A Better Facet of Dynamic Information Flow Control

N.B. When citing this work, cite the original published paper.
A Better Facet of Dynamic Information Flow Control

Minh Ngo  
INRIA, France  
nguyen-nhat-minh.ngo@inria.fr

Nataliia Bielova  
INRIA, France  
nataliia.bielova@inria.fr

Cormac Flanagan  
UCSC, USA  
cormac@ucsc.edu

Tamara Rezk  
INRIA, France  
tamara.rezk@inria.fr

Alejandro Russo  
Chalmers University of Technology, Sweden  
russo@chalmers.se

Thomas Schmitz  
UCSC, USA  
tschmitz@ucsc.edu

ABSTRACT
Multiple Facets (MF) is a dynamic enforcement mechanism which has proved to be a good fit for implementing information flow security for JavaScript. It relies on multi executing the program, once per each security level or view, to achieve soundness. By looking inside programs, MF encodes the views to reduce the number of needed multi-executions.

In this work, we extend Multiple Facets in three directions. First, we propose a new version of MF for arbitrary lattices, called Generalised Multiple Facets, or GMF. GMF strictly generalizes MF, which was originally proposed for a specific lattice of principals. Second, we propose a new optimization on top of GMF that further reduces the number of executions. Third, we strengthen the security guarantees provided by Multiple Facets by proposing a termination sensitive version that eliminates covert channels due to termination.

1 INTRODUCTION
JavaScript has become the de facto programming language of the Web. Web browsers daily execute thousands of JavaScript lines which usually have access to confidential information, for example cookies that mark that the user in a web session is authenticated.

It is not surprising that JavaScript is a common target for attacks. It is not surprising that JavaScript is a common target for attacks. While browsers deploy security measures in the form of access control (e.g., SOP and CSP), they are insufficient [12, 17, 30] to protect confidentiality of data.

Information flow control (IFC) is a promising technology which provides a systematic solution to handle unintentional or malicious leaks of confidential information. Recently, dynamic IFC analyses have received a lot of attention [1–3, 5, 7, 9, 10, 14, 26, 33], due, in part, to its applicability to JavaScript—where static analyses are rather an awkward fit [29].

In order to scale, a suitable IFC technique for the web not only needs to be dynamic but also needs to reduce to the minimum the modifications required to existing JavaScript code. In this light, an interesting dynamic IFC technique which fulfills both of these requirements consists in executing several copies of a program: one execution per each security level or view. In that manner, each copy of the program (view) depends only on information observable to the corresponding security level, where no leaks are therefore possible. Secure Multi Execution (SME) [14] and Multiple Facets (MF) [3] are two techniques based on this idea.

Both techniques have been proved to be a good fit for information flow security in the web since they have been successfully implemented as extensions of the Firefox browser [13, 33]. Although both SME and MF are based on multi-executions, they present important differences [7]. On one hand, SME is black-box [24], i.e., it is a mechanism that does not look inside programs but rather change the semantics of inputs and outputs to ensure security. For a moment, we assume a scenario where security levels are simply sets of principals (e.g., web origins) which denote those authorities with confidentiality concerns over data. In such a scenario, SME needs to spawn one execution for any possible set of principals—where the number of executions grows exponentially with respect to the number of principals! Instead, MF [3] is designed to reduce the number of multi-executions and the memory footprint of SME. It does so by inspecting programs code and multi-executing instructions and multiplexing memory only when needed. While MF is more resource-friendly than SME, SME provides stronger security guarantees when it comes to leaks via abnormal termination [7].

Our broad goal is to augment the efficiency of techniques based on MF and SME to general cases. In particular, we discovered that MF might sometimes spawn more multi-executions than SME—something that is counter-intuitive when considering the purpose of MF (see Section 2). Our first contribution consists on a novel technique to further reduce the number of multi-executions (and memory footprint) of MF. Our second contribution is to generalize MF to work for arbitrary finite lattices (see Section 3) rather than being restricted to the security lattice of principals as in the original proposal [3]. This becomes useful when, for instance, a program depends on 5 security levels. In such case, as stated originally, MF will need to encode them by using (at least) 3 principals ($2^3 > 5$), and thus execute the program $2^3 = 8$ times, while SME will execute it only 5 times (one per security level). Finally, we combine MF and SME into a single new dynamic IFC mechanism in order to provide security guarantees as strong as SME (i.e., termination sensitive
2 BACKGROUND ON SME AND MF

In this section, we discuss how on one hand, the underpinning mechanism in MF reduces the number of executions compared to SME, and on the other hand, may run more multi executions than SME because of the security lattice based on principals. Our goal here is partly pedagogical and partly to motivate and provide intuition on the optimization proposed in Section 4.

Language and Semantics To investigate the foundation of multiple facets, we use a simple, deterministic while language. Its syntax includes programs \( P \), variables \( x \), expressions \( e \), and values \( v \). We use the symbol \( \oplus \) for binary expression operators. A value is either an integer value or a boolean value.

\[
\begin{align*}
(\text{programs}) & \quad P ::= \text{skip} \mid x := e \mid \text{if } e \text{ then } P_1 \text{ else } P_2 \\
(\text{expressions}) & \quad e ::= v \mid x \mid e \oplus e
\end{align*}
\]

Figure 1 presents standard big-step semantics of the language. Memories \( \mu \) map variables to values; we overload the notation of memory and use \( \mu(e) \) as the evaluation function for expression \( e \) in memory \( \mu \), where \( \mu(v) = v \) and \( \mu(e_1 \oplus e_2) = \mu(e_1) \oplus \mu(e_2) \). We write \( (P, \mu) \Downarrow \mu' \) to mean that the evaluation of program \( P \) on memory \( \mu \) terminates with memory \( \mu' \). We use \( \mu[x \mapsto v] \) for the memory \( \mu' \) where \( \mu'(y) = \mu(y) \) if \( y \neq x \), and \( \mu'(y) = v \) if \( y = x \).

