Spot the Difference: Secure Multi-Execution and Multiple Facets

Nataliia Bielova and Tamara Rezk

Université Côte d'Azur, Inria, France
name.surname@inria.fr

Abstract. We propose a rigorous comparison of two widely known dynamic information flow mechanisms: Secure Multi-Execution (SME) and Multiple Facets (MF). Informally, it is believed that MF simulates SME while providing better performance. Formally, it is well known that SME has stronger soundness guarantees than MF.
Surprisingly, we discover that even if we approach them to enforce the same soundness guarantees, they are still different. While modeling them in the same language, we are able to precisely identify the features of the semantics that lead to their differences. In the process of comparing them, we also discovered four new mechanisms that share features of MF and SME. We prove that one of them simulates SME, which was falsely believed to be true for MF.

1 Introduction

Information flow security [22] is an important guarantee for computer systems. A common security guarantee, called *noninterference*, requires that the secret inputs to the program do not influence (*flow into*) public outputs. In recent years, with the growing impact of highly dynamic languages such as JavaScript, a significant number of dynamic mechanisms [3–5, 12, 15, 19, 24] were proposed for information flow control and enforcement of noninterference.

A dynamic information flow mechanism is *sound* if it ensures equal observable outputs when executions start in equal observable inputs. In other words, a sound dynamic mechanism must detect all insecure executions and enforce noninterference by modifying the insecure executions. An important property of dynamic mechanisms is transparency [7, 13]. A dynamic mechanism is *transparent* if it does not modify the executions of the program that are already secure. In other words, a transparent mechanism does not have any "false positives" when it comes to detecting secure executions.

Secure Multi Execution (SME) [12] and Multiple Facets (MF) [5] are two dynamic sound mechanisms to enforce noninterference. For brevity, we call these mechanisms SME monitor and MF monitor.

The main idea behind SME is to execute a program multiple times, one for each security level. Each execution receives only input visible to its level, and a default value for inputs that should not be visible. In this way, executions cannot depend on non observable inputs. Moreover, SME uses a low priority scheduler

so that non-termination does not depend on high inputs. This allows SME to prevent information leaks due to program non-termination.

The main idea behind MF is to execute a program using faceted values, one facet for each security level. When a facet possesses nothing to be observed, there is a special value to signal this. Moreover, based on the Fenton strategy [14], MF also skips assignment to public variables in a context that depends on a secret to prevent implicit information flows.

By appropriately manipulating the faceted values, a single execution of MF is claimed to *simulate* the multiple executions of SME with the primary benefit of being more performant [2, 5, 25]:

> "Faceted evaluation is a technique for *simulating secure multi-execution* with a single process" – from [25, p. 4]
> "Austin and Flanagan [6] show how secure multi-execution can be *optimized* by executing a single program on faceted values" – from [15, p.15]

One of the two formally studied differences between SME and MF before this work is their soundness guarantee. SME enforces the soundness guarantee of *Termination-Sensitive Noninterference* (TSNI), by preventing information flows when a program has a different termination behaviour based on a secret input. However, MF is only proven to enforce *Termination-Insensitive Noninterference (TINI)*, a weaker information flow policy [22] that does not prevent leaks due to program non-termination. For transparency, SME has been proved to be TSNI precise [12,27], a flavour of monitor transparency which means that SME outputs without changes any execution of a noninterferent program. In contrast, MF is recently demonstrated not to be TINI precise [9].

In this work, we investigate if the generalized belief on the equivalence of SME and MF can be formally supported by appropriate hypotheses. Hence, we raised the following questions:

- Are these monitors essentially different or do they become semantically equivalent when adapted to the same soundness guarantees ?
- Can SME and MF actually be adapted to other soundness guarantees?

Our contributions are the following:

- A formal demonstration of the differences between SME and MF in a simple programming language. We underline their different guarantees in Section 4.
- A comparison of different SME-based and MF-based monitors with respect to soundness and transparency. We have discovered four new monitors:
  - SME-TINI monitor, based on SME, which enforces a weaker termination-insensitive noninterference policy than SME.
  - MFd monitor, based on MF, which is semantically equivalent to SME-TINI (Section 5).
  - MFd-TSNI monitor, based on MFd, semantically equivalent to the original SME (Section 6).
  - MF-TSNI monitor, based on MF, which enforces a stronger termination-sensitive noninterference policy than original MF.

The comparison of the guarantees of all the monitors described in this paper is summarized in Fig. 8 (Section 8). The companion technical report [1] includes all the proofs as well as more details and a formalization of the MFd-TSNI monitor in a language with input and output channels as the one of [12].

2 Soundness and Transparency

The syntax of the language to demonstrate our technical results is:

(programs) $P ::= \mathsf{skip} \mid x{:=}\ e \mid P_1; P_2 \mid \mathsf{if}\ x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \mid \mathsf{while}\ x\ \mathsf{do}\ P$
(expressions) $e ::= \ v \ \mid x \mid e_1 \oplus e_2$

The language's expressions include constants or values (v), variables (x) and operators ($\oplus$) to combine them. We present the standard big-step deterministic semantics denoted by $(P, \mu) \Downarrow \mu'$, where P is the program, and μ is a memory mapping variables to values (Fig. 1).

$$\text{SKIP}\ \frac{}{(\mathsf{skip}, \mu) \Downarrow \mu} \qquad \text{ASSIGN}\ \frac{}{(x := e, \mu) \Downarrow \mu[x \mapsto [\![e]\!]_\mu]} \qquad \text{SEQ}\ \frac{(P_1, \mu) \Downarrow \mu' \qquad (P_2, \mu') \Downarrow \mu''}{(P_1; P_2, \mu) \Downarrow \mu''}$$

$$\text{IF}\ \frac{[\![x]\!]_\mu = \alpha \qquad (P_\alpha, \mu) \Downarrow \mu'}{(\mathsf{if}\ x\ \mathsf{then}\ P_{true}\ \mathsf{else}\ P_{false}, \mu) \Downarrow \mu'} \qquad \text{WHILE}\ \frac{(\mathsf{if}\ x\ \mathsf{then}\ P; \mathsf{while}\ x\ \mathsf{do}\ P\ \mathsf{else}\ \mathsf{skip}, \mu) \Downarrow \mu'}{(\mathsf{while}\ x\ \mathsf{do}\ P, \mu) \Downarrow \mu'}$$

where $[\![x]\!]_\mu = \mu(x)$, $[\![v]\!]_\mu = v$ and $[\![e_1 \oplus e_2]\!]_\mu = [\![e_1]\!]_\mu \oplus [\![e_2]\!]_\mu$

Fig. 1: Language semantics

Noninterference We assume a two-element security lattice with $L \sqsubseteq H$ and a security environment Γ that maps program variables to security levels. By $\mu_{\mathtt{L}}$ we denote the projection of the memory μ on low variables, according to an implicitly parameterized security environment Γ. We first define noninterferent executions, following [9].

