyset$ for all ℓ, running MEST[*divergeIfHPresent*]$(\emptyset)$ yields the expected result of $\emptyset$.

The proof of security for MEST relies on the first observation above, and the proof of transparency relies on the second.

Theorem 4. *MEST is TI-secure*

Proof. To see that MEST is TI-secure, consider any level ℓ, program p, and inputs $x \sim_\ell y$. It is the case that for all $\ell' \sqsubseteq \ell$ the greatest $\jmath$ in $C(\mathcal{L}(x))$ and $C(\mathcal{L}(y))$ such that $\jmath \sqsubseteq \ell'$ are the same, as $\mathcal{L}(x\downarrow\ell') = \mathcal{L}(y\downarrow\ell')$. If both MEST$[p](x)^{\checkmark}$ and MEST$[p](y)^{\checkmark}$ then:

$$\begin{aligned} &\mathrm{MEST}[p](x)\ @\ell' \\ &= p(\phi^\jmath_{p,x\downarrow\jmath}(0))@\ell' \\ &= p(\phi^\jmath_{p,y\downarrow\jmath}(0))@\ell' \\ &= \mathrm{MEST}[p](y)\ @\ell' \end{aligned}$$

From which we conclude that MEST$[p](x) \sim_\ell$ MEST$[p](y)$, in other words MEST is TI-secure. □

Theorem 5. *MEST is TI-transparent.*

Proof. Note that $p(x)^{\checkmark} \Leftrightarrow$ MEST$[p](x)^{\checkmark}$. If p is TI-secure then it holds that for all ℓ and $x \sim_\ell y$, $p(y)@\ell = p(x)@\ell$. Consider now the greatest $\jmath \in C(\mathcal{L}(x))$ such that $\jmath \sqsubseteq \ell$, if $y \sim_\jmath x$ then $y \sim_\ell x$ and so $\phi^\jmath_{p,x\downarrow\jmath}(0) \sim_\ell x$ and so, for all ℓ:

$$\mathrm{MEST}[p](x)@\ell = p(\phi^\jmath_{p,x\downarrow\jmath}(0))@\ell = p(x)@\ell$$

Which gives us that MEST$[p](x) = p(x)$. □

While MEST answers the question of whether or not TI-TI enforcement is possible, it does so in a somewhat unsatisfactory manner. Computing MEST$[p](x)$ is very slow, not only because we are computing something once for each ℓ in $C(\mathcal{L}(x))$ (which is an exponential number of levels!), but also because the computation of $\phi^\ell_{p,x\downarrow\ell}(0)$ requires interleaving several (possibly an exponential number!) of runs of p.

V. Transparent Enforcement is Multi-Execution

Consider again the program *combineAll* from Example 1.

$$\textit{combineAll}(x) \triangleq \{|x|^{\bigsqcup \mathcal{L}(x)}\}$$

Running MEF[*combineAll*](x) results in one run of *combineAll* for each level in $C(\mathcal{L}(x))$ and thus, MEF[*combineAll*](x) produces one output for each level in $C(\mathcal{L}(x))$. In the worst case, the size of the output of MEF[*combineAll*](x) is exponential in the size of x whereas the original output of p is polynomial sized in the size of x.

This observation that ME and MEF cause exponential overhead in the output of some programs causes severe issue for multi-execution. Many applications "in the wild" have large legacy code bases with vulnerabilities and would therefore benefit from applying transparent IFC techniques. If these techniques introduce extraneous overhead, applying them to large legacy systems is out of the question.

As insecure programs cause ME and MEF to produce exponentially sized outputs, we know there can be no *semantics preserving* optimisation of these methods that gets rid of this overhead. This means that the work on OGMF [15] and some of the "data-oriented" optimisations of Algehed et al. [16], can not hope to make multi-execution practical *on their own*. Such techniques may be integral to the eventual success of multi-execution, but not without incorporating other measures as well.

There is still hope that we may find some efficient enforcement mechanism, but this mechanism may need to behave differently to ME on programs such as *combineAll* above. Insecure programs aside, we begin by defining the least restrictive notion of an efficient enforcement mechanism, that it "only" produce polynomial overhead on already secure programs.

Definition 9. An enforcement mechanism E is τ-efficient if and only if $E[p](x)$ runs in time polynomial in $|p| + |x|$ whenever p is τ-secure and runs in time polynomial in $|x|$. ■

In this section we consider the connection between arbitrary enforcement mechanisms and multi-execution and show that any enforcement mechanism for TI-security and MT-transparency gives rise to an efficient strategy for multi-execution. In spirit, it is similar to MEF, we do multi-execution only for the levels that are necessary. We begin by generalising the definition of ME to allow it to selectively multi-execute at particular levels.

Definition 10. A partial function $\mathbb{L}$ is a *Level Assignment* if given a program $p : \mathcal{P}(A \times \mathcal{L}) \to_p \mathcal{P}(B \times \mathcal{L})$ and an input $x \in \mathcal{P}(A \times \mathcal{L})$ we have $\mathbb{L}(p, x) \subseteq \mathcal{L}$. ■

Given a level assignment $\mathbb{L}$, we define the enforcement mechanism $\text{ME}_{\mathbb{L}}$:

$$\text{ME}_{\mathbb{L}}[p](x) \triangleq \bigcup \{\ p(x{\downarrow}\ell)@\ell \mid \ell \in \mathbb{L}(p, x)\ \}$$

Note that $\text{ME}_{\mathbb{L}}$ is simply a generalisation of ME. Level assignments naturally inherit definitions from enforcement mechanisms.