MF may use fewer resources than SME SME [14] multi executes programs, in a blackbox manner, as many times as security levels in a lattice. Let’s define an SME memory as a function that maps each variable to an array of values, one value per security level. For the sake of simplicity, let’s consider first a security lattice with only two elements \( H \) and \( L \) where \( H \not \subseteq L \) is the only disallowed flow. Thus, an SME memory \( \hat{\mu} \) maps variables to an array of 2 (possibly different) values: one corresponding to the \( H \) view and one corresponding to the \( L \) view. Let’s denote such array of values as \( \langle v_1 : v_2 \rangle \), where \( v_1 \) is a private, \( H \) view and \( v_2 \) is a public, \( L \) view. Assume that \( H(\hat{\mu}) \) (resp. \( L(\hat{\mu}) \)) is a memory in the standard semantics, obtained by projection of \( \hat{\mu} \), mapping variables to single values of the high view (resp. low view). Then, the SME monitoring rule\(^1\) for such a language can be given by the relation \( \hat{\mu} \Downarrow \hat{\mu} \) as follows:

\[
\begin{align*}
\text{SKIP} & \quad \Downarrow (\text{skip} \mu) \\
\text{ASSIGN} & \quad (x := e, \mu) \Downarrow (x := e, \mu[x \mapsto v]) \\
\text{IF} & \quad (\text{if } e \text{ then } P_1 \text{ else } P_2) \Downarrow (P_1, \mu) \Downarrow (P_2, \mu') \Downarrow \mu'' & \text{SEQ} & \quad (P_1, \mu) \Downarrow (P_2, \mu') \Downarrow \mu''
\end{align*}
\]

Figure 1: Language semantics

non-interference) while avoiding multi-executions as much as our optimized version of MF allows it. All proofs can be found in [23].

\[\Downarrow \hat{\mu} \Downarrow \hat{\mu} \]

where \( \hat{\mu} \Downarrow \hat{\mu} \) combines two normal memories into a SME memory in such a way that \( H(\mu_1 \hat{\hat{\mu}}) \Downarrow \mu_1 \) and \( L(\mu_2 \hat{\hat{\mu}}) \Downarrow \mu_2 \). The SME mechanism will blindly execute the program as many times as possible views (or positions of the array) may exist.

Consider a program \( h := l \) where initial views for variables \( l \) and \( h \) are given by: \( \hat{\mu}(h) = (1 : 0) \) and \( \hat{\mu}(l) = (1 : 1) \). In SME, using the SME-TINI rule, the assignment will be executed twice: once with \( H(\hat{\mu}) = [h \mapsto 1, l \mapsto 1] \) for the high view and once with \( L(\hat{\mu}) = [h \mapsto 0, l \mapsto 1] \) for the low view. After execution, the final SME memory will map \( h \) to \( (1 : 1) \). One way to reduce the number of executions is to exploit the knowledge that the high and the low view for variable \( l \) are equal, i.e., \( H(\hat{\mu})(l) = L(\hat{\mu})(l) \). Since the semantics is deterministic, there is no need to execute the program twice. We can use this knowledge by specialising SME at the granularity of commands and include the following assignment rule:

\[
\text{SME-optm} \quad \Downarrow \hat{\mu}(l) \Downarrow \hat{\mu}(l) \quad \Downarrow (x := e, L(\hat{\mu})) \Downarrow (x := e, \hat{\mu}[x \mapsto (\mu(x), \mu(x))])
\]

Notice that this SME optimization requires to look inside the shape of the program to evaluate if expression \( e \) of an assignment satisfies the hypothesis.

In general, in order to reduce the number of executions using the multi-execution technique of SME-TINI, it is sufficient to (i) identify in an SME memory which values in the array of values are equal and (ii) remember which values correspond to which views. MF uses the multi-execution technique, implements (i) and (ii) and hence, reduces the number of executions. MF encodes values in SME memories (arrays with as many positions as lattice elements) as ordered binary trees, where the order is given by the elements of the lattice. For example, for a SME memory where \( \hat{\mu}(h) = (1 : 0 : 0 : 0) \) for a lattice of 4 elements with top element \( T \), an equivalent MF memory encodes this array as \( (T 1 0 : 0) \) with the meaning that 1 is the view for \( T \) and 0 for the rest. Every execution that depends on that value, will multi execute twice instead of 4 times as in SME.

Moreover, MF further uses the view information provided by the encoding in order to multi execute less in case of branching commands. For example, for SME-TINI with SME memory \( \hat{\mu}(h) = (1 : 0 : 0 : 0) \) the program:

\[
\begin{align*}
1 & \quad \text{if } h = 0 \text{ then} \\
2 & \quad h := h + 1
\end{align*}
\]

executes 4 times (where the assignment at line 2 executes 3 times).

Using the MF memory encoding \( \hat{\mu}(h) = (T 1 0 : 0) \), MF remembers that at line 2 there is no possible observation for the view \( T \) (because for view \( T \) the value of \( h \) is 1 so it doesn’t take the then branch). Hence, the assignment \( h := h + 1 \) only executes once with a memory where \( h = 0 \) (the view of variable \( h \) corresponding to the 3 levels which are not \( T \)).