Definition 1 (Termination-Sensitive Noninterference for $\mu_{\mathtt{L}}$). *Given a semantics relation $\Downarrow$, program P is* termination-sensitive noninterferent for an initial low memory $\mu_{\mathtt{L}}$, *written $TSNI_{\Downarrow}(P, \mu_{\mathtt{L}})$, if and only if for all memories μ^1 and μ^2, such that $\mu^1_{\mathtt{L}} = \mu^2_{\mathtt{L}} = \mu_{\mathtt{L}}$ we have $\exists \mu'.(P, \mu^1) \Downarrow \mu' \Rightarrow \exists \mu''.(P, \mu^2) \Downarrow \mu'' \wedge \mu'_{\mathtt{L}} = \mu''_{\mathtt{L}}$.*

Program P is *termination-sensitive noninterferent*, written $TSNI_{\Downarrow}(P)$, if all its executions are TSNI, that is, for all $\mu_{\mathtt{L}}$, $TSNI_{\Downarrow}(P, \mu_{\mathtt{L}})$ holds.

Example 1. Consider Program 1, where variable `h` can take only two possible values: 0 and 1.

```
1 l = 1; if h = 1 then (while true do skip)
```
Program 1

If an attacker observes that `l=1`, she learns that `h` was 0, and if she doesn't see any program output (divergence), then she learns that `h` was 1. TSNI captures this kind of information leakage, hence TSNI doesn't hold.

A weaker security condition, called *termination-insensitive noninterference* (*TINI*), allows information leakage through program divergence.

Definition 2 (Termination-Insensitive Noninterference for $\mu_{\mathtt{L}}$). *Given a semantics relation $\Downarrow$, program P is* termination-insensitive noninterferent for an initial low memory $\mu_{\mathtt{L}}$*, written $TINI_{\Downarrow}(P, \mu_{\mathtt{L}})$, if and only if for all μ^1 and μ^2, such that $\mu^1_{\mathtt{L}} = \mu^2_{\mathtt{L}} = \mu_{\mathtt{L}}$, we have $\exists \mu'.(P, \mu^1) \Downarrow \mu' \wedge \exists \mu''.(P, \mu^2) \Downarrow \mu'' \Rightarrow \mu'_{\mathtt{L}} = \mu''_{\mathtt{L}}$.*

Program P is *termination-insensitive noninterferent*, written $TINI_{\Downarrow}(P)$, if all its executions are TINI, that is, for all $\mu_{\mathtt{L}}$, $TINI_{\Downarrow}(P, \mu_{\mathtt{L}})$ holds[1].

Termination-insensitive noninterference is a strictly weaker property than termination-sensitive noninterference [22]. For example, Program 1 is insecure with respect to TSNI, however it is secure with respect to TINI since whenever a program execution terminates, it always finishes in a memory with `l=1`.

Monitor Soundness and Transparency To define a *sound monitor* for termination-sensitive (resp., -insensitive) noninterference, we only substitute the semantics relation $\Downarrow$ with the monitor semantics relation $\Downarrow_M$ in the definitions of TINI and TSNI. Instead of using a subscript $\Downarrow_M$ (e.g., in $TINI_{\Downarrow_M}$) for a semantics of a monitor M, we will use a subscript M (e.g., $TINI_M$).

Definition 3 (Soundness). *Monitor M is* termination-sensitive (resp., -insensitive) sound *if for all programs P, $TSNI_M(P)$ (resp., $TINI_M(P)$).*

A number of works on dynamic information flow monitors try to analyse transparency of monitors. Intuitively, transparency describes how often a monitor accepts (doesn't block or modify) secure program executions without changing the original semantics. Different approaches have been taken to compare transparency of monitors (see [9] for a survey): in this work, we adhere to the standard meaning [7, 13] of "transparency" as the capability of a monitor to accept secure executions and use the term "precision" as the capability to accept all executions of secure programs. To formally define transparency, we first define a predicate $\mathcal{A}(P, \mu, M)$ (where $\mathcal{A}$ stands for "accepted") that holds if:

- whenever a program P terminates for an initial memory μ, then the monitor M will also terminate on μ, producing the same final memory as the original program: $\exists \mu'.\ (P, \mu) \Downarrow \mu' \Rightarrow (P, \mu) \Downarrow_M \mu'$, and
- whenever a program P does not terminate for an initial memory μ (denoted by $\bot$), then the monitor does not terminate for μ: $(P, \mu) \Downarrow \bot \Rightarrow (P, \mu) \Downarrow_M \bot$.

[1] In the following, we don't write the semantics relation $\Downarrow$ when we mean the original program semantics and the semantics is clear from the context.

The notion of *transparency* for TSNI (TINI) requires a monitor to accept all the TSNI (TINI) executions of a program. Our choice of transparency definition is based on the original literature on runtime monitors [7,13], which requires that if a program execution is secure (noninterferent), then the monitor must accept this execution without modifications. Our definition is similar to the one of [21], which considers both terminating and nonterminating executions, however it differs because we don't require the set of executions accepted by a monitor and the set of noninterferent executions to be equal.

Definition 4 (Transparency). *Monitor M is* TSNI (resp., TINI) transparent *if for any program P, and any memory μ, $TINI(P, \mu_{\mathtt{L}}) \Rightarrow \mathcal{A}(P, \mu, M)$ (resp., $TSNI(P, \mu_{\mathtt{L}}) \Rightarrow \mathcal{A}(P, \mu, M)$).*

3 SME and MF Original Semantics

In order to compare SME and MF, we first model them in the same language defined in Section 2. The semantics relation of a command P is denoted by $\Gamma \vdash (P, \mu) \Downarrow_M \mu'$ where Γ is a security environment, and M is the name of the monitor and $\Downarrow_M$ relates a program configuration and a memory. Both SME and MF monitors have deterministic semantics.