Definition 11. We call a level assignment $\mathbb{L}$ τ-secure if $\text{ME}_{\mathbb{L}}$ is τ-secure, and τ-transparent if $\text{ME}_{\mathbb{L}}$ is τ-transparent. Furthermore, we call $\mathbb{L}$ τ-efficient if for all polynomial-time τ-secure programs $\mathbb{L}(p, x)$ takes time polynomial in $|p|+|x|$. ■

Note that, if $\mathbb{L}$ takes time polynomial in $|p| + |x|$ and p takes time polynomial in $|x|$, then $\text{ME}_{\mathbb{L}}[p](x)$ also takes time polynomial in $|p| + |x|$. This is because if $\mathbb{L}$ is polynomial time, then it only produces a polynomially sized output and so $\text{ME}_{\mathbb{L}}$ only executes a polynomial number of runs of the polynomial time program p.

Example 6. Deciding if a level assignment is secure and transparent can be non-trivial. Consider the intuitive level assignment $\mathbb{L}(p, x) = \mathcal{L}(p(x))$. It allows us to compute levels without doing anything other than computing $p(x)$. Unfortunately, however, it is insecure in the $H - L$ lattice, as illustrated by the following program:

$$\textit{leakLevel}(x) \triangleq \text{if } x@H = \emptyset \text{ then } \{0^L\} \text{ else } \emptyset$$

We have that $\emptyset \sim_L \{0^H\}$, but the following holds:

$$\begin{array}{r|r} \textit{leakLevel}(\emptyset) = \{0^L\} & \textit{leakLevel}(\{0^H\}) = \emptyset \\ \mathbb{L}(\textit{leakLevel}, \emptyset) = \{L\} & \mathbb{L}(\textit{leakLevel}, \{0^H\}) = \emptyset \\ \text{ME}_L[\textit{leakLevel}](\emptyset) = \{0^L\} & \text{ME}_L[\textit{leakLevel}](\{0^H\}) = \emptyset \end{array}$$

Which gives us:

$$\text{ME}_{\mathbb{L}}[\textit{leakLevel}](\emptyset) = \{0^L\} \not\sim_H \emptyset = \text{ME}_{\mathbb{L}}[\textit{leakLevel}](\{0^H\})$$

In conclusion, $\text{ME}_{\mathbb{L}}[\textit{leakLevel}]$ is not secure and so neither is $\mathbb{L}$. ■

Clearly, there are some conditions which need to be met in order to produce a secure level assignment. The following lemma establishes one sufficient condition for TI-security, that $\mathbb{L}(p, \_)$ is a secure program. To make this precise, we observe that the notion of projection can be extended to sets of labels by observing that $\mathcal{P}(\mathcal{L})$ is isomorphic to $\mathcal{P}(1 \times \mathcal{L})$ where 1 is the single-element set. Consequently, we extend the definition of projection (and thereby also ℓ-equivalence) for subsets $S \subseteq \mathcal{L}$ as $S{\downarrow}\ell = \{\ \ell' \in L \mid \ell' \sqsubseteq \ell\ \}$.

Lemma 1. *If the program $\lambda x.\ \mathbb{L}(p, x)$ is TI-secure for all p, then $\mathbb{L}$ is TI-secure.*

Proof. See Appendix A. □

This lemma raises two immediate questions:

1) Is the implication strict?
2) Why TI and not MT-secure?

The answer to the first question is that the implication is indeed strict, as demonstrated by the following example.

Example 7. Provided that $\mathcal{L}$ has at least one level ℓ such that $\ell \not\sqsubseteq \bot$, there exists an MT-secure $\mathbb{L}$ for which $\mathbb{L}(p, \_)$ is not secure for all p. To see this, consider the program:

$$\textit{empty}(x) \triangleq \emptyset$$

and define $\mathbb{L}$ to be:

$$\mathbb{L}(q, x) \triangleq \begin{cases} \emptyset & \text{if } \ell \in \mathcal{L}(x) \text{ and } q = \textit{empty} \\ \{\bot\} & \text{if } \ell \notin \mathcal{L}(x) \text{ and } q = \textit{empty} \\ \emptyset & \text{otherwise} \end{cases}$$

In the case where $q = \textit{empty}$ we have that:

$$\text{ME}_{\mathbb{L}}[q](x) = \emptyset = \text{ME}_{\mathbb{L}}[q](y)$$

giving us that $\text{ME}_{\mathbb{L}}[q]$ is noninterfering. Furthermore, in the case where $q \neq \textit{empty}$, we have that $\text{ME}_{\mathbb{L}}[q](x) = \emptyset$, which is also MT-secure. ■

To see the answer to the second question, consider the next example.

Example 8. Consider again the two-point lattice $L \sqsubseteq H$. The level assignment $\mathbb{L}(p, x) \triangleq \{\ H \mid H \notin \mathcal{L}(x)\}$ satisfies the property that $\lambda x.\ \mathbb{L}(p, x)$ is an MT-secure program for all p. Recall the program *divergeIfHAbsent*:

$$\textit{divergeIfHAbsent}(x) \triangleq \text{if } 1^H \notin x \text{ then diverge else } \emptyset$$

Notice that $\textit{divergeIfHAbsent}(\emptyset)^\times$ and therefore we have that:

$$\mathbb{L}(\textit{divergeIfHAbsent}, \emptyset) = \{H\}$$
$$\text{ME}_{\mathbb{L}}[\textit{divergeIfHAbsent}](\emptyset)^\times$$