For a program \( h := l \), where \( \hat{\mu}(l) = (1 : 1) \) in SME, MF keeps only the value 1: a single value represents the fact that all views can observe the same value. Thus the assignment \( h := l \) executes once (and all future executions dependent on \( h \) will also be reduced).

\[\text{SME-TINI} \quad (P, H(\hat{\mu})) \Downarrow \mu_1 \quad (P, L(\hat{\mu})) \Downarrow \mu_2 \quad (P, \hat{\mu}) \Downarrow \hat{\mu} \Downarrow (P, \hat{\mu}) \Downarrow \hat{\mu}(P, \hat{\mu}) \Downarrow \hat{\mu}(P, \hat{\mu}) \Downarrow \hat{\mu}
\]

\[\text{SME-optim} \quad \Downarrow \hat{\mu}(l) \Downarrow \hat{\mu}(l) \quad \Downarrow (x := e, L(\hat{\mu})) \Downarrow (x := e, \hat{\mu}[x \mapsto (\mu(x), \mu(x))])
\]
Hence when encoding of an SME memory can be reduced effec-
tively, multi executions are re-
duced accordingly. As shown in
t he following sections, preserva-
tion of MF memories encoding
through execution requires: to
represent arrays of values as trees
called faceted values and to eval-
uate expressions depending on faceted values. In particular, the
definition of the evaluation of expressions on faceted values de-

dpends highly on the shape of expressions and their values according
to different views, and thus is contradictory to the blackbox prop-
erty of a monitor.

MF may run more multi executions than SME Original MF
has one limitation with respect to SME: it was designed only for
a security lattice of principals: for n principals, such a lattice con-
tains 2^n security levels. The following Ad Exchange platform [35]
example demonstrates that MF may be less efficient than SME in
practice, when the security lattice is not based on principals.

Example 2.1. An Ad Exchange platform needs to put an advertise-
ment on a publisher’s website. For that, it implements a Real-time
Bidding (RTB) system [36], where advertisers can bid for the space
on the publisher’s website to get their ad published. The system
receives as input all the bids offers from bidders and sorts them.
According to the RTB algorithm, the second best offer wins.

We present the lattice of 5 elements for this example in Fig. 2.

For simplicity, we consider only 3 bidders called B1, B2, and B3, an
Ad Exchange (\(\top\)) level which is able to see all the bids, and a public
view \(\bot\). Because MF is designed for a principal lattice, to encode 5
security levels, it uses 3 principals \(k_1, k_2, k_3\), and create a lattice of
8 = \(2^3\) levels, and thus has a potential to run some parts of the
program 8 times, while SME always executes the program 5 times.

We consider one test that naively checks the order of bid offer-
s and decides the winner. The encoding of the lattice is:

\[
\begin{align*}
T & = \{k_1, k_2, k_3\}, \quad B_1 = \{k_1\}, \quad \text{and} \quad \bot = \emptyset. \\
1 & : \text{winner} := 0; \\
2 & : \text{test} := (x_1 \leq x_2) \text{ and } (x_2 \leq x_3); \\
3 & : \text{if test then winner := 2 else skip.}
\end{align*}
\]

The bid values from bidders are \(x_1 = (k_1 ? 10 : 0), x_2 = (k_2 ? 5 : 0),\)
and \(x_3 = (k_3 ? 7 : 0)\). Thus, the resulting value of test at line 2 is
\((k_1 ? (k_2 ? (k_3 ? 7 : 0) : 5 : 0) : 10 : 0)\).

Therefore, the original MF executes the if instruction 8 times
with 3 useless executions for levels \(k_1, k_2, k_3\), and \(k_1, k_3\).

Moreover, because different views of a variable may contain the
same values, MF may execute the same statement several times.
For example, in the execution described above, original MF executes
the then branch 3 times, while it only needs to run once since the
threes executions for the then branch can be merged into one.

3 MF FOR ARBITRARY SECURITY LATTICE

We present an extension to the original Multiple Facets mecha-
nism [3] for an arbitrary security lattice \((\mathcal{L}, \sqsubseteq)\), which we call
Generalised Multiple Facets mechanism, or GMF. Similarly to Multiple
Facets, GMF operates over a faceted memory \(\hat{\mu}\) that maps variables
to simple values or faceted values. A faceted value is of the form
\(\langle l ? V_1 : V_2 \rangle\) where \(l \in \mathcal{L}\) is a security level, and \(V_1\) can be either
a faceted value or a simple value. The first facet \(V_1\) of \(\langle l ? V_1 : V_2 \rangle\)
is called private, and visible to the observers at security level \(l\) or higher
levels in the lattice; the second facet \(V_2\) is called public, and
visible to security levels that are lower or incomparable to \(l\). We
use \(V\) as a meta-variable for faceted values or simple values. Every
evaluation in GMF (see Fig. 3) is marked with a set of security levels
\(pc\), for which the current computation is visible.

3.1 Expression evaluation

By \(\hat{\mu}^{pc}(e)\) we denote the evaluation of expression \(e\) in faceted
memory \(\hat{\mu}\) with set of security lev-

els \(pc\). The definition of \(\hat{\mu}^{pc}(e)\) is presented in Fig. 4. For example,
consider the evaluation of \(x\) when the faceted value \(x\) in memory \(\hat{\mu}\) is
\(\langle l ? V_1 : V_2 \rangle\). To define which facet is
useful given a \(pc\), we consider
the following cases:

- All the levels in \(pc\) are greater than or equal to \(l\), denoted
  \(l \leq pc\) (i.e. \(\forall l' \in pc. \ l \sqsubseteq l'\)):
  the evaluation can use the private facet \(V_1\) because the public fact
  \(V_2\) is anyway not useful for every level in this \(pc\).
- All the levels in \(pc\) are lower than or incomparable to \(l\),
  denoted \(l \not\leq pc\) (i.e. \(\forall l' \in pc. l \not\sqsubseteq l'\)):
  the evaluation can only use the public facet \(V_2\) because \(V_2\) is a faceted visible to any
  view that is lower than or incomparable to \(l\).
- Otherwise, we say that \(l\) and \(pc\) are incomparable and denote
  it by \(l \not\equiv pc\) (i.e. \(\exists l' \in pc. l \equiv l'\)):
  we first evaluate \(V_1\) with \(pc_1 = \{l' \in pc | l \equiv l'\}\) - the set of all levels in \(pc\)
  which are greater than or equal to \(l\). Then, we evaluate \(V_2\) with
  \(pc_2 = pc \setminus pc_1\) which is the set of all levels in \(pc\) which are
  lower than or incomparable to \(l\). Finally, we combine the
two results in a new faceted value.

To evaluate a variable \(x\), we use a special unary operator \(\odot^{pc}(\hat{\mu}(x))\),
which returns the value that is visible to all the levels in the \(pc\).
Let’s consider the case of \(\odot^{pc}(\langle l ? V_1 : V_2 \rangle)\). Notice that, if \(pc\) and \(l\)
are incomparable, meaning that there are some levels in \(pc\) that are
higher than or equal to \(l\) and other levels in \(pc\) that are lower than
or incomparable to \(l\), denoted by \(l \not\equiv pc\), then the evaluation returns
the faceted value \(\langle l ? \odot^{pc_1}(V_1) : \odot^{pc_2}(V_2) \rangle\).
The form of the result of
\(\hat{\mu}^{pc}(e)\) is described in Lemma 3.1.

Lemma 3.1. If \(\hat{\mu}^{pc}(e) = \langle l ? V_1 : V_2 \rangle\), then \(l \equiv pc\).

Example 3.2 (Expression evaluation). Consider the lattice \((\mathcal{L}, \sqsubseteq)\) from
Fig. 3, and the evaluation of \(x + y\) in \(\hat{\mu}\), where \(\hat{\mu}(x) = (M_1 ? 10 : 0), \hat{\mu}(y) = (M_2 ? 5 : 0)\).

Suppose that \(pc = \{M_1, H\}\). Since all the levels in \(pc\) are higher than or equal to \(M_1\), the evaluation of \(x + y\) returns \(\hat{\mu}^{pc}(x) = 10\). Since
\(pc\) and \(M_2\) are incomparable, the evaluation of \(y\) returns \(\hat{\mu}^{pc}(y) = (M_2 ? 5 : 0)\). Next, the evaluation of \(10 + \hat{\mu}^{pc}(M_2 ? 5 : 0)\) is split into
two: one uses a facet visible to \(M_2\) (and hence \(H\)), and another one
The projection function $\hat{\mu}(x) = I_1(x)$ for a faceted memory is defined as $\hat{\mu}(x) = \mu(x)$ if $x \in \Gamma_{\mu}(x)$ and $\mu(x)$ otherwise. The projection function $I_{\mu}(x) = \text{glb}(\mathcal{L})$, where $\mu(x) = \text{glb}(\mathcal{L})$. The projection function $I_1(x) = \text{glb}(\mathcal{L})$, where $I_1(x) = \text{glb}(\mathcal{L})$.

### 3.2 Semantics

We abuse the notation and use $I$ as a projection function on simple values, faceted values, and faceted memories. For any $V$, $I(V)$ returns the value in $V$ which is visible to users at level $l$. For any $\hat{\mu}$, $I_\hat{\mu}(x)$ returns the memory in $\hat{\mu}$ which is visible to users at level $l$.

$I(V) = \mu(I_1(V_1), V_2) = \begin{cases} I(V_1) & \text{if } I_1 \subseteq I, \\ I(V_2) & \text{otherwise}. \end{cases}$

The projection function $I$ is defined in the use of the definition of $\hat{\mu}$ function that converts a faceted memory to a simple memory (see Fig. 5).

The semantics of GFM is defined in Fig. 6 as a big-step evaluation relation $\Gamma \vdash (P, \mu) \Downarrow \mu'_{PC}$, where program $P$ is executed in a memory $\mu$ and a security environment $\Gamma$ that maps variables to security levels in a given security lattice $(\mathcal{L}, \sqsubseteq)$.

The main rule GFM first constructs a faceted memory from the standard memory using the transformation $\mu \uparrow_{\Gamma}^{\text{def}}$ from Fig. 5, where $\text{glb}(\mathcal{L})$ is the greatest lower bound of $\mathcal{L}$. The resulting faceted memory keeps original value of each variable $x$ in a private facet, and adds default values (defined by $\text{def}$ function) in a public facet. In a special case when the level of $x$ is the smallest level in a lattice, we keep only a simple value $\mu(x)$ that is visible to all security levels. We then evaluate the program with the constructed faceted memory and $PC = L$. The resulting faceted memory is transformed back to a normal memory by using the projection function $\hat{\mu}_{PC}$.

The semantics rules for if instruction, while loop, and sequence are straightforward. The GAssign rule uses a faceted evaluation $\hat{\mu}_{PC}$ defined in Section 3.1.

