A Taxonomy of Information Flow Monitors

Nataliia Bielova, Tamara Rezk

A Taxonomy of Information Flow Monitors

Nataliia Bielova and Tamara Rezk

Inria, France
name.surname@inria.fr

Abstract. We propose a rigorous comparison of information flow monitors with respect to two dimensions: soundness and transparency.
For soundness, we notice that the standard information flow security definition called Termination-Insensitive Noninterference (TINI) allows the presence of termination channels, however it does not describe whether the termination channel was present in the original program, or it was added by a monitor. We propose a stronger notion of noninterference, that we call Termination-Aware Noninterference (TANI), that captures this fact, and thus allows us to better evaluate the security guarantees of different monitors. We further investigate TANI, and state its formal relations to other soundness guarantees of information flow monitors.
For transparency, we identify different notions from the literature that aim at comparing the behaviour of monitors. We notice that one common notion used in the literature is not adequate since it identifies as better a monitor that accepts insecure executions, and hence may augment the knowledge of the attacker. To discriminate between monitors' behaviours on secure and insecure executions, we factorized two notions that we call true and false transparency. These notions allow us to compare monitors that were deemed to be incomparable in the past.
We analyse five widely explored information flow monitors: no-sensitive-upgrade (NSU), permissive-upgrade (PU), hybrid monitor (HM), secure multi-execution (SME), and multiple facets (MF).

1 Introduction

Motivated by the dynamic nature and an extensive list of vulnerabilities found in web applications in recent years, several dynamic enforcement mechanisms in the form of information flow monitors [5–7, 9, 12, 14, 17, 23, 27, 33], have been proposed. In the runtime monitor literature [8, 13], two properties of monitors are considered specially important: soundness and transparency. In this work, we rigorously compare information flow monitors with respect to these two dimensions. We analyse five widely explored information flow monitor techniques: no-sensitive-upgrade (NSU) [33], permissive-upgrade (PU) [6], hybrid monitor (HM) [14], secure multi-execution (SME) [12], and multiple facets (MF) [7].

Soundness An information flow monitor is sound when it ensures that observable outputs comply with a given information flow policy. In the case of noninterference, the monitor must ensure equal observable outputs if executions start in

equal observable inputs. We notice that some monitoring techniques introduce new termination channels, whereas others don't. The standard information flow security definition called *Termination-Insensitive Noninterference (TINI)* does not account for termination: only initial memories in which the program terminates should lead to equal observable outputs. Thus, TINI allows the presence of termination channels, however it does not describe whether the termination channel was present in the original program, or it was added by a monitor. *Termination-Sensitive Noninterference* (TSNI), on the other hand, is a stronger policy that disallows the presence of any termination channel. However, most information flow monitors do not satisfy TSNI. Hence, existing definitions do not allow us to discriminate between different monitors with respect to the security guarantees that they provide. We propose a notion of noninterference, stronger than TINI but weaker than TSNI, that we call *Termination-Aware Noninterference (TANI)*, that captures the fact that the monitor does not introduce a new termination channel, and thus allows to better evaluate the security guarantees of different monitors. We discovered that HM, SME, and MF do satisfy TANI, while NSU and PU do not satisfy TANI.

Example 1 (NSU introduces a termination channel). Consider the following program, where each variable can take only two possible values: 0 and 1.

Program 1

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

This program is leaking confidential information – upon observing output `l=0` (`l=1`), it's possible to derive that `h=1` (`h=0`). In spite of this fact, NSU allows the execution of this program starting in a memory [`h=1, l=0`] and blocks the execution otherwise, thus introducing a new termination channel.

Transparency An information flow monitor is transparent when it preserves program semantics if the execution complies with the policy. In the case of noninterference, the monitor must produce the same output as an original program execution with a value that only depends on observable inputs. We identify different common notions from the literature that aim at comparing the behaviour of monitors: precision, permissiveness, and transparency. We notice that permissiveness is not adequate since it identifies as better a monitor that accepts insecure executions, and hence may augment the knowledge of the attacker, given that the attacker has knowledge based on the original executions. To discriminate between monitors' behaviours on secure and insecure executions, we factorized two notions that we call true and false transparency. *True transparency* corresponds to the standard notion of transparency in the field of runtime monitoring: the ability of a monitor to preserve semantics of secure executions. An information flow monitor is *false transparent* when it preserves semantics of the original program execution that does not comply with the security policy. False transparency might seem contradictory to soundness at first sight but this is not the case since information flow is not a property of one execution [2, 24] but a property of several executions, also called a hyperproperty [11, 29]. These two

notions of transparency allow us to compare monitors that were deemed to be incomparable in the past. In particular, we prove that HM is *more TSNI precise* (more true transparent for the set of TSNI secure programs) than NSU and NSU is more false transparent than HM. Proofs can be found in the companion technical report [1].

Our contributions are the following:

1. We propose a new information flow policy called termination-aware noninterference (TANI) that allows us to evaluate monitors according to their soundness guarantees. We prove that TANI is stronger than TINI but weaker than TSNI that disallows any termination channels.
2. We identify two different notions of transparency that are used in the literature as the same notion and we call them true and false transparency.
3. We generalize previous results from Hedin et al. [16]: we show that dynamic and hybrid monitors become comparable when the two flavors of transparency are separated into true and false transparency.
4. We analyse and compare five major monitors previously proved sound for TINI: NSU, PU, HM, SME and MF. Table 1 in Section 8 summarizes our results for TANI, true and false transparency.

2 Knowledge

We assume a two-element security lattice with $L \sqsubseteq H$ and we use $\sqcup$ as the least upper bound. A security environment Γ maps program variables to security levels. By $\mu_{\mathtt{L}}$ we denote a projection of low variables of the memory μ, according to an implicitly parameterized security environment Γ. The program semantics is defined as a big-step evaluation relation $(P, \mu) \Downarrow (v, \mu')$, where P is a program that produces only one output v at the end of execution. We assume that v is visible to the attacker at level L and that the program semantics is deterministic. The attacker can gain knowledge while observing output v. Following Askarov and Sabelfeld [3, 4], we define knowledge as a set of low-equal memories, that lead to the program observation v.

Definition 1 (Knowledge). *Given a program P, the low part $\mu_{\mathtt{L}}$ of an initial memory μ, and an observation v, the* knowledge *for semantics relation $\Downarrow$ is a set of memories that agree with μ on low variables and can lead to an observation v: $k_{\Downarrow}(P, \mu_{\mathtt{L}}, v) = \{\mu' \mid \mu_{\mathtt{L}} = \mu'_{\mathtt{L}} \wedge \exists \mu''.(P, \mu') \Downarrow (v, \mu'')\}$.*

Notice that knowledge corresponds to *uncertainty* about the environments in the knowledge set: any environment is a possible program input. The attacker believes that the environments outside of the knowledge set are *impossible* inputs. Upon observing a program output, the uncertainty might decrease because the new output may render some inputs impossible. This means that the knowledge set may become smaller, thus increasing the knowledge of the attacker.

To specify a security condition, we define what it means for an attacker not to gain any knowledge. Given a program P, and a low part $\mu_{\mathtt{L}}$ of an initial

memory μ, the attacker's knowledge before the execution of the program is a set of memories that agree with μ on low variables. This set is an equivalence class of low-equal memories: $[\mu]_{\mathtt{L}} = \{\mu' \mid \mu_{\mathtt{L}} = \mu'_{\mathtt{L}}\}$.