Furthermore, as a general fact we have that if $\mathbb{L}(p, x) = \emptyset$ then $\mathrm{ME}_{\mathbb{L}}[p](x) = \emptyset$ for all $\mathbb{L}$, p, and x. Therefore, the following holds:

$$\mathbb{L}(\mathit{divergeIfHAbsent}, \{1^H\}) = \emptyset$$
$$\mathrm{ME}_{\mathbb{L}}[\mathit{divergeIfHAbsent}](\{1^H\}) = \emptyset$$

However, because $\emptyset = \{1^H\}\downarrow L$, we see that $\mathrm{ME}_{\mathbb{L}}[\mathit{divergeIfHAbsent}]$ is not an MT-secure program, as it is not MT-terminating; $\mathrm{ME}_{\mathbb{L}}[\mathit{divergeIfHAbsent}](\{1^H\})$ is defined while $\mathrm{ME}_{\mathbb{L}}[\mathit{divergeIfHAbsent}](\{1^H\}\downarrow L)$ is not. ■

Lemma 1 suggests the following class of level assignment techniques that are secure by construction.

Proposition 6. *If $\mathbb{L}(p, x) = C(\mathcal{L}(x)) \cap \mathbb{L}'(p)$ for some total $\mathbb{L}'$, then $\mathbb{L}$ is an MT-secure level assignment.*

Proof. See Appendix A. □

For programs p that satisfy the property that for all x, $\mathcal{L}(p(x)) \subseteq C(\mathcal{L}(x))$, such that $\mathbb{L}'(p) \supseteq \mathrm{domain}(\mathcal{L} \circ p)$ the level assignment $\mathbb{L}$ above is MT-transparent (see Lemma 2 below). This explains the "computation-oriented" optimisations of Algehed et al. [16]. In their paper programs only produce outputs at levels in $C(\mathcal{L}(x))$ and they attach so called "label-expressions", booleans expressions over lattice elements, to program points to restrict the levels for which multi-execution is carried out. In one of their examples, program p takes input from multiple different inputs, but only ever writes to channels labeled with single principals like Alice or Bob. The authors then attach label-expressions on the form Alice$\wedge\neg$Bob$\vee\neg$Alice$\wedge$Bob, which represents the set of labels $\{\{\text{Alice}\}, \{\text{Bob}\}\}$ to the program p during multi-execution. Note that the sets of labels attached to a program in this scheme is constant, it does not depend on the input x and so $\mathbb{L}(p, x)$ is on the form required by Proposition 6.

With an appropriate sufficient condition for security established, the natural next question is if a similar sufficient condition exists for transparency.

Lemma 2. *$\mathbb{L}$ is MT-transparent if for all MT-secure programs p and inputs x, we have that $p(x)^{\checkmark}$ if and only if $\mathbb{L}(p, x)^{\checkmark}$ and $\mathbb{L}(p, x) = \mathcal{L}(p(x))$.*

Proof. See Appendix A. □

Next we develop one of the key contributions of this paper. We use the theory of level assignments to show that any TI-secure and MT-transparent enforcement mechanism E gives rise to a TI-secure and MT-transparent level assignment $\mathbb{L}_E$. Furthermore, it follows trivially that multi-executing a secure, polynomial time, programs p using $\mathbb{L}_E$ is only polynomially slower than executing $E[p]$. Concretely, what this demonstrates is that any efficient solution to the problem of MT-transparent enforcement gives rise to an efficient solution to the problem of computing level assignments.

Definition 12. Given an enforcement mechanism E, define the level assignment $\mathbb{L}_E(p, x) \triangleq \mathcal{L}(E[p](x))$. ■

Theorem 6 (TI-Security)**.** *If E is TI-secure, then $\mathbb{L}_E$ is a TI-secure level assignment.*

Proof. Immediate by Lemma 1. □

Theorem 7 (MT-Transparency)**.** *If E is MT-transparent, then so is $\mathbb{L}_E$.*

Proof. Immediate by Lemma 2. □

Proposition 7. *If $E[p](x)$ runs in time polynomial in $|p| + |x|$ for all polynomial time MT-secure programs p and inputs x, then $\mathbb{L}_E$ is MT-efficient.*

This polynomial-time correspondence between multi-execution and any transparent enforcement mechanism means that any limit on the efficiency of multi-execution is a limit on the efficiency of transparent enforcement. This is a sobering thought, to the best of our knowledge there is no efficient way to do multi-execution. In fact, it is not clear that there should be any efficient way to do it. In a sense, the level assignment problem asks that we produce a program $\mathbb{L}$ that is capable of efficiently answering arbitrarily tricky questions about program executions.

VI. Efficient Black-Box Enforcement is Impossible for Decentralised Lattices

As seen empirically in previous work, transparent enforcement suffers from exponential overheads [7, 14, 16, 22]. In this section we argue that there may be fundamental reasons why this is the case by showing that it is impossible to do efficient, i.e. polynomial overhead, black-box enforcement. Intuitively, an enforcement mechanism E is black-box if $E[p](x)$ can only do a series of tests of $p : \mathcal{P}(A\times\mathcal{L}) \to_p \mathcal{P}(B\times\mathcal{L})$, where each test consists of a test-input $x_t \in \mathcal{P}(A\times\mathcal{L})$ and a "continuation" which takes the result of $p(x_t)$ and either returns a final result $E[p](x) \in \mathcal{P}(B\times\mathcal{L})$ or continues testing. Thus, we define the set of *test sequences* T from A to B inductively as:

$$T ::= \mathcal{P}(B \times \mathcal{L}) \mid \mathcal{P}(A \times \mathcal{L}) \times (\mathcal{P}(B \times \mathcal{L}) \to_p T)$$

A test sequence $t \in T$ is either finished, $t \in \mathcal{P}(B \times \mathcal{L})$, or it is a test $a \in \mathcal{P}(A \times \mathcal{L})$ together with a continuation $c \in \mathcal{P}(B \times \mathcal{L}) \to_p T$.