Before describing the semantics of if instruction, we first define several auxiliary functions. Let $\text{dom}(\hat{\mu})$ be the domain of $\hat{\mu}$ and $y$ be a fresh variable, i.e., $y \notin \text{dom}(\hat{\mu})$. By $\hat{\mu}(y)$ we denote a new memory $\mu'$, such that $\text{dom}(\mu') = \text{dom}(\hat{\mu}) \cup \{y\}$, $\mu'(y) = V$ and for all $x \in \text{dom}(\hat{\mu})$, $\mu'(x) = \hat{\mu}(x)$. By $\hat{\mu}(y)$ we remove $y$ from the domain of $\hat{\mu}$, which is $\mu'(y)$. By $\mu(y)$, we remove $y$ from the domain of $\mu$, which is $\mu'(y)$. The semantics rules for if, while loop, and sequence are straightforward. The GAssign rule uses a faceted evaluation $\hat{\mu}_{PC}$ defined in Section 3.1.
Figure 7: Optimisation of a faceted value.

Notice that the form of a faceted value constructed by combining values can be reduced. For example, a faceted value of the form \( (H : V_11 : V_{12}) : V_2 \) can be reduced to \( (H : V_{11} : V_{12}) : V_2 \) because \( M_1 \subseteq H \) and the projection of the original value at any level is either \( V_{11} \) or \( V_{12} \) We use the optimisation on the constructed faceted values from Fig. 7.

Therefore, in the GIF-\( S \) rule after the evaluation of \( P' \) in two contexts, we combine the resulting faceted memories \( \tilde{\mu}_1^\prime \mathbf{y} \) and \( \tilde{\mu}_2^\prime \mathbf{y} \) and apply an optimisation operator \( \llbracket \cdot \rrbracket \) for each newly constructed faceted value. The correctness of \( \llbracket \cdot \rrbracket \) used to optimize faceted values is proven in Lemma 3.3.

**Lemma 3.3.** For all \( I, \) all \( V, \) it follows that \( \mathnormal{\llbracket V \rrbracket} = \mathnormal{\llbracket\llbracket V \rrbracket\rrbracket} \).

**Example 3.4 (Evaluation of if instruction).** Consider the security lattice \( (L_\mathcal{O}, \subseteq) \) from Fig. 3 and the evaluation of the following program \( P \) with \( pc = L_\mathcal{O} \) and \( \tilde{\mu} \), where \( \tilde{\mu}(x) = \langle M_1 \? (H \times tt \times ff) \rangle \).

1. if \( x \) then \( z := 10 \) else \( z := 5 \)

The evaluation follows the GIF-\( S \) rule since \( M_1 \mid (L_\mathcal{O}). \) We construct \( P'' \) if \( y_1 \) then \( P_1 \) else \( P_2 \) and first evaluate \( P'' \) with \( P_1 = \{ \langle \mathbb{L} \rangle \in pc \mid M_1 \subseteq \mathbb{L} \} = \{ M_1, H \} \) and \( \tilde{\mu}(x) = \tilde{\mu}(y) = \langle H \times tt \times ff \rangle \), and then evaluate \( P'' \) with \( P_2 = pc \setminus \{ M_1 \} = \{ M_2, L \} \) and \( \tilde{\mu}(x) = \langle H \times tt \times tt \rangle \).

Since \( P_1 = \{ M_1, H \} \) and \( \tilde{\mu}(y) = \langle H \times tt \times ff \rangle \), the evaluation of \( P'' \) with \( P_1 \) and \( \tilde{\mu} \) is split again to two evaluations: one with \( P''' = \{ P_1 \} \) and \( \tilde{\mu} \) and the other one with \( P'''' = \{ P_2 \} \) and \( \tilde{\mu} \) where \( \tilde{\mu}'(x) = \langle H \times 10 \times 5 \rangle \). The evaluation of \( P'''' \) with \( P_2 \) follows the GIF-\( C \) rule and we get two faceted memories \( \tilde{\mu}'_1 \mathbf{y} \) and \( \tilde{\mu}'_2 \mathbf{y} \) where \( \tilde{\mu}'_1(z) = 10 \) and \( \tilde{\mu}'_2(z) = 5 \). Then, \( \tilde{\mu}'_1 \mathbf{y} \) and \( \tilde{\mu}'_2 \mathbf{y} \) are combined and we get \( \tilde{\mu}'(z) = \langle H \times 10 \times 5 \rangle \).

The evaluation of \( P_2 \) with \( P_2 \) follows the GIF-\( C \) rule and the result is \( \tilde{\mu}' \mathbf{y} \), where \( \tilde{\mu}'(z) = 10 \). At this point, \( \tilde{\mu}' \mathbf{y} \) and \( \tilde{\mu}' \mathbf{y} \) are combined and the result is \( \tilde{\mu}' \), where \( \tilde{\mu}'(z) = \langle M_1 \? (H \times 10 \times 5) \rangle \).

Following GMF rule, the program is evaluated with \( pc = L_\mathcal{O} = \{ M_1, M_2, L \} \). For the assignment instruction, the value of \( x \) is updated to \( \tilde{\mu}'(x) = \langle M_1 \? (H \times tt \times ff) \rangle \). The rest of the evaluation is described in Example 3.4, and the resultant faceted memory is \( \tilde{\mu}' \), where \( \tilde{\mu}'(z) = \langle M_1 \? (H \times 10 \times 5) \rangle \).

The memory after the application of rule GMF is \( \tilde{\mu}' = \tilde{\mu}' \mathbf{y} \). Since \( \tilde{\mu}'(z) = H \), the value of \( z \) is \( \tilde{\mu}'(z) = H \langle (M_1 \? (H \times 10 \times 5) \rangle \rangle ) = 10 \).

**3.3 Equivalence to SME-TINI and Security Guarantee**

**SME-TINI.** The semantics of SME-TINI, termination-insensitive version of SME, for an arbitrary security lattice is presented below, where \( \tilde{\mu} \) is a vector that maps levels to normal memories; \( \mu \) is a memory where variables of values at levels that are not visible to \( l \) are replaced by default values; \( \tilde{\mu}'(x) = \tilde{\mu}'[x] \) constructs a memory by combining all memories in \( \tilde{\mu} \); and \( \mu \) is a function mapping variables to default values.