Secure Multi-Execution (SME) Devriese and Piessens proposed secure multi-execution (SME) [12]. The idea of SME is to execute the program multiple times: one for each security level. SME has two mechanisms to enforce noninterference:

- Each execution receives only inputs visible to its security level and a fixed default value for each input that should not be visible to the execution. This default value predefines a so-called "default" execution, so that under SME all the interferent executions would behave like a "default" execution.
- A low priority scheduler ensures that lower executions do not depend on the termination of higher executions. Therefore, the low priority scheduler ensures that the program termination based on a secret input does not influence a public output, and hence enforces TSNI.

$$\text{SME}\ \frac{(P, \mu|_\Gamma) \Downarrow \mu_2 \qquad \mu_1 = \begin{cases} \mu' & \mathsf{if}\ \exists \mu'.(P, \mu) \Downarrow \mu' \\ \bot & \mathsf{otherwise} \end{cases}}{\Gamma \vdash (P, \mu) \Downarrow_{\mathbf{SME}} \mu_1 \odot_\Gamma \mu_2}$$

where

$$\mu|_\Gamma(x) = \begin{cases} \mathsf{def}_H & \Gamma(x) = H \\ \mu(x) & \Gamma(x) = L \end{cases} \qquad \mu_1 \odot_\Gamma \mu_2(x) = \begin{cases} \mu_1(x) & \Gamma(x) = H \\ \mu_2(x) & \Gamma(x) = L \end{cases}$$

Fig. 2: Secure Multi-Execution semantics (SME)

The SME adaptation for the while language, taken from [9], is given in Fig. 2, with executions for levels L and H. The $\mu|_\Gamma$ function substitutes the values of all the high variables in μ with the default value def_H, such that all the insecure program executions will behave as an execution predefined by def_H.

Example 2 (SME imitates "default" executions). Consider the following program and assume that the SME's default value is $\mathtt{def}_H\mathtt{=0}$.

```
1 l = 1; if h = 0 then l = 0
```
Program 2

A "default" execution would take $\mathtt{def}_H$ value instead of a real high value and compute the final memory with `l=0`. This program is not TINI, and therefore all its executions will terminate under SME with the memory where `l=0`.

In our SME semantics, the special runtime value $\perp$ represents the idea that no value can be observed (notice that original programs use only standard values). We overload the symbol to also denote a memory that maps every variable to $\perp$. Using memory $\perp$ we simulate the low priority scheduler of SME in our setting: *if the high execution does not terminate*, the low observer will still see the low part of the memory in the SME semantics. In this case all the high variables, whose values should correspond to values obtained in the normal execution of the program, are given value $\perp$. We model the final memory by a merging function $\odot_\Gamma$ that combines high and low parts of two final memories from high and low executions. Notice that even though the semantics becomes non computable, this model allows us to prove the same results as for the original SME and further use it for comparison with MF.

Example 3 (SME prevents leakage through non-termination). Consider Program 3:

```
1 if l = 0 then (while h=0 do skip)
2 else (while h=1 do skip)
```
Program 3

This program is TINI but not TSNI. Assume $\mu = [\mathtt{h=1,\ l=0}]$ and that the default high value used by SME is $\mathtt{def}_H\mathtt{=1}$. The program terminates on memory μ, producing `l=0`, however there exists a memory $\mu' = [\mathtt{h=0,\ l=0}]$, low-equal to μ, on which the original program doesn't terminate, thus leaking secret information through non-termination. SME prevents such leakage, because SME terminates on both memories μ and μ' producing `l=0`.

Multiple Facets (MF) Austin and Flanagan [5] proposed multiple facets (MF). In MF, each variable is mapped to several values or facets, one for each security level: each value corresponds to the view of the variable for an observer at different security level. The main mechanisms used by MF are the following:

- MF uses a special value $\perp$ to signal that a variable contains no information to be observed at a given security level.
- MF uses the Fenton strategy [14] that skips *sensitive upgrades*. A sensitive upgrade is an assignment to a low variable in a high security context that may cause an implicit information flow. If there is a sensitive upgrade, MF semantics does not update the observable facet. Otherwise, if there is no sensitive upgrade, MF semantics updates it according to the original semantics.

$$\text{MF}\ \boxed{\frac{(P, \mu\uparrow_\Gamma) \downarrow_{MF} \hat{\mu}}{\Gamma \vdash (P, \mu) \Downarrow_{\mathbf{MF}} \hat{\mu}\downarrow_\Gamma}} \qquad \text{SKIP}\ \frac{}{(\mathsf{skip}, \hat{\mu}) \downarrow_{MF} \hat{\mu}}$$

$$\text{ASSIGN}\ \frac{}{(x := e, \hat{\mu}) \downarrow_{MF} \hat{\mu}[x \mapsto [e]_{\hat{\mu}}]} \qquad \text{SEQ}\ \frac{(P_1, \hat{\mu}) \downarrow_{MF} \hat{\mu}' \qquad (P_2, \hat{\mu}') \downarrow_{MF} \hat{\mu}''}{(P_1; P_2, \hat{\mu}) \downarrow_{MF} \hat{\mu}''}$$

$$\text{IF-BOT}\ \frac{[x]_{\hat{\mu}} = \langle \alpha : \bot \rangle \qquad (P_\alpha, \hat{\mu}) \downarrow_{MF} \hat{\mu}'}{(\mathsf{if}\ x\ \mathsf{then}\ P_{true}\ \mathsf{else}\ P_{false}, \hat{\mu}) \downarrow_{MF} \hat{\mu}' \otimes \hat{\mu}}$$

$$\text{IF-VAL}\ \frac{[x]_{\hat{\mu}} = \langle \alpha_1 : \alpha_2 \rangle \qquad \alpha_2 \neq \bot \qquad (P_{\alpha_1}, \hat{\mu}) \downarrow_{MF} \hat{\mu}_1 \qquad (P_{\alpha_2}, \hat{\mu}) \downarrow_{MF} \hat{\mu}_2}{(\mathsf{if}\ x\ \mathsf{then}\ P_{true}\ \mathsf{else}\ P_{false}, \hat{\mu}) \downarrow_{MF} \hat{\mu}_1 \otimes \hat{\mu}_2}$$

$$\text{WHILE}\ \frac{(\mathsf{if}\ x\ \mathsf{then}\ P; \mathsf{while}\ x\ \mathsf{do}\ P\ \mathsf{else}\ \mathsf{skip}, \hat{\mu}) \downarrow_{MF} \hat{\mu}'}{(\mathsf{while}\ x\ \mathsf{do}\ P, \hat{\mu}) \downarrow_{MF} \hat{\mu}'}$$

where

$$\mu\uparrow_\Gamma(x) = \begin{cases} \langle \mu(x) : \bot \rangle & \text{if } \Gamma(x) = H \\ \langle \mu(x) : \mu(x) \rangle & \text{if } \Gamma(x) = L \end{cases} \qquad \hat{\mu}\downarrow_\Gamma(x) = \begin{cases} \hat{\mu}(x)_1 & \text{if } \Gamma(x) = H \\ \hat{\mu}(x)_2 & \text{if } \Gamma(x) = L \end{cases}$$

and $\hat{\mu}_1 \otimes \hat{\mu}_2(x) = \langle \hat{\mu}_1(x)_1 : \hat{\mu}_2(x)_2 \rangle$