For example, MEF can be understood as producing a test sequence where it tests all $x\downarrow\ell$ for ℓ in $C(\mathcal{L}(x))$, keeps the relevant parts of the output around, and constructs the final union when it produces its finished result.

We define the program $\mathrm{Exec} : \mathrm{Program} \times T \to_p \mathcal{P}(B \times \mathcal{L})$, which takes a program p and a test sequence t and produces the result of evaluating t with p.

$$\begin{aligned}\mathrm{Exec}(p, (y, c)) &\triangleq \mathrm{Exec}(p, c(p(y)))\\ \mathrm{Exec}(p, z) &\triangleq z\end{aligned}$$

If $e : \mathcal{P}(A \times \mathcal{L}) \to_p T$ takes an input and produces a test sequence, then we can execute e by running the following program $\mathrm{Exec}(p, e(x))$: If $\mathrm{Exec}(p, e(x))$ terminates, we can obtain the *trace* of tested inputs and the corresponding output of p (i.e. a list of $\mathcal{P}(A \times \mathcal{L}) \times \mathcal{P}(B \times \mathcal{L})$) by computing $\text{Trace}(p, e(x))$:

$$\begin{aligned}\text{Trace}(p, (y, c)) &\triangleq (y, p(y)) : \text{Trace}(p, c(p(y))) \\ \text{Trace}(p, z) &\triangleq []\end{aligned}$$

Because we are interested in entirely black-box methods, we require that the enforcement mechanism E should work for any choice of lattice. One consequence of this choice is that E must *discover* elements of $\mathcal{L}$. We can formalise the idea that E must discover its labels by defining a trace to be consistent with an initially known set of labels S inductively as:

$$\frac{}{\text{Consistent}(S, [])} \qquad \frac{\mathcal{L}(y) \subseteq C(S) \quad \text{Consistent}(S \cup \mathcal{L}(z), t)}{\text{Consistent}(S, (y, z) : t)}$$

With these definitions in place, we can define what it means for an enforcement mechanism to be black-box:

Definition 13. E is a *black-box* enforcement mechanism if and only if there exists a function $e : \mathcal{P}(A \times \mathcal{L}) \to_p T$ such that:

$$E[p](x) = \text{Exec}(p, e(x))$$

We call $\text{Trace}(p, e(x))$ the trace of E at p, x and require that $\text{Consistent}(\emptyset, \text{Trace}(p, e(x)))$ always holds. ■

We refer to set of test inputs in the trace t produced by E at p and x, i.e. the set of first elements of the tuples in t, as the *test-set* of E. Note that our definition pre-supposes that E initially does not know *any* of the labels in $\mathcal{L}$. This restriction can be lifted to allow E to have prior knowledge of a finite proper subset of $\mathcal{L}$ and all the results in this section still hold.

The intuition for the proof that efficient black-box enforcement does not exist is that all an efficient black-box enforcement mechanism E can do is run the program p a polynomial number of times. This means that there is always a large number of subsets of the input x which E has not tried p on. We exploit this fact by constructing, from E, two secure programs p and q and an insecure program w. We then pick two ℓ-equivalent inputs such that w behaves like p on one input and q on the other. Because E is black-box, it can not tell the two runs of w apart from the runs of p and q, and so E is forced to preserve the semantics of w, which is insecure.

Theorem 8. *There is no Total-efficient TI-Total black-box enforcement mechanism.*

Proof. Let $\mathcal{L} = \mathcal{P}(\mathbb{N})$, the powerset lattice over the natural numbers. We consider programs where the domain and codomain are both the singleton set $\{1\}$. In other words, we consider programs $p : \mathcal{P}(\{1\} \times \mathcal{L}) \to_p \mathcal{P}(\{1\} \times \mathcal{L})$.

Assume E is a Total-efficient TI-Total black-box enforcement mechanism. For any finite subset S of $\mathbb{N}$ define:

$$\textit{toInput}(S) \triangleq \{1^{\{n\}} \mid n \in S\}$$

Define the Total-secure program $\textit{empty}(x) \triangleq \emptyset$. Let $X(S)$ be test-set of E for $\textit{empty}, \textit{toInput}(S)$. Because E is Total-efficient, we can fix a sufficiently large S such that there exists an $S' \subset S$ such that there is no $x \in X(S)$ that satisfies both $\textit{toInput}(S') \subseteq x$ and $\cup\mathcal{L}(x) = S'$, this is because the size of $X(S)$ is polynomial in $|S|$ but the number of choices for S' grows exponentially in $|S|$. Note that $\textit{toInput}(S) \sim_{S'} \textit{toInput}(S')$, as $\textit{toInput}(S) \downarrow S' = \textit{toInput}(S') = \textit{toInput}(S') \downarrow S'$.