Definition 2 (Possible outputs). *Given a program P and the low part $\mu_{\mathtt{L}}$ of an initial memory μ, a set of observable outputs for semantics relation $\Downarrow$ is:* $\mathcal{O}_{\Downarrow}(P, \mu_{\mathtt{L}}) = \{v \mid \exists \mu', \mu''.\ \mu_{\mathtt{L}} = \mu'_{\mathtt{L}} \wedge (P, \mu') \Downarrow (v, \mu'')\}$.

In the following, we don't write the semantics relation $\Downarrow$ when we mean the program semantics; the definitions in the rest of this section can be also used with the subscript parameter $\Downarrow$ when semantics has to be explicit.

We now specify the security condition as follows: by observing a program output, the attacker is not allowed to gain any knowledge.

Definition 3 (Termination-Sensitive Noninterference). *Program P is* termination-sensitively noninterferent *for an initial low memory $\mu_{\mathtt{L}}$, written $TSNI(P, \mu_{\mathtt{L}})$, if for all possible observations $v \in \mathcal{O}(P, \mu_{\mathtt{L}})$, we have*

$$[\mu]_{\mathtt{L}} = k(P, \mu_{\mathtt{L}}, v)$$

A program P is termination-sensitively noninterferent*, written $TSNI(P)$, if for all possible initial memories μ, $TSNI(P, \mu_{\mathtt{L}})$.*

The above definition is *termination-sensitive* because it does not allow an attacker to learn the secret information from program divergence. Intuitively, if the program terminates on all low-equal memories, and it produces the same output v then it satisfies TSNI. If the program doesn't terminate on some of the low-equal memories, then for all possible observations v, the knowledge $k(P, \mu_{\mathtt{L}}, v)$ becomes a subset of $[\mu]_{\mathtt{L}}$ and doesn't satisfy the definition.

Example 2. Consider Program 2. If the attacker observes that `l=1`, then he learns that `h` was 0, and if the attacker doesn't see any program output (divergence), the attacker learns that `h` was 1. TSNI captures this kind of information leakage, hence TSNI doesn't hold.

Program 2

```
1 l = 1;
2 (while (h=1) do skip);
3 output l
```

Proposition 1. *$TSNI(P)$ holds if and only if for all pairs of memories μ^1 and μ^2, we have: $\mu^1_{\mathtt{L}} = \mu^2_{\mathtt{L}} \wedge \exists \mu'.(P, \mu^1) \Downarrow (v_1, \mu') \Rightarrow \exists \mu''.(P, \mu^2) \Downarrow (v_2, \mu'') \wedge v_1 = v_2$.*

Termination-sensitive noninterference sometimes is too restrictive as it requires a more sophisticated program analysis or monitoring that may reject many secure executions of a program. A weaker security condition, called *termination-insensitive noninterference* ($TINI$), allows information flows through program divergence, while still offering information flow security.

To capture this security condition, we follow the approach of Askarov and Sabelfeld [4], by limiting the allowed attacker's knowledge to the set of low-equal memories where the program terminates. Since termination means that some output is observable, a set that we call a *termination knowledge*, is a union of all knowledge sets that correspond to some program output: $\bigcup_{v'} k(P, \mu_{\mathtt{L}}, v')$.

Definition 4 (Termination-Insensitive Noninterference). *Program P is* termination-insensitively noninterferent for an initial low memory $\mu_{\mathtt{L}}$*, written* $TINI(P, \mu_{\mathtt{L}})$*, if for all possible observations $v \in \mathcal{O}(P, \mu_{\mathtt{L}})$, we have*

$$\bigcup_{v' \in \mathcal{O}(P,\mu_{\mathtt{L}})} k(P, \mu_{\mathtt{L}}, v') = k(P, \mu_{\mathtt{L}}, v).$$

A program P is termination-insensitively noninterferent*, written* $TINI(P)$*, if for all possible initial memories μ,* $TINI(P, \mu_{\mathtt{L}})$.

Example 3. TINI recognises the Program 2 as secure, since the attacker's *termination knowledge* is only a set of low-equal memories where the program terminates. For example, for $\mu_{\mathtt{L}}$= [`l=0`], only one observation `l=1` is possible when `h=0`, therefore $TINI$ holds: $\bigcup_{v' \in \{1\}} k(P, \mathtt{l=0}, v') = [\mathtt{h=0,\ l=0}] = k(P, \mathtt{l=0}, 1)$.

Proposition 2. $TINI(P)$ *holds if and only if for all pairs of memories μ^1 and μ^2, we have:* $\mu^1_{\mathtt{L}} = \mu^2_{\mathtt{L}} \wedge \exists \mu'.(P, \mu^1) \Downarrow (v_1, \mu') \wedge \exists \mu''.(P, \mu^2) \Downarrow (v_2, \mu'') \Rightarrow v_1 = v_2$.

3 Monitor soundness

In this section, we consider dynamic mechanisms for enforcing information flow security. For brevity, we call them "monitors". The monitors we consider are purely dynamic monitors, such as NSU and PU, hybrid monitors in the style of Le Guernic et al. [20,21] that we denote by HM, secure multi-execution (SME), and multiple facets monitor (MF). All the mechanisms we consider have deterministic semantics denoted by $\Downarrow_M$, where M represents a particular monitor. All the monitors enforce at least *termination-insensitive noninterference (TINI).*[1] Since TINI accepts termination channels, it also allows the monitor to introduce new termination channels even if an original program did not have any. In the next section, we will propose a new definition for soundness of information flow monitors, capturing that a monitor should not introduce a new termination channel. But, first, we set up the similar definitions of termination-sensitive and -insensitive noninterference for a monitored semantics. Instead of using a subscript $\Downarrow_M$ for a semantics of a monitor M, we will use a subscript M.

Definition 5 (Soundness of TSNI enforcement). *Monitor M soundly enforces* termination-sensitive noninterference*, written* $TSNI(M)$*, if for all possible programs P,* $TSNI_M(P)$.

[1] This is indeed a lower bound since some monitors, like SME, also enforce termination- and time-sensitive noninterference.

Proposition 1 proves that this definition of TSNI soundness is equivalent to the standard two-run definition if we substitute the original program semantics with the monitor semantics. Similarly, to define a sound TINI monitor, we restate Definition 4 of TINI with the monitored semantics. The definition below is equivalent to the standard two-run definition (see Proposition 2).

Definition 6 (Soundness of TINI enforcement). *Monitor M soundly enforces* termination-insensitive noninterference*, written $TINI(M)$, if for all possible programs P, $TINI_M(P)$.*

This definition compares the initial knowledge and the final knowledge of the attacker under the monitor semantics. But in practice, an attacker has also the initial knowledge of the original program semantics (see Example 1).

4 Termination-Aware Noninterference

We propose a new notion of soundness for the monitored semantics, called *Termination-Aware Noninterference (TANI)* that does not allow a monitor to introduce a new termination channel.