Fig. 3: Multiple Facets semantics (MF)

Our adaptation of MF semantics is given in Fig. 3, where we use the following notation: a faceted value, denoted $\hat{v} = \langle v_1 : v_2 \rangle$, is a pair of values v_1 and v_2. The first value presents the view of an observer at level H and the second value the view of an observer at level L. In the syntax, we interpret a constant v as the faceted value $\langle v : v \rangle$. The evaluation of faceted expressions is strict in $\bot$ – if an expression contains $\bot$ then it evaluates to $\bot$ – and it is defined as follows:

$$[\hat{v}]_{\hat{\mu}} = \hat{v} \qquad [x]_{\hat{\mu}} = \hat{\mu}(x) \qquad [e_1 \oplus e_2]_{\hat{\mu}} = [e_1]_{\hat{\mu}} \oplus [e_2]_{\hat{\mu}}$$

where $\langle v_1 : v_1' \rangle \oplus \langle v_2 : v_2' \rangle = \langle v_1 \oplus v_2 : v_1' \oplus v_2' \rangle$.

Faceted memories, ranged over by $\hat{\mu}$, are mappings from variables to faceted values. A function $\mu\uparrow_\Gamma$ creates a faceted memory from a memory μ using the labelling function Γ, and function $\hat{\mu}\downarrow_\Gamma$ erases facets from the faceted memory $\hat{\mu}$ and returns a normal memory μ. We use the notation $\hat{\mu}(x)_i$ $(i \in \{1, 2\})$ for the first or second projection of a faceted value stored in x. Similar to the formalisation of SME, the special runtime value $\bot$ represents the idea that no value can be observed (program syntax only uses standard values). Moreover, MF skips any operation that depends on a value $\bot$ (see rule IF-BOT in Fig. 3).

Example 4 (MF uses $\bot$ to signal "no information"). Consider the following program, that copies the secret from `h` to low variable `l` : `l = h` . Given an initial environment μ= [`h=1, l=0`], the function $\mu\uparrow_\Gamma$ creates a faceted memory $\hat{\mu}$, where `h` = $\langle 1 : \bot \rangle$. After assignment, the variable `l` will contain the faceted value of `h`, that will be projected to the $\bot$ value using the function $\hat{\mu}\downarrow_\Gamma$ to erase facets in the end of the execution.

Example 5 (MF "skips" sensitive upgrades). Consider Program 4.

```
1 l = 0; if h = 1 then l = 1 else l = 2
```
Program 4

In MF, the L facet of variable `l` will be the initial value of variable `l` since MF will not update a low variable in a high context. Therefore, all the executions of Program 4 are modified by MF, producing the final memory with `l=0`.

4 Differences between SME and MF

Even though MF is claimed to simulate SME, the example below demonstrates that even in a simple language, SME and MF semantics are different.

Example 6 (SME and MF semantics are different). Consider Program 5 and an initial memory [`h=0, l=0`].

```
1 l = 0;
2 if h = 0 then l = 1;
3 if l = 1 then l = 2 else l = 3
```
Program 5

This program terminates in MF with the final memory where `l=3` because the value of `l` is not updated due to a sensitive upgrade. In contrast, under SME with $\mathsf{def}_H = 0$, the program terminates with the final memory where `l=2`.

Projection Theorem of MF For MF semantics, a Projection Theorem [5, Thm. 1] states that a computation over a 2-faceted memory simulates 2 non-faceted computations, one per each security level. The theorem uses a projection of a faceted memory into a normal memory using the following functions (simplified for our setting), where Lev represents either a high viewer H or a low viewer L, so that $H(\langle v_1 : v_2\rangle) = v_1, L(\langle v_1 : v_2\rangle) = v_2$, and $Lev(\hat{\mu}) = \lambda x.Lev(\hat{\mu}(x))$.

The Projection Theorem states that whenever the monitor terminates[2] for some memory $\hat{\mu}$, $\Gamma \vdash (P, \hat{\mu}) \downarrow_{MF} \hat{\mu}'$, then for any viewer Lev,

$$(P, Lev(\hat{\mu})) \Downarrow Lev(\hat{\mu}').$$

The Projection Theorem may resemble to an equivalence between SME and MF semantics, however, as we have shown above, MF is not equivalent to SME.

Example 7 (Projection Theorem of MF doesn't imply equivalence to SME). Consider Program 6 and an initial memory μ=[`h=1, l=1`]. This program is TINI and TSNI and SME would terminate in the memory μ' = [`h=1, l=0`].

```
1 if h = 0 then l = 0 else l = 0
```
Program 6

[2] Notice that the original program semantics in [5] already contains rules that deal with special $\perp$ values, that skip any operation that involves a $\perp$ value.

The Projection theorem is based on the assumption that MF terminates on a given initial faceted memory. We use the function $\mu \uparrow_\Gamma$ that creates a faceted memory from a normal memory μ given a security labelling Γ. The obtained memory is $\hat{\mu} = [\texttt{h} = \langle 1 : \bot \rangle, \texttt{l} = \langle 1 : 1 \rangle]$. Upon a faceted execution of the program, the final faceted memory is $\hat{\mu}' = [\texttt{h} = \langle 1 : \bot \rangle, \texttt{l} = \langle 0 : 1 \rangle]$.

For a viewer at level H, the initial projected memory is $H(\hat{\mu}) = [\texttt{h=1, l=1}]$, and the final projected memory is $H(\hat{\mu}')$=[h=1, l=0], which corresponds to the original final memory μ'. However, only a viewer a level H is able to see this memory, while a viewer at level L will see a different memory.

For a viewer at level L, the projected initial memory is $L(\hat{\mu}) = [\texttt{h=}\bot\texttt{, l=1}]$, and the final projected memory is also $L(\hat{\mu}') = [\texttt{h=}\bot\texttt{, l=1}]$, since the mechanism of MF skips the sensitive upgrades and the value of `l` is not changed. It means that a viewer at level L will see `l=1` in MF, however will see `l=0` in SME.

Soundness Monitors that enforce TSNI and TINI are comparable with respect to soundness thanks to the fact that TSNI is a stronger guarantee than TINI [22]. SME was previously proven TSNI sound [12], and therefore SME is also TINI sound. Example 3 demonstrated how SME enforces TSNI and hence TINI soundness. In contrast, MF was previously proven TINI sound [5], however it is unable to enforce TSNI.