Now define another Total-secure program:

$$\textit{greater}(x) \triangleq \text{if } \textit{toInput}(S') \subseteq x \text{ then } \{1^{S'}\} \text{ else } \emptyset$$

and the TI-insecure program:

$$\textit{equal}(x) \triangleq \text{if } \textit{toInput}(S') \subseteq x \wedge \cup\mathcal{L}(x) = S' \text{ then } \{1^{S'}\} \text{ else } \emptyset$$

Next consider two runs of $E[\textit{equal}]$. The first is $E[\textit{equal}](\textit{toInput}(S))$. Recall that our choice of S' guarantees that E never tests *empty* on any x such that $\textit{toInput}(S') \subseteq x$ and $\cup\mathcal{L}(x) = S'$. Furthermore, *equal* behaves like *empty* on all other inputs. Hence we know that:

$$E[\textit{equal}](\textit{toInput}(S)) = E[\textit{empty}](\textit{toInput}(S)) = \emptyset$$

The second run we consider is $E[\textit{equal}](\textit{toInput}(S'))$, for which *equal* behaves as *greater*. To see why, consider that $\textit{equal}(x) \neq \textit{greater}(x)$ implies that the two tests in *equal* and *greater* evaluate to different things on x. Hence, $\textit{toInput}(S') \subseteq x$ and $\cup\mathcal{L}(x) \neq S'$, and so x must contain labels not in S'. However, consistency means that E can not construct any such x when testing either *equal* or *greater* on $\textit{toInput}(S')$. As a consequence, when E tests *greater* it never does a test that would reveal the difference between *greater* and *equal*. This means that we can conclude that the following holds:

$$E[\textit{equal}](\textit{toInput}(S')) = E[\textit{greater}](\textit{toInput}(S')) = \{1^{S'}\}$$

In conclusion, we have that both of the following two things hold:

$$\textit{toInput}(S) \sim_{S'} \textit{toInput}(S')$$

$$E[\textit{equal}](\textit{toInput}(S)) = \emptyset \not\sim_{S'} \{1^{S'}\} = E[\textit{equal}](\textit{toInput}(S'))$$

Which contradictions our assumption that E is TI-secure. □

Because Theorem 8 proves the *easiest* τ-τ' enforcement to be impossible to do efficiently, a simple corollary is the following.

Corollary 3. *Efficient black-box TI-TI, TI-MT, TI-TS, MT-MT, MT-TS, and MT-Total enforcement are all impossible.*

VII. Future Work

There are a number of limitations to our model. Firstly, we do not consider reactive (like [13, 17]) or interactive (like [21]) programs. We expect that extending our model to cover this case will introduce new challenges related to termination and progress channels. For example, there is no reason to expect that reactive systems are *any easier* to secure than batch-job ones, as in the batch-job setting we have access to everything, including the program and all the inputs, from the start. Whereas in the reactive or interactive setting the enforcement mechanism needs to react to new information and act on partial knowledge of the input. Furthermore, in the interactive case, the fact that the attacker can dynamically choose inputs to respond to the action of the enforcement mechanism may pose additional issues for efficient transparent enforcement.

Likewise, our model does not yet deal with declassification. Extensional accounts of declassification in multi-execution already exist [21, 33], and we expect they can be adapted to our setting.

The final, and we believe most important, future work is to resolve the question of whether or not efficient transparent enforcement is at all possible even for our simple model. The connection between multi-execution and transparent enforcement established in Section V means that efficiency of enforcement and efficiency of multi-execution are intimately related. However, to the best of our knowledge multi-execution requires something akin to efficient program analysis, either static or dynamic, to break the exponential blowup barrier [16]. The inherent limitations posed by undecidability therefore leads us to conjecture that efficient transparent enforcement is impossible.

Conjecture 1. *Total-efficient TI-Total enforcement is impossible.*

This conjecture states the strongest possible variant of this theorem in our framework. TI-Totalenforcement is our weakest condition, which means that impossibility of efficient TI-Total enforcement implies impossibility of efficient τ-τ' enforcement for all τ and τ' in this paper. We believe that a proof of this conjecture could use Theorems 6 and 7 and Proposition 7 from Section V, which we formalise in the following lemma statement.

Lemma 3. *Unless there is a Total-efficient TI-Total level assignment $\mathbb{L}$ such that:*

1) *For all Total-secure p, $\mathcal{L}(p(x)) = \mathbb{L}(p, x)$*
2) *For all p, $\lambda x.\ \mathbb{L}(p, x)$ is a TI-secure program*

There is no Total-efficient TI-Total enforcement mechanism.

The lemma uses the $\mathbb{L}_E$ level-assignment from Section V. Condition (1) follows from the fact that $\mathcal{L}(E[p](x)) = \mathcal{L}(p(x))$ when E is Total-transparent and p Total-secure. Condition (2) meanwhile follows from E being TI-secure.

VIII. Related Work

There exists a large body of work on the expressive power and precision of different enforcement mechanisms for noninterference. Bielova and Rezk [30] study a number of different mechanisms and rank them in order of precision. Ngo et al. [25] show that TS-TS enforcement is impossible. Hamlen et al. also study the complexity classes of monitors and enforcement mechanisms [27].

Ngo et al. [34] provide both a generic black-box enforcement mechanism for reactive programs that is transparent, as well as lower and upper bounds on what hyperproperties can and cannot be black-box transparently enforced. While this work is impressive, it has two limitations. Firstly, in the interest of tractability the paper partly side-steps the issue of termination by demanding that programs always make progress. Secondly, the paper is only concerned with what can and cannot be enforced in a black-box manner. We believe that a natural and exciting direction for future work is to marry their formalism and ours to allow careful study of white-box transparency with non-termination in a reactive setting.

A number of authors have provided multi-execution based enforcement mechanism with subtly different τ-τ' properties [7, 13–18, 21, 24, 26]. However, only a small number of articles focus on the efficiency trade-offs related to transparency. Austin and Flanagan [7] compare the performance of SME and MF on a number of micro-benchmarks. MF was originally proposed as an optimisation of SME. However, it has since been made clear that MF and SME have some complementary performance behaviours [14]. Schmitz et al. [14] study the time-memory trade-off in FSME, a system which attempts to get "the best of both worlds" performance between MF and SME. Ngo et al. [15] show a qualitative optimisation of MF.