Intuitively, all the low-equal memories, on which the original program terminates, should be treated by the monitor in the same way, meaning the monitor should either produce the same result for all these memories, or diverge on all of them. In terms of knowledge, it means that the knowledge provided by the monitor, should be smaller or equal than the knowledge known by the attacker before running the program. Additionally, in the case the original program always diverges, TANI holds if the monitor also always diverges or if the monitor always terminates in the same value.

Definition 7 (Termination-Aware Noninterference). *A monitor $\Downarrow_M$ is* Termination-Aware Noninterferent (TANI)*, written $TANI(M)$, if for all programs P, initial memories μ, and possible observations $v \in \mathcal{O}_M(P, \mu_{\mathtt{L}})$, we have:*

- $\mathcal{O}(P, \mu_{\mathtt{L}}) \neq \emptyset \implies \bigcup_{v' \in \mathcal{O}(P, \mu_{\mathtt{L}})} k(P, \mu_{\mathtt{L}}, v') \subseteq k_M(P, \mu_{\mathtt{L}}, v)$
- $\mathcal{O}(P, \mu_{\mathtt{L}}) = \emptyset \implies (\mathcal{O}_M(P, \mu_{\mathtt{L}}) = \emptyset \vee [\mu]_{\mathtt{L}} = k_M(P, \mu_{\mathtt{L}}, v))$

Notice that, for the case that the original program sometimes terminate ($\mathcal{O}(P, \mu_{\mathtt{L}}) \neq \emptyset$)), we do not require equality of the two sets of knowledge since the knowledge set of the monitored program can indeed be bigger than the knowledge set of the attacker before running the program[2]. The knowledge set may increase when a monitor terminates on the memories where the original program did not terminate (e.g., SME from Section 5 provides such enforcement).

Example 4 (TANI enforcement). Coming back to Program 1, TANI requires that on two low-equal memories [`h=0, l=0`] and [`h=1, l=0`] where the original program terminates, the monitor behaves in the same way: either it terminates on both memories producing the same output, or it diverges on both memories.

[2] Remember that the bigger knowledge set corresponds to the smaller knowledge or to the increased uncertainty.

It is well-known that TSNI is a strong form of noninterference that implies TINI. We now formally state the relations between TINI, TANI and TSNI.

Theorem 1. *$TSNI(M) \Rightarrow TANI(M)$ and $TANI(M) \Rightarrow TINI(M)$.*

5 Which monitors are TANI?

We now present five widely explored information flow monitors and prove whether these monitors comply with TANI. In order to compare the monitors, we first model all of them in the same language. Thus, our technical results are based on a simple imperative language with one output (see Figure 1). The language's expressions include constants or values (v), variables (x) and operators ($\oplus$) to combine them. We present the standard big-step program semantics in Figure 2.

$P ::= S;\ \mathsf{output}\ x$
$S ::= \mathsf{skip} \mid x := e \mid S_1; S_2 \mid \mathsf{if}\ x\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \mid \mathsf{while}\ x\ \mathsf{do}\ S$
$e ::= v \mid x \mid e_1 \oplus e_2$

Fig. 1: Language syntax

$$\textsc{skip}\ \frac{}{(\mathsf{skip}, \mu) \Downarrow \mu} \qquad \textsc{assign}\ \frac{}{(x := e, \mu) \Downarrow \mu[x \mapsto [\![e]\!]_\mu]} \qquad \textsc{seq}\ \frac{(S_1, \mu) \Downarrow \mu' \qquad (S_2, \mu') \Downarrow \mu''}{(S_1; S_2, \mu) \Downarrow \mu''}$$

$$\textsc{if}\ \frac{[\![x]\!]_\mu = \alpha \qquad (S_\alpha, \mu) \Downarrow \mu'}{(\mathsf{if}\ x\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}, \mu) \Downarrow \mu'} \qquad \textsc{while}\ \frac{(\mathsf{if}\ x\ \mathsf{then}\ S; \mathsf{while}\ x\ \mathsf{do}\ S\ \mathsf{else}\ \mathsf{skip}, \mu) \Downarrow \mu'}{(\mathsf{while}\ x\ \mathsf{do}\ S, \mu) \Downarrow \mu'}$$

$$\textsc{output}\ \frac{[\![x]\!]_\mu = v}{(\mathsf{output}\ x, \mu) \Downarrow (v, \mu)}$$

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

Fig. 2: Language semantics

The semantics relation of a command S is denoted by $pc \vdash (\Gamma, S, \mu) \Downarrow_M (\Gamma', \mu')$ where pc is a program counter, M is the name of the monitor and Γ is a security environment mapping variables to security levels. All the considered monitors are flow-sensitive, and Γ may be updated during the monitored execution. We assume that the only output produced by the program is visible to the attacker at level L. Since our simple language supports only one output at the end of the program, the OUTPUT rule of the monitors is defined only for $pc = L$, and thus only checks the security level of an output variable x.

No-sensitive upgrade (NSU) The *no-sensitive upgrade approach* (NSU) first proposed by Zdancewic [33] and later applied by Austin and Flanagan [5] is

based on a purely dynamic monitor that controls only one execution of the program. To avoid implicit information flows, the NSU disallows any upgrades of a low security variables in a high security context. Consider Program 1: since the purely dynamic monitor accepts its execution when `h=1`, it should block the execution when `h=0` to enforce TINI. NSU does so by blocking the second execution since the low variable `l` is updated in a high context.

$$\textsc{skip}\ \frac{}{pc \vdash (\Gamma, \mathsf{skip}, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma, \mu)}$$

$$\textsc{assign}\ \frac{[\![e]\!]_\mu = v \qquad pc \sqsubseteq \Gamma(x) \qquad \Gamma' = \Gamma[x \mapsto \Gamma(e) \sqcup pc]}{pc \vdash (\Gamma, x := e, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma', \mu[x \mapsto v])}$$

$$\textsc{seq}\ \frac{pc \vdash (\Gamma, S_1, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma', \mu') \qquad pc \vdash (\Gamma', S_2, \mu') \Downarrow_{\mathbf{NSU}} (\Gamma'', \mu'')}{pc \vdash (\Gamma, S_1; S_2, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma'', \mu'')}$$

$$\textsc{if}\ \frac{[\![x]\!]_\mu = \alpha \qquad pc \sqcup \Gamma(x) \vdash (\Gamma, S_\alpha, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma', \mu')}{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma', \mu')}$$

$$\textsc{while}\ \frac{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S; \mathsf{while}\ x\ \mathsf{do}\ S\ \mathsf{else}\ \mathsf{skip}, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma', \mu')}{pc \vdash (\Gamma, \mathsf{while}\ x\ \mathsf{do}\ S, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma', \mu')}$$

$$\textsc{output}\ \frac{[\![x]\!]_\mu = v \qquad \Gamma(x) = L}{L \vdash (\Gamma, \mathsf{output}\ x, \mu) \Downarrow_{\mathbf{NSU}} (v, \Gamma, \mu)}$$

Fig. 3: NSU semantics

Our NSU formalisation for a simple imperative language is similar to that of Bichhawat *et al.* [10]. The main idea of NSU appears in the ASSIGN rule: the monitor blocks "sensitive upgrades" when a program counter level pc is not lower than the level of the assigned variable x. Figure 3 represents the semantics of NSU monitor. We use $\Gamma(e)$ as the least upper bound of all variables occurring in expression e. If e contains no variables, then $\Gamma(e) = L$. NSU was proven to enforce termination-insensitive noninterference (TINI) (see [5, Thm. 1]).