Example 8 (MF is not TSNI sound). Consider Program 1. When `h=1`, the MF semantics will diverge because the faceted value of `h` is $\langle 1 : \bot \rangle$ and the premises of the IF-BOT rule are not satisfied (the program diverges on line 3). However when `h=0`, the MF semantics will terminate with final memory where `l=1`.

Transparency Devriese and Piessens [12, Thm. 2] have proven that SME is TSNI precise, meaning that for TSNI secure programs, all their executions are not modified by SME.

Theorem 1 ([12, Thm. 2]). *SME is TSNI precise, meaning that for any program P, the following holds: $TSNI(P) \Rightarrow \forall \mu.\ \mathcal{A}(P, \mu, SME)$.*

In this paper, we prove a more fine-grained guarantee for SME, which is TSNI transparency. Notice that TSNI transparency is stronger than TSNI precision because it requires that the monitor not only does not modify any executions of secure programs, but also secure executions of insecure programs.

Theorem 2. *SME is TSNI transparent.*

Example 9. Consider Program 7. This program is not TSNI, however there are TSNI-secure executions of this program when initially `l=1`. For an initial memory where `l=1`, and for any default high value $\texttt{def}_H$, SME will terminate in a final memory, where `l=1`, like the original program.

---

1 `if l=0 then (while h=0 do skip)` **Program 7**

$$\text{SME-TINI}\ \frac{(P,\mu|_\Gamma) \Downarrow \mu_2 \qquad (P,\mu) \Downarrow \mu_1}{\Gamma \vdash (P,\mu) \Downarrow_{\textsc{smetini}} \mu_1 \odot_\Gamma \mu_2}$$

Fig. 4: SME semantics for TINI (SME-TINI)

Example 10 (MF is not TSNI and not TINI transparent). Consider Program 6, which is TSNI secure, and an initial memory [`h=1, l=1`]. The MF semantics will modify this execution. Since the test depends on a high variable `h`, the IF-BOT rule will be used to evaluate the conditional, and only the high facet of the value in `l` will be updated, getting the value 0, while the low facet will not be updated, hence the new faceted value of `l` is $\langle 0 : 1\rangle$. Following the definition of the $\downarrow_\Gamma$ function, the final memory will contain `l=1` because $\Gamma(\texttt{l}) = L$, while the original program would terminate in the memory where `l=0`. Hence, this is a counter example for TSNI and TINI transparency of MF.

5 SME vs MF by downgrading SME to TINI

The first reason for SME and MF to be incomparable is that SME enforces termination-sensitive noninterference (TSNI), whereas MF enforces a weaker version of noninterference called termination-insensitive noninterference (TINI). To formally compare SME and MF, we modify SME semantics in order for SME to enforce the same version of noninterference as MF, which is TINI.

SME that enforces TINI (SME-TINI) We propose a version of SME, that we call SME-TINI and present its semantics in Fig. 4. SME-TINI runs the program multiple times like SME, but it does not have a low priority scheduler and hence is not sensitive to termination leaks.

Example 11 (SME-TINI does not enforce TSNI). Consider Program 1 and a default value for SME is $\texttt{def}_H = 0$. In an initial memory where `h=1`, the program will diverge in the SME-TINI semantics whereas it will terminate with the memory `l=1` in the SME semantics. In an initial memory where `h=0`, the program will terminate with `l=1` in both SME-TINI and SME semantics. This example shows that, in contrast to SME, SME-TINI does not enforce TSNI.

Theorem 3. *SME-TINI is TINI sound.*

Example 12 (SME-TINI is TINI sound). Consider Program 4 and an initial memory [`h=1,l=0`]. SME-TINI with $\texttt{def}_H = 0$ enforces TINI by always terminating in a final memory where `l=2`.

However, differently from original SME, SME-TINI does not provide transparency guarantee.

$$\text{MFD}\ \boxed{\frac{(P, \mu \uparrow_\Gamma^{\mathsf{def}}) \downarrow_{MF} \hat{\mu}}{\Gamma \vdash (P, \mu) \Downarrow_{\mathbf{MFd}} \hat{\mu} \downarrow_\Gamma}}$$

Fig. 5: Multiple Facets with default (MFd).

Example 13 (SME-TINI is not TINI transparent). Consider Program 3, an initial memory μ=[h=0, l=1] and def_H=1. The original program terminates on memory μ= [h=0, l=1]. Though program is TINI, SME-TINI does not terminate on μ because its low execution does not terminate since def_H=1.

Surprisingly, we find out that even if we downgrade SME to only enforce TINI, and SME-TINI and MF now have the same soundness guarantees, still SME-TINI and MF semantics are different.

Example 14 (SME-TINI and MF semantics are different). Consider again Program 4 and an initial memory [h=1, l=0]. SME-TINI with $\mathsf{def}_H = 0$ enforces TINI by always producing an output 2, however MF does not execute an alternative else-branch, and keeps an initial value of l, terminating with the final memory where l=0.

The main reason for a different semantics now is the way in which SME-TINI and MF treat insecure executions: while SME forces all insecure executions to behave like the "default" executions, MF uses the Fenton strategy to skip sensitive upgrades.

Multiple Facets with Default (MFd) To propose a version of MF that has the same semantics as SME-TINI, we replace the $\perp$ value of MF with def_H as the default high value (this is exactly as the default of SME). In fact, there is a maybe different default high value for each high variable, so in fact def_H is a vector of variables but for simplicity of presentation (and without loss of generality), we call it a default value and use only one high variable in our examples.

The new version of MF, that we call MFd, uses the semantics rules of MF, and instead of a $\mu \uparrow_\Gamma$ function that creates a faceted memory in the MF rule, it uses a $\mu \uparrow_\Gamma^{\mathsf{def}}$ function, that is defined as follows:

$$\mu \uparrow_\Gamma^{\mathsf{def}} (x) = \begin{cases} \langle \mu(x) : \mu(x) \rangle & \text{if } \Gamma(x) = L \\ \langle \mu(x) : \mathsf{def}_H \rangle & \text{if } \Gamma(x) = H \end{cases}$$

Therefore, the MFd semantics is presented with only one rule shown in Fig. 5. Since the MFd semantics never introduces a runtime value $\perp$, the MFd rules do not include the rule IF-BOT of the original MF semantics (Fig. 3). Notice that, the fact that the rule IF-BOT is not included implies that one of the bases of original MF, which is to skip sensitive upgrades as originally proposed by Fenton [14], is made obsolete.