Pfeffer et al. [19] study a byte-code level multi-execution technique which they optimise by selectively multi-executing similarly to FSME [14]. However, they work in a statically labeled setting with small lattices, and do are not exposed to the issues of exponential overhead we study here.

Algehed et al. [16] provide a framework for optimising FSME which inspired our discussion of level-assignments. Their "computation-oriented" optimisations are capable of reducing exponential to polynomial overhead. However, these methods have to be applied by manual analysis of the program being optimised. It is unlikely that the technique can be automated to work *for all programs*, as this requires a complete static analysis tool for level assignments.

Some authors have also studied multi-execution in other contexts than security, notably with applications to debugging [22] and program repair [35]. While an implementation of these methods has been heavily optimised [22], they suffer from the same unavoidable exponential overheads as other multi-execution techniques discussed in this paper.

IX. Conclusion

In this paper we have presented a thorough study of the feasibility and efficiency of transparent information flow control. Our results indicate that while transparent IFC is possible to achieve for a diverse set of security criteria including Monotonically Terminating Noninterference and Termination Insensitive Noninterference, current methods are practically infeasible for many systems due to inherent limitations on efficiency. We show that any enforcement mechanism that is secure and transparent also gives rise to a secure and transparent multi-execution based enforcement with polynomial slowdown. We also show that any traditional multi-execution based enforcement mechanism can expect to have large runtime overheads. Furthermore, we prove that efficient black-box IFC is impossible.

We also pose the conjecture that transparent, efficient, and secure IFC enforcement is impossible in general, not just in the black-box case. We give a lemma which we think will help researchers find a proof for our conjecture that is based on our proof that transparent enforcement is intimately linked to multi-execution. Our conjecture, as stated, is without reference to any hardness assumptions for other problems in the complexity theory literature. Readers familiar with complexity theory may note that such un-qualified lower bounds are typically difficult to prove, and we therefore think that a proof of a weakened version of our conjecture may be one suitable next step for the community.

Taken together, our results indicate that the quest for transparent IFC is unlikely to generate a panacea for securing third-party and legacy code. This is a troubling result, it indicates that there is no "easy way out" for securing existing software systems. At the same time, we believe that our results indicate a clear future path for the IFC and programming languages communities. If efficient transparent enforcement is impossible, the community ought instead to focus on tools to help software engineers build secure systems from scratch.

We also note that while our results paint a bleak picture for multi-execution *in general*, they do not mean that there are no application areas for the techniques. For example, when the size of the security lattice is small multi-execution can still be applicable [19, 20]. The techniques are also useful in less efficiency-sensitive contexts than general enforcement, for example in testing [22] and program repair [35]. Similarly, efficient transparent enforcement being unlikely to work does not preclude designing enforcement for the common case to achieve good average-case performance.

References

[1] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[2] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[3] V. Simonet and I. Rocquencourt, "Flow caml in a nutshell," in *Proceedings of the first APPSEM-II workshop*, 2003, pp. 152–165.

[4] A. Banerjee and D. A. Naumann, "Secure information flow and pointer confinement in a java-like language." in *CSFW*, vol. 2, 2002, p. 253.

[5] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1999, pp. 147–160.

[6] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in haskell," in *ACM Sigplan Notices*, vol. 46, no. 12. ACM, 2011, pp. 95–106.

[7] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow," in *ACM Sigplan Notices*, vol. 47, no. 1. ACM, 2012, pp. 165–178.

[8] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*. IEEE, 1982, pp. 11–11.

[9] J. McLean, "Proving noninterference and functional correctness using traces," *Journal of Computer security*, vol. 1, no. 1, pp. 37–57, 1992.

[10] S. Zdancewic and A. C. Myers, "Observational determinism for concurrent program security," in *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.* IEEE, 2003, pp. 29–43.

[11] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *International Conference on Information Systems Security*. Springer, 2008, pp. 56–70.

[12] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, "An empirical study of information flows in real-world javascript," *arXiv preprint arXiv:1906.11507*, 2019.

[13] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 109–124.

[14] T. Schmitz, M. Algehed, C. Flanagan, and A. Russo, "Faceted secure multi execution," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1617–1634.

[15] M. Ngo, N. Bielova, C. Flanagan, T. Rezk, A. Russo, and T. Schmitz, "A better facet of dynamic information flow control," in *WWW'18 Companion: The 2018 Web Conference Companion*, 2018, pp. 1–9.

[16] M. Algehed, A. Russo, and C. Flanagan, "Optimising Faceted Secure Multi-Execution," in *Proc. of the 2019 32nd IEEE Computer Security Foundations Symp.*, ser. CSF '19. IEEE Computer Society, 2019.

[17] D. Zanarini, M. Jaskelioff, and A. Russo, "Precise enforcement of confidentiality for reactive systems," in *2013 IEEE 26th Computer Security Foundations Symposium*. IEEE, 2013, pp. 18–32.

[18] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, "Precise, dynamic information flow for database-backed applications," in *ACM SIGPLAN Notices*, vol. 51, no. 6. ACM, 2016, pp. 631–647.

[19] T. Pfeffer, T. Göthel, and S. Glesner, "Efficient and precise information flow control for machine code through demand-driven secure multi-execution," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. ACM, 2019, pp. 197–208.

[20] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "Flowfox: a web browser with flexible and precise information flow control," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 748–759.

[21] W. Rafnsson and A. Sabelfeld, "Secure multi-execution: Fine-grained, declassification-aware, and transparent," *Journal of Computer Security*, vol. 24, no. 1, pp. 39–90, 2016.

[22] C.-P. Wong, J. Meinicke, L. Lazarek, and C. Kästner, "Faster variational execution with transparent bytecode transformation," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 117, 2018.