Example 5 (NSU is not TANI). Consider Program 1 and an initial memory [`h=1, l=0`]. NSU does not satisfy TANI, since the monitor terminates only on one memory, i.e., $k_M(P, \mu_{\mathtt{L}}, v) =$ [`h=1, l=0`], while the original program terminates on both memories, low-equal to [`l=0`].

Permissive Upgrade (PU) The NSU approach suffices to enforce TINI, however it often blocks a program execution pre-emptively. Consider Program 3. This program is TINI, however NSU blocks its execution starting in memory [`h=0, l=0`] because of a sensitive upgrade under a high security context.

```
1 if h = 0 then l = 1;
2 l := 0;
3 output l
```

Program 3

Austin and Flanagan proposed a less-restrictive strategy called *permissive upgrade* (PU) [6]. Differently from NSU, it allows the assignments of low variables under a high security context, but labels the updated variable as *partially-leaked* or 'P'. Intuitively, P means that the content of the variable is H but it may be L in other executions. If later in the execution, there is a branch on a variable marked with P, or such variable is to be output, the monitor stops the execution.

$$\textsc{assign}\ \frac{[\![e]\!]_\mu = v \qquad \Gamma' = \Gamma[x \mapsto \Gamma(e) \sqcup \mathsf{lift}(pc, \Gamma(x))]}{pc \vdash (\Gamma, x := e, \mu) \Downarrow_{\mathbf{PU}} (\Gamma', \mu[x \mapsto v])}$$

$$\textsc{if}\ \frac{\Gamma(x) \neq P \qquad [\![x]\!]_\mu = \alpha \qquad pc \sqcup \Gamma(x) \vdash (\Gamma, S_\alpha, \mu) \Downarrow_{\mathbf{PU}} (\Gamma', \mu')}{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}, \mu) \Downarrow_{\mathbf{PU}} (\Gamma', \mu')}$$

where

$$\mathsf{lift}(pc, l) = \begin{cases} L & \text{if } pc = L \\ H & \text{if } pc = H \wedge l = H \\ P & \text{if } pc = H \wedge l \neq H \end{cases}$$

Fig. 4: PU semantics

We present a permissive upgrade monitor (PU) for a two-point lattice extended with label P with $H \sqsubset P$. The semantics of PU is identical to the one of NSU (see Fig. 3) except for the ASSIGN and IF rules, that we present in Fig. 4. Rule ASSIGN behaves like the ASSIGN rule of NSU, if $pc \sqsubseteq \Gamma(x)$ and $\Gamma(x) \neq P$. Otherwise, the assigned variable is marked with P. Rule IF is similar to the rule IF in NSU, but the semantics gets stuck if the variable in the test condition is partially leaked. PU was proven to enforce TINI (see [6, Thm. 2]). However, PU is not TANI since it has the same mechanism as NSU for adding new termination channels.

Example 6 (PU is not TANI). Consider Program 1 and an initial memory [`h=1, l=0`]. PU does not satisfy TANI, since the monitor terminates only on one memory, i.e., $k_M(P, \mu_{\mathtt{L}}, v) =$ [`h=1, l=0`], while the original program terminates on both memories, low-equal to [`l=0`].

Hybrid Monitor (HM) Le Guernic *et al.* were the first to propose a *hybrid monitor* (HM) [14] for information flow control that combines static and dynamic analysis. This mechanism statically analyses the non-executed branch of each test in the program, collecting all the possibly updated variables in that branch. The security level of such variables are then raised to the level of the test, thus preventing information leakage.

Example 7. Consider Program 1 and its execution starting in [h=1, l=0]. This execution is modified by HM because the static analysis discovers that variable l could have been updated in a high security context in an alternative branch.

$$\textsc{assign}\ \frac{[\![e]\!]_\mu = v \qquad \Gamma' = \Gamma[x \mapsto pc \sqcup \Gamma(e)]}{pc \vdash (\Gamma, x := e, \mu) \Downarrow_{\mathbf{HM}} (\Gamma', \mu[x \mapsto v])}$$

$$\textsc{if}\ \frac{\begin{array}{c}\Gamma'' = \mathsf{Analysis}(S_{\neg\alpha}, pc \sqcup \Gamma(x), \Gamma) \\ [\![x]\!]_\mu = \alpha \qquad pc \sqcup \Gamma(x) \vdash (\Gamma, S_\alpha, \mu) \Downarrow_{\mathbf{HM}} (\Gamma', \mu')\end{array}}{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}, \mu) \Downarrow_{\mathbf{HM}} (\Gamma' \sqcup \Gamma'', \mu')}$$

$$\textsc{output}\ \frac{\Gamma(x) = L \Rightarrow v = [\![x]\!]_\mu \qquad \Gamma(x) \neq L \Rightarrow v = \mathsf{default}}{L \vdash (\Gamma, \mathsf{output}\ x, \mu) \Downarrow_{\mathbf{HM}} (v, \Gamma, \mu)}$$

Fig. 5: HM semantics

The semantics of HM is identical to NSU except for the ASSIGN, IF and OUTPUT rules that we show in Figure 5. The ASSIGN rule does not have any specific constraints. The static analysis $\mathsf{Analysis}(S, pc, \Gamma)$ in the IF rule explores variables assigned in S and upgrades their security level according to pc. We generalize the standard notation $\Gamma[x \mapsto l]$ to sets of variables and use $\mathsf{Vars}(S)$ for the sets of variables assigned in command S.

$$\mathsf{Analysis}(S, pc, \Gamma) = \Gamma[\{y \mapsto pc \sqcup \Gamma(y) \mid y \in \mathsf{Vars}(S)\}$$

HM was previously proven to enforce TINI [14, Thm. 1] and we prove in the companion technical report [1] that HM satisfies TANI.

Theorem 2. *HM is TANI.*

Secure Multi-Execution (SME) Devriese and Piessens were the first to propose secure multi-execution (SME) [12]. The main idea of SME is to execute the program multiple times: one for each security level. 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. Different executions are executed with a low priority scheduler to avoid leaks due to divergence of high executions because SME enforces TSNI.

Example 8 (SME "fixes" termination channels). Consider Program 4:

Program 4

```
1 if l = 0 then
2   while h=0 do skip;
3 else
4   while h=1 do skip;
5 output l
```

Assume $\mu_{\mathtt{L}} = [\mathtt{l{=}0}]$ and the default high value used by SME is `h=1`. Then, there exists a memory $\mu' = [\mathtt{h{=}0,\ l{=}0}]$, low-equal to $\mu_{\mathtt{L}}$, on which the original program doesn't terminate: $\mu' \notin \bigcup_{v'} k(P, \mu_{\mathtt{L}}, v')$, but SME terminates: $\mu' \in k_M(P, \mu_{\mathtt{L}}, \mathtt{l{=}0})$. Notice that SME makes the attacker's knowledge smaller.

$$\text{SME}\ \frac{(P, \mu\downarrow_\Gamma) \Downarrow (v, \mu') \qquad \mu''' = \begin{cases} \mu' \odot_\Gamma \mu'' & \text{if } \exists \mu''.(P,\mu) \Downarrow (v', \mu'') \\ \mu' \odot_\Gamma \bot & \text{otherwise} \end{cases}}{pc \vdash (\Gamma, P, \mu) \Downarrow_{\mathbf{SME}} (v, \Gamma, \mu''')}$$

where $\mu\downarrow_\Gamma (x) = \begin{cases} \mu(x) & \Gamma(x) = L \\ \mathsf{default} & \Gamma(x) = H \end{cases} \qquad \mu' \odot_\Gamma \mu''(x) = \begin{cases} \mu'(x) & \Gamma(x) = L \\ \mu''(x) & \Gamma(x) = H \end{cases}$

Fig. 6: SME semantics

The SME adaptation for our while language is given in Figure 6, with executions for levels L and H. The special value $\bot$ represents the idea that no value can be observed and we overload the symbol to also denote a memory that maps every variable to $\bot$. Using memory $\bot$ we simulate the low priority scheduler of SME in our setting: if the execution corresponding to the H security level does not terminate, the SME semantics still terminates. In this case all the variables with level H, which values should correspond to values obtained in the normal execution of the program, are given value $\bot$.