Theorem 4. *MFd is TINI sound.*

To prove that MFd is equivalent to SME-TINI, we first propose the following definition of an equivalence relation on two monitor semantics.

Definition 5. *A monitor A is semantically equivalent to a monitor B, written $A \approx B$, if and only if for all programs P, all memories μ and μ', and all labelling functions Γ, the following holds:*

$$\Gamma \vdash (P, \mu) \Downarrow_A \mu' \iff \Gamma \vdash (P, \mu) \Downarrow_B \mu'.$$

Theorem 5. *MFd $\approx$ SME-TINI.*

Example 15 (MFd and SME-TINI semantics are equivalent). Consider Program 4 and an initial memory [h=1, l=0]. SME-TINI with $\mathsf{def}_H = 0$ always terminates in a final memory where l=2. MFd also terminates in a final memory with l=2, because differently from original MF, it does not skip the sensitive upgrades but rather uses the results of the "default" execution, like SME.

6 SME vs MF by upgrading MFd to TSNI

By analysing SME and MF semantics, we concluded that they are different for two reasons. First, SME enforces TSNI, while MF enforces TINI. In the previous section we have downgraded SME to enforce a weaker property TINI, however the resulting SME-TINI monitor was not semantically equivalent to MF. Therefore, we have found the second reason for their difference: while SME is using a default value for high variables in the low execution, MF uses special runtime values $\perp$, allowing the execution of some branches to be skipped.

In the previous section we proposed a new version of MF, called MFd, that solves the second difference of SME and MF, but does not solve the first one: MFd does not have the same strong soundness guarantee, TSNI, that original SME has. Therefore, we propose modifications to the MFd semantics in order for MFd to enforce TSNI.

We propose a new monitor, that we call MFd-TSNI, and present its semantics in Fig. 6. The main difference between MFd and MFd-TSNI is the embedding of a low priority scheduler to schedule with priority the low facet in the execution. This can be observed in the rules IF-VAL and IF-BOT-VAL. The rule IF-VAL simulates the idea behind the low priority scheduler from original SME. The symbol $\perp$ is overloaded to denote a memory that maps every variable to $\perp$ when the high execution does not terminate. We illustrate the efficiency of MFd-TSNI in enforcing TSNI in the following example.

Example 16. Consider Program 8 which is not TSNI and initial memory μ= [h=1, l=1], the default value used to create a faceted memory is def_H= 0.

Program 8

```
1 if h=1 then (while true skip);
2 if h=0 then l=0
```

$$\text{MFD-TSNI}\ \boxed{\frac{(P, \mu \uparrow_\Gamma^{\mathsf{def}}) \downarrow_{MFT} \hat{\mu}}{\Gamma \vdash (P, \mu) \Downarrow_{\mathbf{MFdT}} \hat{\mu} \downarrow_\Gamma}} \qquad \text{SKIP}\ \frac{}{(\mathsf{skip}, \hat{\mu}) \downarrow_{MFT} \hat{\mu}}$$

$$\text{ASSIGN}\ \frac{}{(x := e, \hat{\mu}) \downarrow_{MFT} (\hat{\mu}[x \mapsto [e]_{\hat{\mu}}])} \quad \text{SEQ}\ \frac{(P_1, \hat{\mu}) \downarrow_{MFT} \hat{\mu}' \qquad (P_2, \hat{\mu}') \downarrow_{MFT} \hat{\mu}''}{(P_1; P_2, \hat{\mu}) \downarrow_{MFT} \hat{\mu}''}$$

$$\text{IF-BOT-VAL}\ \frac{[x]_{\hat{\mu}} = \langle \bot : \alpha \rangle \qquad \alpha \neq \bot \qquad (P_\alpha, \hat{\mu}) \downarrow_{MFT} \hat{\mu}'}{(\mathsf{if}\ x\ \mathsf{then}\ P_{true}\ \mathsf{else}\ P_{false}, \hat{\mu}) \downarrow_{MFT} \bot \otimes \hat{\mu}'}$$

$$\text{IF-VAL}\ \frac{\begin{array}{c}[x]_{\hat{\mu}} = \langle \alpha_1 : \alpha_2 \rangle \qquad \alpha_1 \neq \bot \\ \alpha_2 \neq \bot \qquad (P_{\alpha_2}, \hat{\mu}) \downarrow_{MFT} \hat{\mu}_2 \qquad \hat{\mu}_1 = \begin{cases} \hat{\mu}' & \mathsf{if}\ \exists \hat{\mu}'.(P_{\alpha_1}, \hat{\mu}) \downarrow_{MFT} \hat{\mu}' \\ \bot & \mathsf{otherwise} \end{cases}\end{array}}{(\mathsf{if}\ x\ \mathsf{then}\ P_{true}\ \mathsf{else}\ P_{false}, \hat{\mu}) \downarrow_{MFT} \hat{\mu}_1 \otimes \hat{\mu}_2}$$

$$\text{WHILE}\ \frac{(\mathsf{if}\ x\ \mathsf{then}\ P; \mathsf{while}\ x\ \mathsf{do}\ P\ \mathsf{else}\ \mathsf{skip}, \hat{\mu}) \downarrow_{MFT} \hat{\mu}'}{(\mathsf{while}\ x\ \mathsf{do}\ P, \hat{\mu}) \downarrow_{MFT} \hat{\mu}'}$$

Fig. 6: Multiple Facets semantics with default for TSNI (MFd-TSNI)

An initial value of `h` in the new faceted memory $\hat{\mu}$ is `h`=$\langle 1 : 0 \rangle$, while `l` = $\langle 1 : 1 \rangle$. Upon the first test, the IF-VAL rule is applied. This rule first requires that the execution corresponding to the low facet terminates, which is the case and the final faceted memory after the first test is $\hat{\mu}_2 = \hat{\mu}$. However, the program does not terminate if we use the high facet of `h`, therefore all the program variables get assigned to $\bot$ in a memory $\hat{\mu}_1$. After the combination of memories, we get the final memory after the first test, which is $\hat{\mu}_1 \otimes \hat{\mu}_2$, where `h` = $\langle \bot : 0 \rangle$ and `l` = $\langle \bot : 1 \rangle$.

Upon the second test, the IF-BOT-VAL rule is applied since the high facet of variable `h` is now $\bot$. Therefore, MFd-TSNI executes only one branch where `h`=0 and computes the final memory where `l` = $\langle 0 : 0 \rangle$.

We prove that the new monitor MFd-TSNI is semantically equivalent to original SME.

Theorem 6. *MFd-TSNI* $\approx$ *SME.*

As a direct consequence of the semantical equivalence to SME, MFd-TSNI is TSNI sound and TSNI transparent. Notice that MFd was not transparent.