[23] D. Hedin and A. Sabelfeld, "A perspective on information-flow control." *Software Safety and Security*, vol. 33, pp. 319–347, 2012.

[24] N. Bielova and T. Rezk, "Spot the difference: Secure multi-execution and multiple facets," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 501–519.

[25] M. Ngo, F. Piessens, and T. Rezk, "Impossibility of precise and sound termination-sensitive security enforcements," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 496–513.

[26] M. Jaskelioff and A. Russo, "Secure multi-execution in haskell," in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 2011, pp. 170–178.

[27] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Computability classes for enforcement mechanisms," Cornell University, Tech. Rep., 2003.

[28] S. A. Zdancewic, "Programming languages for information security," Ph.D. dissertation, Ithaca, NY, USA, 2002, aAI3063751.

[29] T. H. Austin and C. Flanagan, "Permissive dynamic information flow analysis," in *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ser. PLAS '10. New York, NY, USA: ACM, 2010, pp. 3:1–3:12. [Online]. Available: http://doi.acm.org/10.1145/1814217.1814220

[30] N. Bielova and T. Rezk, "A taxonomy of information flow monitors," in *International Conference on Principles of Security and Trust*. Springer, 2016, pp. 46–67.

[31] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell, "Disjunction category labels," in *Nordic conference on secure IT systems*. Springer, 2011, pp. 223–239.

[32] H. Rogers, *Theory of recursive functions and effective computability*. McGraw-Hill New York, 1967, vol. 5.

[33] I. Boloşteanu and D. Garg, "Asymmetric secure multi-execution with declassification," in *International Conference on Principles of Security and Trust*. Springer, 2016, pp. 24–45.

[34] M. Ngo, F. Massacci, D. Milushev, and F. Piessens, "Runtime enforcement of security policies on black box reactive programs," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015, pp. 43–54.

[35] C.-P. Wong, J. Meinicke, and C. Kästner, "Beyond testing configurable systems: applying variational execution to automatic program repair and higher order mutation testing," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and*

Symposium on the Foundations of Software Engineering. ACM, 2018, pp. 749–753.

Appendix

A. Proofs

Proofs showing that $C(S)$ partitions $\mathcal{L}$

Lemma 4. *Given any finite $S \subseteq \mathcal{L}$ and $\ell, \jmath \in S$ such that $\ell \neq \jmath$, $\ell \uparrow S$ is disjoint from $\jmath \uparrow S$.*

Proof. Assume $\hat{\ell} \in \ell \uparrow S$ and $\hat{\ell} \in \jmath \uparrow S$. Clearly $\ell \sqsubseteq \hat{\ell}$ and $\jmath \sqsubseteq \hat{\ell}$, and so $\jmath \sqsubseteq \ell$ and $\ell \sqsubseteq \jmath$, therefore $\ell = \jmath$, a contradiction. □

Lemma 5. *Given any finite $S \subseteq \mathcal{L}$ we have that $\bigcup\{\ \ell \uparrow C(S) \mid \ell \in C(S)\ \} = \mathcal{L}$.*

Proof. Left-to-right inclusion is trivial. For all $\ell \in \mathcal{L}$ it is the case that there is a greatest element $\jmath \in C(S)$ such that $\jmath \sqsubseteq \ell$ (as $\bot \in C(S)$) and $\ell \in \jmath \uparrow C(S)$. □

Proposition 4. *Given any $S \subseteq \mathcal{L}$, the family which maps each $\ell \in C(S)$ to $\ell \uparrow C(S)$ partitions $\mathcal{L}$.*

Proof. Immediate from lemmas 4 and 5. □

Properties of MEF

Theorem 3. *For all p, x, if $\mathcal{L}$ is finite we have that:*

$$\mathit{MEF}[p]x = \mathit{ME}[p]x$$

Proof. We first prove that $\text{MEF}[p](x) \subseteq \text{ME}[p](x)$. If $a^\ell \in \text{MEF}[p](x)$, then there exists some $\jmath$ such that $a^\ell \in p(x \downarrow \jmath)$ and $x \downarrow \ell = x \downarrow \jmath$ and so $a^\ell \in p(x \downarrow \ell)@\ell$ and so $a^\ell \in \text{ME}[p](x)$.

Next we show that $\text{ME}[p](x) \subseteq \text{MEF}[p](x)$. If $a^\ell \in \text{ME}[p](x)$, then clearly $a^\ell \in p(x \downarrow \ell)$ and there is an $\jmath \in C(\mathcal{L}(x))$ such that $x \downarrow \ell = x \downarrow \jmath$ and $\ell \in \jmath \uparrow C(\mathcal{L}(x))$ and, consequently, $a^\ell \in p(x \downarrow \jmath)@(\jmath \uparrow C(\mathcal{L}(x)))$ and so $a^\ell \in \text{MEF}[p](x)$.

We have that $\text{MEF}[p](x) \subseteq \text{ME}[p](x)$ and $\text{ME}[p](x) \subseteq \text{MEF}[p](x)$ when both are defined, and so $\text{MEF}[p](x) = \text{ME}[p](x)$ provided both are defined.