SME was previously proven TSNI [12, Thm. 1] and we prove that it also enforces TANI: this can be directly inferred from our Theorem 1.

Theorem 3. *SME is TANI.*

Multiple Facets Austin and Flanagan proposed multiple facets (MF) in [7]. 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 from the point of view of observers at different security levels. The main idea in MF is that 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.

Example 9. Consider the TINI Program 5. In MF, the output observable at level L (or the L facet of variable `l`) is always the initial value of variable `l` since MF will not update a low variable in a high context. Therefore, all the executions of Program 5 starting with `l=1` are modified by MF, producing the output `l=1`.

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

Our adaptation of MF semantics is given in Figure 7 where we use the following notation: a faceted value, denoted $\langle v_1 : v_2 \rangle$, is a pair of values v_1 and v_2.

$$\text{MF Rule}\ \boxed{\frac{pc \vdash (\Gamma, P, \mu \uparrow_\Gamma) \downarrow_{MF} (\langle v_1 : v_2\rangle, \Gamma', \hat{\mu})}{pc \vdash (\Gamma, P, \mu) \Downarrow_{\mathbf{MF}} (v_2, \Gamma', \hat{\mu} \downarrow_{\Gamma'})}}\ \text{SKIP}\ \frac{}{pc \vdash (\Gamma, \mathsf{skip}, \hat{\mu}) \downarrow_{MF} (\Gamma, \hat{\mu})}$$

$$\text{ASSIGN}\ \frac{[e]\hat{\mu} = \langle v_1 : v_2\rangle \qquad \hat{v} = \begin{cases} \langle v_1 : \hat{\mu}(x)_2\rangle & \text{if } pc = H \wedge \Gamma(x) = L \\ \langle v_1 : v_2\rangle & \text{if } pc = L \vee \Gamma(x) \neq L \end{cases} \qquad \Gamma'(y) = \begin{cases} \Gamma(e) & \text{if } pc = L \wedge y = x \\ \Gamma(y) & \text{otherwise} \end{cases}}{pc \vdash (\Gamma, x := e, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}[x \mapsto \hat{v}])}$$

$$\text{SEQ}\ \frac{pc \vdash (\Gamma, S_1, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}') \qquad pc \vdash (\Gamma', S_2, \hat{\mu}') \downarrow_{MF} (\Gamma'', \hat{\mu}'')}{pc \vdash (\Gamma, S_1; S_2, \hat{\mu}) \downarrow_{MF} (\Gamma'', \hat{\mu}'')}$$

$$\text{IF-HIGH}\ \frac{[x]\hat{\mu} = \langle \alpha_1 : \alpha_2\rangle \qquad pc = H \vee \Gamma(x) = H \qquad H \vdash (\Gamma, S_{\alpha_1}, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}')}{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}')}$$

$$\text{IF-LOW}\ \frac{[x]\hat{\mu} = \langle \alpha_1 : \alpha_2\rangle \qquad pc = L \wedge \Gamma(x) = L \qquad L \vdash (\Gamma, S_{\alpha_1}, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}_1) \qquad L \vdash (\Gamma, S_{\alpha_2}, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}_2)}{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}, \mu) \downarrow_{MF} (\Gamma', \hat{\mu}_1 \otimes_\Gamma \hat{\mu}_2)}$$

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

$$\text{OUTPUT}\ \frac{[x]\hat{\mu} = \hat{v}}{L \vdash (\Gamma, \mathsf{output}\ x, \hat{\mu}) \downarrow_{MF} (\hat{v}, \Gamma, \hat{\mu})}$$

where

$$[\hat{v}]\hat{\mu} = \hat{v}, \quad [x]\hat{\mu} = \hat{\mu}(x)$$

$$[e_1 \oplus e_2]\hat{\mu} = \langle v_1 \oplus v_2 : v_1' \oplus v_2'\rangle, \text{ where } [e_1]\hat{\mu} = \langle v_1 : v_1'\rangle, [e_2]\hat{\mu} = \langle v_2 : v_2'\rangle$$

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

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

Fig. 7: Multiple Facets semantics

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 $\langle v : v\rangle$. Faceted memories, ranged over $\hat{\mu}$, are mappings from variables to faceted values. 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. As in SME, the special value $\bot$ represents the idea that no value can be observed. MF was previously proven TINI [7, Thm. 2] and we prove that it satisfies TANI.

Theorem 4. *MF is TANI.*

6 Precision, permissiveness and transparency

A number of works on dynamic information flow monitors try to analyse precision of monitors. Intuitively, precision describes how often a monitor blocks (or modifies) secure programs. Different approaches have been taken to compare precision of monitors, using definitions such as "precision", "permissiveness" and "transparency". We propose a rigorous comparison of these definitions.

In the field of runtime monitoring, a monitor should provide two guarantees while enforcing a security property: soundness and transparency. *Transparency* [8] means that whenever an execution satisfies a property in question, the monitor should output it without modifications[3].

Precision (versus well typed programs) Le Guernic et al. [21] were among the first to start the discussion on transparency for information flow monitors. The authors have proved that their hybrid monitor accepts all the executions of a program that is well typed under a flow-insensitive type system similar to the one of Volpano et al. [31]. Le Guernic [19] names this result as *partial transparency*. Russo and Sabelfeld [25] prove a similar result: they show that a hybrid monitor accepts all the executions of a program that is well typed under the flow-sensitive type system of Hunt and Sands [18].

Precision (versus secure programs) Devriese and Piessens [12] propose a stronger notion, called *precision*, that requires a monitor to accept all the executions of all secure programs. Notice that this definition is stronger because not only the monitor should recognise the executions of well typed programs, but also of secure programs that are not well typed. Devriese and Piessens have proven that such precision guarantee holds for SME versus TSNI programs.

Transparency (versus secure executions) As a follow-up, Zanarini et al. [32] have proven that another monitor based on SME satisfies *transparency for TSNI*. This monitor accepts all the TSNI executions of a program, even if the program itself is insecure.