**SME-TINI**

\[
\forall l \in L : (P, \mu \downarrow \Gamma) \Downarrow \tilde{\mu}(l)
\]

\[
\Gamma + (P, \mu) \Downarrow \text{SME-TINI} \Downarrow \tilde{\mu}(\tilde{\mu})
\]

\[
\mu \Downarrow \Gamma \Downarrow \begin{cases} 
\text{def}(x) & \text{if } (\Gamma \setminus x) \subseteq \mathbb{L} \\
\mu(x) & \text{if } (\Gamma \setminus x) \subseteq \mathbb{L}
\end{cases}
\]

We now prove that SME-TINI enforces termination-insensitive noninterference (TINI). Two memories \( \mu \) and \( \mu' \) are equivalent at \( l \) w.r.t. \( \Gamma \) (denoted by \( \mu \equiv_l \mu' \)) iff all for \( x, (\Gamma \setminus x) \subseteq \mathbb{L} \Rightarrow \mu(x) = \mu'(x) \).

When \( \Gamma \) is clear from the context, \( \mu \equiv_l \mu' \) is written as \( \mu \equiv_l \mu' \).

**Definition 3.6 (TINI).** An enforcement mechanism \( A \) is termination insensitive non-interference (TINI) if for all security environments \( \Gamma \), programs \( P \), and memories \( \mu_1 \) and \( \mu_2 \), we have

\[
\mu_1 \equiv_l \mu_2 \land (\Gamma + (P, \mu_1) \Downarrow A \mu'_1 \land (\Gamma + (P, \mu_2) \Downarrow A \mu'_2 \downarrow A \equiv_l \mu'_1 \equiv_l \mu'_2
\]

**Theorem 3.7.** SME-TINI is TINI.

**Equivalence to SME-TINI.** To prove the equivalence between GMF and SME-TINI, we formally define the semantic equivalence of two mechanisms.

**Definition 3.8.** Two enforcement mechanisms \( A \) and \( B \) are equivalent if for any \( \Gamma \), \( P \), and \( \mu \), we have that \( \Gamma + (P, \mu) \Downarrow A \mu' \Downarrow A \Gamma + (P, \mu) \Downarrow B \mu' \).

We next establish the relation between the execution with GMF semantics and the execution with the standard semantics.

**Lemma 3.9.** (\( P, \mu \)) \( \mu \downarrow \Gamma \Downarrow \Gamma \Downarrow \mu \Downarrow \Gamma \) for all \( l \in pc \).

Thanks to Lemma 3.9, we now prove the equivalence of GMF and SME-TINI.

**Theorem 3.10.** GMF and SME-TINI are equivalent.

As a consequence, we have that GMF is TINI.

**Remark 3.1.** MF [3] is constructed for a set of principals. When the set of principals is fixed, we can use GMF to encode MF: we construct the lattice \( (\mathbb{P}, \subseteq) \), where each element is a set of principals; we prove that GMF for \( (\mathbb{P}, \subseteq) \) and MF for \( \mathbf{P} \) are equivalent [23].
4 OPTIMIZING GMF

In Section 3, we presented the semantics of Generalised Multiple Facets (GMF) for arbitrary lattice and have proven it to be equivalent to SME-TINI. However, GMF from Fig. 6 can be further optimized and avoid repeating evaluations of the same commands. The following example demonstrates the sub-optimality of GMF.

Example 4.1 (GMF is not optimal). We consider the below program from Example 2.1. The lattice is $\langle \mathcal{L}_B, \sqsubseteq \rangle$ from Fig. 2.

1. winner := 0;
2. test := (x1 ≤ x2) and (x2 ≤ x3);
3. if test then winner := 2 else skip

Suppose that the bid offers of $B_1$, $B_2$, and $B_3$ are respectively 10, 5, and 7, and the default values for $B_1$ are 0. W.r.t. this setting, the initial faceted memory is $\hat{\mu}$, where $\hat{\mu}(X_1) = (B_1 \uparrow 10 \downarrow 0)$, $\hat{\mu}(X_2) = (B_2 \uparrow 5 \downarrow 0)$, and $\hat{\mu}(X_3) = (B_3 \uparrow 7 \downarrow 0)$. We consider the execution of the program with GMF.

After line 2, test := (B1 ? B2 ? ff : ff) := (B1 ? B2 ? tt : tt)). Following the semantics of GMF, the assignment instruction winner := 2 is evaluated twice with pcB3 = {B3}, and pcL = {}; the skip instruction is evaluated three times with pcT = {T}, pcB1 = {B1}, and pcB2 = {B2}.

The main idea of our optimisation lies in reducing the number of sub-evaluations and hence the number of faceted memory combinations. For Example 4.1, we propose a mechanism that merges the evaluations corresponding to pcB3 and pcL into one evaluation with pcC = {B3, L}. This simplification is possible since test denotes the same value (i.e., tt) under pcB3 and pcL. Similarly, our simplification merges the evaluations corresponding to pcT, pcB1, and pcB2, where test denotes ff, into one evaluation with pcC = {T, B1, B2}, and thus evaluates each branch of the if command only once.

In this section, we propose semantics of optimized GMF (OGMF) that reduces the number of sub-evaluations, and hence is more resource-friendly than GMF.

4.1 Semantics

The ideas behind the OGMF rule, and the rules for skip, assignment, sequence, and while instructions are similar to the corresponding ones of GMF. The functions $\hat{\mu}^\text{def}_x(x)$ and $\hat{\mu}^\text{pc}_x(x)$ are defined in Fig. 5. We now explain the semantic rules for the conditional instruction.