Next we show that when $\mathcal{L}$ is finite, $\text{MEF}[p](x)^\checkmark \Leftrightarrow \text{ME}[p](x)^\checkmark$. If $\text{MEF}[p](x)^\checkmark$ then $p(x \downarrow \ell)^\checkmark$ for all $\ell \in C(\mathcal{L}(x))$. But, for all $\jmath \in \mathcal{L}$ it is the case that $x \downarrow \jmath = x \downarrow \ell$ for some $\ell \in C(\mathcal{L}(x))$ and so $p(x \downarrow \ell)^\checkmark$, and consequently $\text{ME}[p](x)^\checkmark$. If $\text{ME}[p](x)^\checkmark$ then clearly $p(x \downarrow \ell)^\checkmark$ for all $\ell \in C(\mathcal{L}(x))$ and so $\text{MEF}[p](x)^\checkmark$. □

Theorem 9. *MEF is MT-transparent.*

Proof. By Theorem 3 we have that for all p and x it is the case that $\text{MEF}[p](x) = \text{ME}[p](x)$ if $\text{ME}[p](x)$ and $\text{MEF}[p](x)$ are both defined. Furthermore, by Theorem 2 we have that $\text{ME}[p](x) = p(x)$ if p is MT-secure. If the lattice $\mathcal{L}$ is non-finite ME can only be read as a specification, and not an algorithm. However, the argument in Theorem 3 suffices to show that MEF implements the ME specification when it terminates. Therefore, it remains only to show that given an MT-secure p, it is the case that $\text{MEF}[p](x)^\checkmark \Leftrightarrow p(x)^\checkmark$. If $\text{MEF}[p](x)^\checkmark$ then $p(x \downarrow \bigsqcup \mathcal{L}(x))^\checkmark$ and so because $x = x \downarrow \bigsqcup \mathcal{L}(x)$ we have that $p(x)^\checkmark$. Conversely, if $p(x)^\checkmark$ then by p MT-secure we know that $p(x \downarrow \ell)^\checkmark$ for all ℓ and so $\text{MEF}[p](x)^\checkmark$. □

Level Assignments

Proposition 6. *If $\mathbb{L}(p, x) = C(\mathcal{L}(x)) \cap \mathbb{L}'(p)$ for some total $\mathbb{L}'$, then $\mathbb{L}$ is an MT-secure level assignment.*

Proof. If $x \sim_\ell y$ then $C(\mathcal{L}(x)) \sim_\ell C(\mathcal{L}(y))$ and so $\mathbb{L}(p, x) \sim_\ell \mathbb{L}(p, y)$. Consequently, $\mathbb{L}(p, \_)$ is TI-secure. By Lemma 1 we have that $\mathbb{L}$ is TI-secure and so $\text{ME}_{\mathbb{L}}$ is noninterfering. To see that $\text{ME}_{\mathbb{L}}[p]$ is MT-terminating, observe that $\mathbb{L}(p, x \downarrow \ell) \subseteq \mathbb{L}(p, x)$ and so if $\text{ME}_{\mathbb{L}}[p](x)^\checkmark$ then $p(x \downarrow \ell)^\checkmark$ for $\ell \in \mathbb{L}(p, x)$ and so $\text{ME}_{\mathbb{L}}[p](x \downarrow \ell)^\checkmark$. □

Lemmas for Theorems 6 and 7

Recall the extended interpretation ℓ-equivalence from Section V to subsets of $\mathcal{L}$ based on the following definition of $S \downarrow \ell$:

$$S \downarrow \ell = \{\ \ell' \in L \mid \ell' \sqsubseteq \ell\ \}$$

Lemma 1. *If the program $\lambda x.\ \mathbb{L}(p, x)$ is TI-secure for all p, then $\mathbb{L}$ is TI-secure.*

Proof. Consider any ℓ, p, $x \sim_\ell y$ such that $\mathbb{L}(p, x)^\checkmark$ and $\mathbb{L}(p, y)^\checkmark$. Clearly, for all $\jmath \sqsubseteq \ell$, $\jmath \in \mathbb{L}(p, x)$ if and only if $\jmath \in \mathbb{L}(p, y)$ and so if this is the case then:

$$\begin{aligned} &\text{ME}_{\mathbb{L}}[p](x)@\jmath \\ &= p(x \downarrow \jmath) \quad @\jmath \\ &= p(y \downarrow \jmath) \quad @\jmath \\ &= \text{ME}_{\mathbb{L}}[p](y)@\jmath \end{aligned}$$

And otherwise $\text{ME}_{\mathbb{L}}[p](x)@\jmath = \emptyset = \text{ME}_{\mathbb{L}}[p](y)@\jmath$. As $\text{ME}_{\mathbb{L}}[p](x)@\jmath = \text{ME}_{\mathbb{L}}[p](y)@\jmath$ for all $\jmath \sqsubseteq \ell$ we have that $\text{ME}_{\mathbb{L}}[p](x) \sim_\ell \text{ME}_{\mathbb{L}}[p](y)$ and so $\text{ME}_{\mathbb{L}}[p]$ is noninterfering. Consequently, $\text{ME}_{\mathbb{L}}[p]$ is TI-secure for all p and then so is $\mathbb{L}$. □

Lemma 2. *$\mathbb{L}$ is MT-transparent if for all MT-secure programs p and inputs x, we have that $p(x)^\checkmark$ if and only if $\mathbb{L}(p, x)^\checkmark$ and $\mathbb{L}(p, x) = \mathcal{L}(p(x))$.*

Proof. Assume $\mathbb{L}$ satisfies the stated condition and p is an MT-secure program. Clearly, $\text{ME}_{\mathbb{L}}[p](x)^\checkmark$ if and only if $p(x)^\checkmark$. Furthermore $a^\ell \in p(x)$ if and only if $a^\ell \in p(x \downarrow \ell)$ which means that $a^\ell \in \text{ME}_{\mathbb{L}}[p](x)$. For the other direction, if $a^\ell \in \text{ME}_{\mathbb{L}}[p](x)$ then $a^\ell \in p(x \downarrow \ell)$ and so $a^\ell \in p(x)$. □