Permissiveness (versus executions accepted by other monitors) In his PhD thesis, Le Guernic [19] compares his hybrid monitor with another hybrid monitor that performs a more precise static analysis, and proves an *improved precision* theorem stating that whenever the first hybrid monitor accepts an execution, the second monitor accepts it as well. Following this result, Besson et al. [9] investigate other hybrid monitors and prove relative precision in the style of Le Guernic, and Austin and Flanagan [6, 7] use the same definition to compare their dynamic monitors. Hedin et al. [16] name the same notion by *permissiveness* and compare the sets of accepted executions: one monitor is more permissive than another one if its set of accepted executions contains a set of accepted executions of the other monitor.

To compare precision of different information flow monitors, we propose to distinguish two notions of transparency. *True transparency* defines the secure

[3] Bauer et al. [8] actually provide a more subtle definition, saying a monitor should output a semantically equivalent trace.

executions accepted by a monitor, and *false transparency* defines the insecure executions accepted by a monitor.

True Transparency We define a notion of *true transparency* for TINI. Intuitively, a monitor is true transparent if it accepts all the TINI executions of a program.

Definition 8 (True Transparency). *Monitor M is* true transparent *if for any program P, and any memories μ, μ' and output v, the following holds:*

$$TINI(P, \mu_{\mathtt{L}}) \wedge (P, \mu) \Downarrow (v, \mu') \Rightarrow (P, \mu) \Downarrow_M (v, \mu')$$

There is a well-known result that a truncation automata cannot recognise more than computable safety properties [15, 28]. Since noninterference can be reduced to a safety property that is not computable [29], and NSU and PU can be modeled by truncation automata, it follows that they are not true transparent. We show that the monitors of this paper, that cannot be modeled by truncation automata, are not true transparent for TINI neither.

Example 10 (HM is not true transparent). Consider Program 5: it always terminates with `l=0` and hence it is secure. Any execution of this program will be modified by HM because `l` will be marked as high.

Example 11 (MF is not true transparent). Consider again TINI Program 5. The MF semantics will not behave as the original program semantics upon an execution starting in [`h=1, l=1`]. The sensitive upgrade of the test will assign faceted value [`l=`$\langle 0 : 1\rangle$] to variable `l` and the output will produce the low facet of `l` which is 1, while the original program would produce an output 0. Hence, this is a counter example for true transparency of MF.

Example 12 (SME is not true transparent for TINI). Since SME enforces TSNI, it eliminates all the termination channels, therefore even if the original program has TINI executions, SME might modify them to achieve TSNI.

Consider TINI Program 4 and an execution starting in [`h=0,l=1`]. SME (with default value `h=1`) will diverge because it's "low" execution will diverge upon `h=1`. Therefore, SME is not true transparent for TINI.

Even though none of the considered monitors are true transparent for TINI, this notion allows us to define a relative true transparency to better compare the behaviours of information flow monitors when they deal with secure executions.

Given a program P and a monitor M, we define a set of initial memories that lead to secure terminating executions of program P, and a monitor M does not modify these executions:

$$\mathcal{T}(M, P) = \{\mu \mid TINI(P, \mu_{\mathtt{L}}) \wedge \exists \mu', v.\ (P, \mu) \Downarrow (v, \mu') \Rightarrow (P, \mu) \Downarrow_M (v, \mu')\}$$

Definition 9 (Relative True Transparency). *Monitor A is* more true transparent *than monitor B, written $A \sqsupseteq_{\mathcal{T}} B$, if for any program P, the following holds: $\mathcal{T}(A, P) \supseteq \mathcal{T}(B, P)$.*

Austin and Flanagan [5, 6] have proven that MF is more true transparent than PU and PU is more true transparent than NSU. We restate this result in our notations and provide a set of counterexamples showing that for no other couple of analysed monitors relative true transparency holds.

Theorem 5. $MF \supseteq_{\mathcal{T}} PU \supseteq_{\mathcal{T}} NSU$.

Example 13 ($NSU \not\supseteq_{\mathcal{T}} PU, NSU \not\supseteq_{\mathcal{T}} HM$). Consider TINI Program 3: an execution in initial memory with [`h=0`] is accepted by PU and HM because the security level of `l` becomes low just before the output, and it is blocked by NSU due to sensitive upgrade.

Example 14 ($NSU \not\supseteq_{\mathcal{T}} SME, NSU \not\supseteq_{\mathcal{T}} MF, PU \not\supseteq_{\mathcal{T}} HM$, $PU \not\supseteq_{\mathcal{T}} SME$ and $PU \not\supseteq_{\mathcal{T}} MF$). Program 6 is TINI since `l'` does not depend on `h`. With initial memory [`h=0, l=1`], HM, SME (with default value chosen as 0) and MF terminate with the same output as normal execution. However, NSU will diverge due to sensitive upgrade and PU will diverge because of the branching over a partially-leaked variable `l`.

Program 6

```
1 if h = 0 then l = 1;
2 if l = 1 then l = 0;
3 output l';
```

Example 15 ($HM \not\supseteq_{\mathcal{T}} NSU, HM \not\supseteq_{\mathcal{T}} PU, HM \not\supseteq_{\mathcal{T}} SME, HM \not\supseteq_{\mathcal{T}} MF$). Consider Program 1 and its secure execution starting in [`h=1, l=1`]. NSU, PU, SME (the default value of SME does not matter in this case) and MF terminate with the same output as original program execution, producing `l=1`. However, HM modifies it because the security level of `l` is raised by the static analysis of the non-executed branch.

Example 16 ($SME \not\supseteq_{\mathcal{T}} NSU, SME \not\supseteq_{\mathcal{T}} PU, SME \not\supseteq_{\mathcal{T}} HM, SME \not\supseteq_{\mathcal{T}} MF$). All the terminating executions of TINI Program 4 are accepted by NSU, PU, HM and MF, while an execution starting in [`h=0, l=1`] with default value for SME set to `h=1` doesn't terminate in SME semantics.

Example 17 ($MF \not\supseteq_{\mathcal{T}} HM$). Program 7 is TINI for any execution. HM with [`h=1,l=0,l'=0`] terminates with the original output because the output variable [`l'`] is low. However, MF with [`h=1,l=0,l'=0`] doesn't terminate.

Program 7

```
1 if h=0 then l=0 else l=1;
2 if l=0 then
3   while true do skip;
4 else
5   l=0
6 output l'
```

Example 18 ($MF \not\supseteq_{\mathcal{T}} SME$). Program 5 is TINI for any execution. With [`h=0, l=1`] it terminates in the program semantics and SME semantics (with any default value) producing `l=0`. However, the MF semantics produces `l=1`.

Precision We have discovered that certain monitors (e.g., HM and NSU) are incomparable with respect to true transparency. To compare them, we propose a more coarse-grained definition that describes the monitors' behaviour on secure programs.

Definition 10 (Precision). *Monitor M is* precise *if for any program P, the following holds:*

$$TINI(P) \wedge \forall \mu.(\exists \mu', v.(P, \mu) \Downarrow (v, \mu') \Rightarrow (P, \mu) \Downarrow_M (v, \mu'))$$

This definition requires that all the executions of secure programs are accepted by the monitor. NSU, PU, HM and MF are not precise since they are not true transparent. SME is precise for TSNI, and this result was proven by Devriese and Piessens [12], however SME it not precise for TINI (see Example 12).