Consider evaluation of the program if e then P1 else P2 with pc and memory $\hat{\mu}$, and $\hat{\mu}^\text{pc}_x(e) = V$. In order to evaluate each branch of the conditional only once, we split the pc into two subsets: in the first subset $pc_1$ the visible value of $V$ is true, and in the remaining subset $pc_2$, $V$ is false. We now have three distinct cases.

If $pc_1 = pc$, meaning that for all levels in $pc$, the visible value of $V$ is true, then $P_1$ is evaluated (rule OIf-T). If $pc_2 = pc$, then for all levels in $pc$, the visible value of $V$ is false, and only $P_2$ is evaluated (rule OIf-F). Finally, when $pc$ is split in non-empty $pc_1$ and $pc_2$, then both $P_1$ and $P_2$ are evaluated, and their results ($\hat{\mu}_1(x)$ and $\hat{\mu}_2(x)$) are combined by $\hat{\mu}_l^\text{pc}_1(x) \hat{\mu}_l^\text{pc}_2(x)$ (rule OIf-S) to a new faceted memory. The intuition behind this combination is that the projection of $\hat{\mu}_l^\text{pc}_1(x) \hat{\mu}_l^\text{pc}_2(x)$ at $l \in pc_1$ is taken from the evaluation of $P_1$ and its projection at $l \in pc_2$ is taken from the evaluation of $P_2$.

Figure 8: Optimized multiple facets for arbitrary lattice

In the definition of combination of memories for OGMF (bottom of Fig. 8), we distinguish two cases. If for some variable $x$, its value in both faceted memories is the same, ($\hat{\mu}_1(x) = \hat{\mu}_2(x)$), then we do not need to construct a new faceted value. Instead, we optimize the current value using the optimisation operator from Fig. 7.

If the values of $x$ in $\hat{\mu}_1(x)$ and $\hat{\mu}_2(x)$ are different, then we construct a new faceted value $V = \mathbb{P}(\mathbb{V}, \mathbb{V}, pc_1, pc_2)$ and apply further optimisation on the resulting value $\hat{\mu}$ using a new optimisation operator that takes into account a faceted value and the current pc: $\mathbb{P}[V, pc]$ optimizes the form of $V$ and is described in Fig. 9. We show an example of such optimisation in Example 4.4.

To combine two faceted memories, we first construct a new faceted value by using $\mathbb{P}(\mathbb{V}_1, \mathbb{V}_2, pc_1, pc_2)$:

$$\mathbb{P}(\mathbb{V}_1, \mathbb{V}_2, pc_1, pc_2) = \langle \text{List}(pc_1 \cup pc_2), \mathbb{V}_1, \mathbb{V}_2, pc_1, pc_2 \rangle$$

where List(S) is a list of security levels from a set $S$, such that if $l$ appears before $l'$ in List(S) then $l \not\sqsubseteq l'$. If the relation $\sqsubseteq$ in a given security lattice is not a total order, we can transform it into a total order $\sqsubseteq_T$ provided that $\sqsubseteq$ is a finite partial order. We can then view List(S) as a list such that for any $l$ and $l'$ in this list, if $l$ appears before $l'$, then $l' \sqsubseteq_T l$.

The definition of $\mathbb{P}(\mathbb{V}_1, \mathbb{V}_2, pc_1, pc_2)$ uses the following operator that creates a faceted value based on an ordered list of security

$$\mathbb{P}(\mathbb{V}_1, \mathbb{V}_2, pc_1, pc_2) = \langle \text{List}(pc_1 \cup pc_2), \mathbb{V}_1, \mathbb{V}_2, pc_1, pc_2 \rangle$$

where List(S) is a list of security levels from a set $S$, such that if $l$ appears before $l'$ in List(S) then $l \not\sqsubseteq l'$. If the relation $\sqsubseteq$ in a given security lattice is not a total order, we can transform it into a total order $\sqsubseteq_T$ provided that $\sqsubseteq$ is a finite partial order. We can then view List(S) as a list such that for any $l$ and $l'$ in this list, if $l$ appears before $l'$, then $l' \sqsubseteq_T l$.
levels $L$, two faceted values, $pc_1$ and $pc_2$:

$$
\langle L, V_1, V_2, pc_1, pc_2 \rangle =
\begin{cases}
   \langle l(V_1) \rangle & \text{if } L = l, l \in pc_1, \\
   \langle l(V_2) \rangle & \text{if } L = l, l \in pc_2, \\
   \langle l(V_1), V_2, pc_1, pc_2 \rangle & \text{if } L = I_T, T \neq \emptyset, l \in pc_1, \\
   \langle l(V_2), V_1, pc_1, pc_2 \rangle & \text{if } L = I_T, T \neq \emptyset, l \in pc_2.
\end{cases}
$$

Notice that the form of the faceted value created by $\mathcal{F}(V_1, V_2, pc_1, pc_2)$ may be suboptimal.

**Example 4.2 (Faceted value construction).** Suppose that $V_1 = 2, V_2 = 0, pc_1 = \{B_3, \bot\}, pc_2 = \{T, B_1, B_2\}$, List$(pc_1 \cup pc_2)$ is $\{T, B_1, B_2, B_3, \bot\}$, and the lattice $(\mathcal{L}_B, \sqsubseteq)$ is from Fig. 2.

Following the definition of combination of faceted memories, we have $\mathcal{F}(2, 0, pc_1, pc_2) = \{T : \{B_1, B_2, B_3, \bot\} \}$, and the lattice $(\mathcal{L}_B, \sqsubseteq)$ is from Fig. 2.