Theorem 7. *MFd-TSNI is TSNI sound and TSNI transparent.*

Example 17 (MFd-TSNI is TINI sound and TSNI sound). Consider Program 4 and $\mathsf{def}_H = 0$. For any initial memory, MFd-TSNI always terminates in final memory where `l=2`, thus enforcing TINI and TSNI.

Example 18 (MFd-TSNI is TSNI transparent). Consider again Program 7. For the initial memory where `l=1`, and for any default high value def_H, MFd-TSNI will terminate in a final memory, where `l=1`, like the original program.

$$\text{MF-TSNI}\ \boxed{\frac{(P,\mu\uparrow_\Gamma)\downarrow_{MFT}\ \hat{\mu}}{\Gamma\vdash(P,\mu)\Downarrow_{\mathbf{MFT}}\ \hat{\mu}\downarrow_\Gamma}}$$

$$\text{IF-BOT}\ \frac{[x]_{\hat{\mu}}=\langle\alpha:\bot\rangle \qquad \alpha\neq\bot \qquad \hat{\mu}_1=\begin{cases}\hat{\mu}' & \mathsf{if}\ \exists\hat{\mu}'.(P_\alpha,\hat{\mu})\downarrow_{MFT}\ \hat{\mu}'\\ \bot & \mathsf{otherwise}\end{cases}}{(\mathsf{if}\ x\ \mathsf{then}\ P_{true}\ \mathsf{else}\ P_{false},\hat{\mu})\downarrow_{MFT}\ \hat{\mu}_1\otimes\hat{\mu}}$$

$$\text{IF-BOT-BOT}\ \frac{[x]_{\hat{\mu}}=\langle\bot:\bot\rangle}{(\mathsf{if}\ x\ \mathsf{then}\ P_{true}\ \mathsf{else}\ P_{false},\hat{\mu})\downarrow_{MFT}\ \hat{\mu}}$$

Fig. 7: Additional rules for the Multiple Facet semantics for TSNI (MF-TSNI)

MF that enforces TSNI Given the technique we used to upgrade MFd to MFd-TSNI to enforce termination-sensitive noninterference, in this section we show how to upgrade the original MF in order to enforce TSNI using the same low priority scheduler.

The new monitor, that we call MF-TSNI, uses the $\uparrow_\Gamma$ function from MF to create a faceted memory, and uses all the rules of MFd-TSNI, with additional two rules to incorporate the possibility of having a special $\bot$ value in the low facet of the faceted value. We present these additional rules in Fig. 7.

We now prove that the new MF-TSNI monitor indeed enforces termination-sensitive noninterference.

Theorem 8. *MF-TSNI is TSNI sound.*

Example 19 (MF-TSNI is TSNI sound and TINI sound). Consider Program 1. When `h=1`, the IF-BOT rule of MF-TSNI (Fig. 7) will construct a memory $\hat{\mu}_1$, where all the variables are assigned to $\bot$ value since the high facet execution does not terminate. Therefore, MF-TSNI will terminate with the final memory where $\mathtt{l}=\langle\bot:1\rangle$. When `h=0`, the MF-TSNI will terminate in the final memory where $\mathtt{l}=\langle1:1\rangle$, thus enforcing TINI and TSNI.

MF-TSNI is not TSNI transparent for the same reason that MF is not TSNI transparent: the Fenton strategy of skipping sensitive upgrades prevents a mechanism from being transparent.

Example 20 (MF-TSNI is not TSNI transparent). Consider again Program 6, which is TSNI and TINI. For an initial memory where `l=1`, MF-TSNI will terminate in a final memory, where `l=1`, thus being not transparent.

7 Related Work

We present only SME and MF closely related work. We refer to [22, 23] for a wider overview on information flow properties, to [7, 13, 18] for a wider overview on transparency properties of monitors, and to [9, 17] for a wider overview on information flow monitors.

Originally, Secure Multi-Eexecution is presented in a while language featuring input/output commands and channels [12]. An output command produces a value that is queued in the output channel. An input command reads a value that is read from the input channel. We model SME as in [9], in a while language without channels. Instead of channels, we use memories mapping variables to values. To simulate an input (resp. output) command, our language reads (resp. writes) a variable from memory. In the original SME semantics [12], a configuration contains a pool of threads, one thread for each level. Then, a scheduler selects to execute first all steps of the lower level threads. Hence, all outputs of a low execution appear first in the output channel in the original SME semantics. The low priority scheduler is simulated in our model by the only rule of Fig. 2. In this rule, the low thread executes to the end to obtain the low part of the final memory and, *if the high thread does not terminate*, the high part of the final memory is $\perp$. Hence, the semantics becomes non computable. With the current model we can at least prove the same results as in the original SME monitor, and further use it for comparison with MF. Notice that, at the cost of simplicity we could have used the original SME language and semantics in order to have computability (we have modelled the MFd-TSNI monitor in the original SME language as a proof of concept in the companion technical report [1]).

SME is proved to be TSNI sound in Theorem 1 of [12]. Kashyap *et al.* [16] investigate different strategies for SME to also enforce several flavours of time-sensitive noninterference. Intuitively, time-sensitive noninterference is stronger than termination-sensitive noninterference because it requires that two executions starting in low-equal memories must terminate within the same number of program execution steps. Other works [10, 20, 26] have proposed other information flow properties, declassification properties, for modified SME monitors. We do not study in this work SME-based monitors for declassification. SME is proved to be TSNI precise in Theorem 2 of [12]. Notice that TSNI precision is a weaker property than transparency since a program which is not secure may still have some secure executions.

TSNI transparency does not hold for original SME because the low priority scheduler may reorder outputs compared to the original program semantics, letting outputs of low executions appear first in the output channel. Zanarini et al. [27] propose a modification to SME in order to prove a version of TSNI transparency (In fact, they prove a property called CP precision in Theorem 23 of [27], which is a weaker notion that TSNI transparency because it recognizes as secure a program that silently diverges on one branch, and terminates without producing any outputs on the other branch). In contrast, we can prove TSNI transparency in our SME model (and also CP precision) without need of the SME modifications proposed in Zanarini et al. because reordering is not visible in our model due to the lack of output channels, and intermediate outputs.

Zanarini et al. [27] also prove a version of TINI transparency for their TSNI sound SME-based monitor (Theorem 22 of [27]). Using our notations, their notion of TINI transparency is different from ours since if an execution is secure, if the original program terminates in a final memory μ and *if the monitor ter-*

minates in final memory μ', then μ and μ' should be low equal (in fact, they prove a property called ID-transparency in Theorem 22 of [27], which recognizes as transparent a monitor that always diverges).