To compare monitors' behaviour on secure programs, we define a set of a TINI programs P, where a monitor accepts all the executions of P:

$$\mathcal{P}(M) = \{P \mid TINI(P) \wedge \forall \mu.(\exists \mu', v.(P, \mu) \Downarrow (v, \mu') \Rightarrow (P, \mu) \Downarrow_M (v, \mu'))\}$$

Definition 11 (Relative Precision). *Monitor A is more precise than monitor B, written $A \supseteq_{\mathcal{P}} B$, if $\mathcal{P}(A) \supseteq \mathcal{P}(B)$.*

We have found out that no couple of the five monitors are in relative precision relation. Below we present the counterexamples that demonstrate our findings.

Example 19 ($HM \not\supseteq_{\mathcal{P}} SME$). Consider TINI Program 5. All the executions of this program are accepted by SME. However, HM modifies the program output to default because the security level of `l` is upgraded to H by the static analysis of the non-executed branch.

Example 20 ($HM \not\supseteq_{\mathcal{P}} NSU$, $HM \not\supseteq_{\mathcal{P}} PU$). Consider the following program:

Program 8

```
1 l = 0;
2 if h = 0 then skip
3 else
4   while true do l = 1;
5 output l
```

This TINI program terminates only when [`h=0`]. This execution is accepted by NSU and PU, but the program output is modified by HM since HM analyses the non-executed branch and upgrades the level of `l` to H.

Example 21 ($HM \not\supseteq_{\mathcal{P}} MF$). Consider TINI Program 9. MF accepts all of its executions, while HM modifies the program output to default because the security level of `l` is raised to high.

Program 9

```
1 l = 0;
2 if h = 0 then l = 0 else skip;
3 output l
```

The rest of relative precision counterexamples demonstrated in Table 1 of Section 8 are derived from the corresponding counterexamples for relative true transparency.

Since relative precision does not hold for any couple of monitors, we propose a stronger definition of relative precision for TSNI programs. We first define a set of a TSNI programs P, where a monitor accepts all the executions of P:

$$\mathcal{P}^*(M) = \{P \mid TSNI(P) \wedge \forall \mu.(\exists \mu', v.(P,\mu) \Downarrow (v,\mu') \Rightarrow (P,\mu) \Downarrow_M (v,\mu'))\}$$

Definition 12 (Relative TSNI precision). *A monitor A is more TSNI precise than a monitor B, written $A \supseteq^*_{\mathcal{P}} B$, if $\mathcal{P}^*(A) \supseteq \mathcal{P}^*(B)$.*

Theorem 6. *For all programs without dead code, $HM \supseteq^*_{\mathcal{P}} NSU, HM \supseteq^*_{\mathcal{P}} PU$.*

Notice that SME was proven to be precise for TSNI programs (see [12, Thm. 2]), therefore SME is more TSNI precise than any other monitor. We demonstrate this in Table 1 of Section 8.

False Transparency To compare monitors with respect to the amount of insecure executions they accept, we propose the notion of *false transparency*. Notice that false transparency violates soundness.

Definition 13 (False Transparency). *Monitor M is* false transparent *if for any program P, for all executions starting in a memory μ and finishing in memory μ' with value v, the following holds:*

$$\neg TINI(P,\mu) \wedge (P,\mu) \Downarrow (v,\mu') \Rightarrow (P,\mu) \Downarrow_M (v,\mu').$$

Given a program P and a monitor M, we define a set of initial memories, where a program P terminates, and a monitor M is false transparent for P:

$$\mathcal{F}(M,P) = \{\mu \mid \neg TINI(P,\mu_{\mathtt{L}}) \wedge \exists \mu', v.(P,\mu) \Downarrow (v,\mu') \Rightarrow (P,\mu) \Downarrow_M (v,\mu')\}$$

Definition 14 (Relative False Transparency). *Monitor A is* more false transparent *than monitor B, denoted $A \supseteq_{\mathcal{F}} B$, if for any program P, the following holds: $\mathcal{F}(A,P) \supseteq \mathcal{F}(B,P)$.*

Theorem 7. *The following statements hold: $NSU \supseteq_{\mathcal{F}} HM$, $PU \supseteq_{\mathcal{F}} NSU$, $PU \supseteq_{\mathcal{F}} HM$, $SME \supseteq_{\mathcal{F}} HM$, $MF \supseteq_{\mathcal{F}} NSU$, $MF \supseteq_{\mathcal{F}} PU$ and $MF \supseteq_{\mathcal{F}} HM$.*

Example 22 ($NSU \not\supseteq_{\mathcal{F}} PU$). Execution of Program 10 in the initial memory μ= [`h=0, l=0, l'=0`] is interfering since it produces an output `l=0`, while an execution in the low-equal initial memory where [`h=1`] produces `l=1`. An execution started in μ is accepted by PU but blocked by NSU.

```
1 if h = 0 then l' = 1 else l = 1;
2 output l
```
Program 10

Example 23 (NSU $\not\supseteq_{\mathcal{F}}$ SME, PU $\not\supseteq_{\mathcal{F}}$ SME). Execution of Program 11 starting in memory [h=0, l=0] is not TINI and it is accepted by SME (with default value h=0). However, it is rejected by NSU because of sensitive upgrade and by PU because on the branching over a partially-leaked variable l.

```
1 if h = 0 then l = 0 else l = 1;
2 if l = 0 then l' = 0 else l' = 1;
3 output l'
```
Program 11

Example 24 (NSU $\not\supseteq_{\mathcal{F}}$ MF). The following program always terminates in the normal semantics coping the value of h into l. Hence all of its executions are insecure. Every execution leads to a sensitive upgrade and NSU will diverge with any initial memory. However, in the MF semantics the program will terminate with l=0 if started with memory [h=0,l=0] since the sensitive upgrade of the true branch will assign faceted value [l=$\langle 0 : 0\rangle$] to variable l. Hence, this is a counter example for NSU being more false transparent than MF.

```
1 if h=0 then l=0 else l=1;
2 output l
```
Program 12

Example 25 (PU $\not\supseteq_{\mathcal{F}}$ MF). Program 11 is not TINI for all executions. However MF with [h=1,l=1,l'=1] terminates in the same memory as normal execution, while PU will diverge because l is marked as a partial leak.

Example 26 (HM $\not\supseteq_{\mathcal{F}}$ NSU, HM $\not\supseteq_{\mathcal{F}}$ PU, HM $\not\supseteq_{\mathcal{F}}$ SME, HM $\not\supseteq_{\mathcal{F}}$ MF). Consider Program 1 and an execution starting in memory [h=1, l=0]. This execution is not secure and it is rejected by HM, however NSU, PU and MF accept it. SME also accepts this execution in case the default value for h is 1.

Example 27 (SME $\not\supseteq_{\mathcal{F}}$ NSU, SME $\not\supseteq_{\mathcal{F}}$ PU). Execution of Program 13 starting in memory [h=0, l=0] is interfering and it is accepted by both NSU and PU, producing an output l=0. However, SME (with default value chosen as 1) modifies this execution and produces l=1.

```
1 if l = 0 then
2   if h = 1 then l = 1 else skip
3 else
4   if h = 0 then l = 0 else skip
5 output l
```
Program 13

Example 28 (SME $\not\supseteq_{\mathcal{F}}$ MF and MF $\not\supseteq_{\mathcal{F}}$ SME). Program 14 is not TINI if possible values of h are 0, 1, and 2. MF with [h=1,l=1] terminates in the same memory than normal execution but SME (with default value 0) always diverges.

```
1 if h = 0 then                                          Program 14
2    while true do skip;
3 else
4   if h=1 then l=1 else l=2;
5 output l;
```

On the other hand, with initial memory [`h=1, l=0`], SME (using default value 1) terminates in the same memory as the normal execution, producing `l=1` but MF produces a different output `l=0`.