SME is also shown TSNI sound and TSNI precise for a language featuring dynamic code evaluation [6] and adapted to reactive systems [8]. SME is implemented in a real browser called FlowFox [11], and SME guarantees via program transformations are implemented in JavaScript and Python [6].

Originally, Multiple Facets is presented in a lambda calculus with mutable reference cells and reactive input/output [5]. In contrast, we model MF in an imperative while language without mutable references. Moreover, since our language features memories that map variables to values, we use security environments as a means to create faceted values in our MF model. As we do, the original MF semantics [5] uses the special value $\perp$ in order to model the Fenton strategy [14], which roughly means to skip sensitive upgrades [3, 28] to prevent implicit flows.

MF is also modelled in [9] using an imperative while language as ours. The semantics in [9] uses security environments and program counters in order to implement the Fenton strategy. Our formalisation is simpler since we use facets and $\perp$ to do this, as in [5]. MF is proved to be TINI sound in Theorem 2 of [5] and also is extended to declassification and proved sound in Theorem 6 of [5].

Transparency guarantees of MF are studied in [9]. It was first shown that MF is not TINI transparent (more precisely, TINI transparency is called true transparency in [9]). Using a notion of false transparency, it is then shown that MF can accept more insecure executions than any other information flow monitor with the exception of SME (Table 1 of [9]). Moreover, Theorems 3 and 4 of [5] prove that MF generalizes no-sensitive upgrade monitor (NSU) [3, 28] and permissive-upgrade monitor (PU) [4]. These theorems imply that MF is relatively more transparent than NSU and PU [9].

MF has been implemented in JavaScript as a Firefox browser extension [5] and also as a Haskell Library using monads [25].

8 Conclusion

We have formally compared SME, MF, and other mechanisms derived from them. We present a summary of the comparison in Fig. 8. We have first downgraded SME to enforce only TINI, and proposed a new version of MF, called MFd, which is indeed semantically equivalent to a TINI version of SME. We then upgraded the MFd monitor to enforce TSNI and proposed a new monitor that we call MFd-TSNI. We have proven that

	Soundness		Transparency	
	TINI	TSNI	TINI	TSNI
SME	✓	✓	✗	✓
MF	✓	✗	✗	✗
SME-TINI	✓	✗	✗	✗
MFd	✓	✗	✗	✗
MFd-TSNI	✓	✓	✗	✓
MF-TSNI	✓	✓	✗	✗

Fig. 8: Summary of our results

MFd-TSNI is semantically equivalent to SME, and therefore enjoys the same TSNI soundness and TSNI transparency guarantees as SME. Finally, we propose to upgrade MF semantics so that it can also enforce termination-sensitive noninterference (TSNI). The new monitor, that we call MF-TSNI, is not semantically equivalent to MFd-TSNI, and is not TSNI transparent. Both SME [10, 20, 26], and MF [5] have been extended to handle declassification, a security property more versatile than noninterference. It is left as future work to understand if our results generalize to declassification properties in order to compare SME and MF.

Acknowledgment

We would like to thank Frank Piessens on valuable feedback on earlier versions of this paper and anonymous reviewers who helped us to improve the paper. This work has been partially supported by the ANR project AJACS ANR-14-CE28-0008.

References

1. Spot the Difference: SME and MF Technical Report. `https://goo.gl/b7yoQ9`.
2. T. Austin, K. Knowles, and C. Flanagan. Typed faceted values for secure information flow in haskell. Technical Report UCSC-SOE-14-07, University of California, Santa Cruz, 2014.
3. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS'09*, pages 113–124, 2009.
4. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS'10*, pages 3:1–3:12. ACM, 2010.
5. T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. of the 39th Symposium of Principles of Programming Languages*. ACM, 2012.
6. G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas. Secure multi-execution through static program transformation. In *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE*, 2012.
7. L. Bauer, J. Ligatti, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.
8. N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proc. of the 5th International Conference on Network and System Security (NSS 2011)*, pages 97–104. IEEE, 2011.
9. N. Bielova and T. Rezk. A taxonomy of information flow monitors. In *International Conference on Principles of Security and Trust (POST 2016)*, 2016. To appear.
10. I. Bolosteanu and D. Garg. Asymmetric secure multi-execution with declassification. In *International Conference on Principles of Security and Trust (POST 2016)*, 2016. To appear.
11. W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a Web Browser with Flexible and Precise Information Flow Control. In *Proc. of the 19th ACM Conference on Communications and Computer Security*, pages 748–759, 2012.

12. D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *Proc. of the 2010 Symposium on Security and Privacy*, pages 109–124. IEEE, 2010.
13. U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2003.
14. J. S. Fenton. Memoryless subsystems. *Comput. J.*, 17(2):143–147, 1974.
15. D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive hybrid information flow control for a javascript-like language. In *IEEE 28th Computer Security Foundations Symposium, CSF*, 2015.
16. V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 413–428, 2011.
17. G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University and University of Rennes 1, 2007.
18. J. Ligatti, L. Bauer, and D. Walker. Enforcing Non-safety Security Policies with Program Monitors. In *Proc. of the 10th European Symposium on Research in Computer Security*, volume 3679 of *LNCS*, pages 355–373. Springer-Verlag Heidelberg, 2005.
19. A. G. A. Matos, J. F. Santos, and T. Rezk. An Information Flow Monitor for a Core of DOM - Introducing References and Live Primitives. In *Trustworthy Global Computing - 9th International Symposium, TGC*, 2014.
20. W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *2013 IEEE 26th Computer Security Foundations Symposium*, 2013.
21. W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. *Journal of Computer Security*, 24(1):39–90, 2016.
22. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.
23. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
24. J. F. Santos and T. Rezk. An Information Flow Monitor-Inlining Compiler for Securing a Core of Javascript. In *ICT Systems Security and Privacy Protection - 29th IFIP TC 11 International Conference, SEC 2014*, 2014.
25. T. Schmitz, D. Rhodes, T. H. Austin, K. Knowles, and C. Flanagan. Faceted dynamic information flow via control and data monads. In *POST 2016*, pages 3–23, 2016.
26. M. Vanhoef, W. D. Groef, D. Devriese, F. Piessens, and T. Rezk. Stateful declassification policies for event-driven programs. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014*, pages 293–307, 2014.
27. D. Zanarini, M. Jaskelioff, and A. Russo. Precise enforcement of confidentiality for reactive systems. In *IEEE 26th Computer Security Foundations Symposium*, pages 18–32, 2013.
28. S. A. Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.