7 Related Work

In this section, we discuss the state of the art for taxonomies of information flow monitors with respect to soundness or transparency.

For soundness, no work explicitly tries to classify information flow monitors. However, it is folklore that TSNI, first proposed in [30], is a strong form of non-interference that implies TINI. Since most well-known information flow monitors are proven sound only for TINI [5–7, 14, 33], it is easy, from the soundness perspective, to distinguish SME from other monitors because SME is proven sound for TSNI [12]. However, to the best of our knowledge, no work tries to refine soundness in order to obtain a more fine grain classification of monitors as we achieve with the introduction of TANI.

For transparency, Devriese and Piessens [12] prove that SME is precise for TSNI and Zanarini et al. [32] notice that the result could be made more general by proving that SME is true transparent for TSNI, which makes of SME an effective enforcement [22] for TSNI. In this work, we first compare transparency for TINI: none of the monitors that we have studied is true transparent for TINI. Hedin et al. [16] compare hybrid (HM) and purely dynamic monitors (NSU and PU), and conclude that for these monitors permissiveness is incomparable. By factorizing the notion of permissiveness, we can compare HM and NSU: HM is more precise for TSNI than NSU and PU, and NSU and PU are more false transparent than HM. Using the same definition of permissiveness, Austin and Flanagan [6,7] prove that PU is more permissive than NSU and that MF is more permissive than PU. Looking at this result and the definition of MF, our intuition was that MF could accept exactly the same false transparent executions as NSU and PU. However, we discovered that not only MF is more true transparent than NSU and PU (this is an implication of Austin and Flanagan results) but also MF is strictly more false transparent than NSU and PU. Bichhawat et al. [10] propose two non-trivial generalizations of PU, called puP and puA, to arbitrary lattices and show that puP and puA are incomparable w.r.t. permissiveness. It remains an open question if puP and puA can be made comparable by discriminating true or false transparency, as defined in our work.

	NSU	PU	HM	SME	MF
NSU		$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$
PU	$\supseteq_{\mathcal{T}}$ $\supseteq_{\mathcal{F}}$		$\not\supseteq_{\mathcal{P}} \supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$
HM	$\supseteq^*_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\supseteq^*_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$		$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$
SME	$\supseteq^*_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\supseteq^*_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\supseteq^*_{\mathcal{P}} \supseteq_{\mathcal{F}}$		$\supseteq^*_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$
MF	$\supseteq_{\mathcal{T}} \supseteq_{\mathcal{F}}$	$\supseteq_{\mathcal{T}} \supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	

$\supseteq_{\mathcal{T}}$ more true TINI transparent than
$\supseteq_{\mathcal{P}}$ more TINI precise than ($\not\supseteq_{\mathcal{P}} \implies \not\supseteq_{\mathcal{T}}$)
$\supseteq^*_{\mathcal{P}}$ more TSNI precise than
$\supseteq_{\mathcal{F}}$ more false TINI transparent than
Monitor is TANI
Monitor is TSNI, hence TANI

Table 1: **Taxonomy of five major information flow monitors**

8 Conclusion

In this work we proposed a new soundness definition for information flow monitors, that we call *Termination-Aware Noninterference* (TANI). It determines whether a monitor adds a new termination channel to the program. We have proven that HM, SME and MF, do satisfy TANI, whereas NSU and PU introduce new termination channels, and therefore do not satisfy TANI.

We compare monitors with respect to their capability to recognise secure executions, i.e., true transparency [8]. Since it does not hold for none of the considered monitors, we weaken this notion and define *relative true transparency*, that determines "which monitor is closer to being transparent". We then propose even a more weaker notion, called *precision*, that compares monitor behaviours on secure programs, and allows us to conclude that HM is more TSNI precise than NSU and PU that previously were deemed incomparable [16]. We show that the common notion of permissiveness is composed of *relative true and false transparency* and compare all the monitors with respect to these notions in Table 1.

For simplicity, we consider a security lattice of only two elements, however we expect our results to generalise to multiple security levels. In future work, we plan to compare information flow monitors with respect to other information flow properties, such as declassification [26].

Acknowledgment

We would like to thank Ana Almeida Matos for her valuable feedback and interesting discussions that has lead us to develop the main ideas of this paper, Aslan Askarov for his input to the definition of TANI, and anonymous reviewers for feedback that helped to improve this paper. This work has been partially supported by the ANR project AJACS ANR-14-CE28-0008.

References

1. A Taxonomy of Information Flow Monitors Technical Report. `https://team.inria.fr/indes/taxonomy`.

2. M. Abadi and L. Lamport. Composing Specifications. *ACM Transactions on Programmming Languages and Systems.*, 1993.
3. A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symposium on Security and Privacy*, pages 207–221, 2007.
4. A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 2009 22Nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 43–59. IEEE Computer Society, 2009.
5. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS'09*, pages 113–124, 2009.
6. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS'10*, pages 3:1–3:12. ACM, 2010.
7. 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.
8. 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.
9. F. Besson, N. Bielova, and T. Jensen. Hybrid information flow monitoring against web tracking. In *CSF'13*, pages 240–254. IEEE, 2013.
10. A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Generalizing permissive-upgrade in dynamic information flow analysis. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*, PLAS'14, pages 15:15–15:24. ACM, 2014.
11. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 2010.
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. G. L. Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Proc. of the 11th Asian Computing Science Conference (ASIAN'06)*, volume 4435, pages 75–89. Springer-Verlag Heidelberg, 2006.
15. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, 2006.
16. 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.
17. D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proc. of the 25th Computer Security Foundations Symposium*, pages 3–18. IEEE, 2012.
18. S. Hunt and D. Sands. On flow-sensitive security types. In *POPL'06*, pages 79–90, New York, NY, USA, Jan. 2006. ACM.
19. G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University and University of Rennes 1, 2007.
20. G. Le Guernic. Precise Dynamic Verification of Confidentiality. In *Proc. of the 5th International Verification Workshop*, volume 372 of *CEUR Workshop Proc.*, pages 82–96, 2008.
21. G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based Confidentiality Monitoring. In *Proc. of the Annual Asian Computing Science Conference*, volume 4435 of *LNCS*, pages 75–89. Springer, 2006.

22. J. Ligatti, L. Bauer, and D. Walker. Enforcing Non-Safety Security Policies with Program Monitors. In *ESORICS 05*, 2005.
23. 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.
24. J. McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 1996.
25. A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proc. of the 23rd Computer Security Foundations Symposium*, pages 186–199. IEEE, 2010.
26. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
27. 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.
28. F. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
29. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Static Analysis, 12th International Symposium*, pages 352–367, 2005.
30. D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *In Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168. Society Press, 1997.
31. D. Volpano, G. Smith, and C. Irvine. A Sound Type System For Secure Flow Analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.
32. 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.
33. S. A. Zdancewic. *Programming languages for information security.* PhD thesis, Cornell University, 2002.