We therefore define an optimisation function $\mathcal{F}(V, pc)$ that further optimises the result $V$ of a $\mathcal{F}(\cdot)$ function. The optimisation uses the observation that faceted value returned by $\mathcal{F}(\cdot)$ has the form of $(l ? v : V')$, where $V'$ is either a simple value or a faceted value.

The function $\mathcal{F}(V, pc)$ is defined in Fig. 19. If $V$ is of the form $(l ? v : v')$, then the optimisation is straightforward. Now we consider the case when $V$ is of the form $(l ? v : (l' ? v' : V'))$. For demonstration, consider the lattice $(\mathcal{L}_B, \sqsubseteq)$ from Fig. 2.

If the faceted value $V$ is of the form $(T ? v : (B_1 ? v : V'))$ (formally, $l' \sqsubseteq l$ and $v = v'$), then it can be reduced to $(B_1 ? v : V', pc')$ (formally, $((l' \sqsubseteq v' : v'), pc')$), where $pc' = pc \setminus \{T\}$.

If the faceted value $V$ is of the form $(B_1 ? v : (B_2 ? v : V'))$, $(l$ and $l'$ are incomparable and $v = v'$), and moreover for all the levels in the $pc$, for which either $B_1$ or $B_2$ is visible, it is guaranteed that they observe the same value $v$ (see the definition of cond$(V, pc)$ below), then we distinguish the following two cases:

<table>
<thead>
<tr>
<th>cond$(V, pc)$</th>
<th>$V = (l ? v : (l' ? v' : V'))$ and $\forall l_1 \in pc : \text{glb}(l, l') \sqsubseteq l_1$ $\implies l_1(V) = v$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>If all levels in $pc$ are greater than or equal to $\text{glb}(l, l')$ (i.e., $\text{glb}(l, l') \leq pc$), then $V$ is reduced to $v$. For example, if $pc = {B_1, B_2, B_3}$, $\text{glb}(B_1, B_2) = \bot$, then $\text{glb}(B_1, B_2) \leq pc$, and thanks to the cond$(V, pc)$ we know that $B_1(V) = B_2(V) = B_3(V) = v$, then we can reduce such faceted value to simply $v$ because value $V'$ is not useful for such $pc$.</td>
<td></td>
</tr>
</tbody>
</table>
| If only some levels in $pc$ are greater than or equal to $\text{glb}(l, l')$ (i.e., $\text{glb}(l, l')\| pc$), then $V$ is reduced to $(\text{glb}(l, l') ? v : V')$ and this value is reduced further recursively. Consider that we add one more security level $L$ to the lattice $(\mathcal{L}_B, \sqsubseteq)$ such that $L \sqsubseteq \bot$. If $pc = \{B_1, B_2, L\}$, $\text{glb}(B_1, B_2) = \bot$, then $\text{glb}(B_1, B_2)\| pc$ because $\bot \sqsubseteq L$. We then construct a set of security levels $S$ from $pc$, which are higher or equal than $\text{glb}(l, l')$, and therefore the view on $V$ from all these levels is $v$ (because cond$(V, pc)$ holds). In our example, $S = \{B_1, B_2\}$, and we construct a new faceted value $V'' = ((\text{glb}(L, V')) ? l(V')) = L(V')$. We then define a new $pc' = (pc \setminus S) \cup \{\text{glb}(l, l')\} = (L, l)$, and we need to keep $\text{glb}(l, l')$ in $pc'$ because we must ensure that all the levels present in the new faceted value are also present in $pc$. Therefore, the reduced faceted value for our example is $((\bot ? v : L(V')) ? l(V)) = (L, l)$.

Finally, if none of the above conditions hold then we recursively reduce the facet $(l'' ? v' : V')$.

The correctness of $\hat{\mu}_1 \circ pc_1 \circ pc_2 \hat{\mu}_2$ in the $\text{Off-S}$ rule is proven in Lemma 4.3.

**Lemma 4.3.** For all levels $l$, variables $x$, sets of security levels $pc_1$ and $pc_2$, and memories $\hat{\mu}_1$ and $\hat{\mu}_2$,

- if $l \in pc_1$, then $l(\hat{\mu}_1 \circ pc_1 \circ pc_2 \hat{\mu}_2)(x) = l(\hat{\mu}_1)(x)$,
- if $l \in pc_2$, then $l(\hat{\mu}_1 \circ pc_1 \circ pc_2 \hat{\mu}_2)(x) = l(\hat{\mu}_2)(x)$.

**Example 4.4 (Optimisation of faceted value).** Consider a faceted value $(T ? 0 : (B_1 ? 0 : (B_2 ? 0 : (B_3 ? 2 : 2))))$ and $pc = \{T, B_1, B_2, B_3, \bot\}$ from Example 4.2. We show how this value is optimised with our optimisation function $\mathcal{F}(\cdot)$:

$$
\begin{align*}
\langle \langle T ? 0 : (B_1 ? 0 : (B_2 ? 0 : (B_3 ? 2 : 2))) \rangle \rangle & = \\
\langle \langle B_1 ? 0 : (B_2 ? 0 : (B_3 ? 2 : 2)) \rangle \rangle & = \\
\langle \langle B_2 ? 0 : ((B_2 ? 0 : (B_3 ? 2 : 2)) \rangle \rangle & = \\
\langle \langle B_3 ? 2 : 2 \rangle \rangle & = (B_3 ? 2 : 0)
\end{align*}
$$

**Example 4.5 (OGMF is more resource-friendly than GMF).** Consider the program from Example 4.1. To show optimisation of OGMF, we evaluate it with $pc = \{T, B_1, B_2, B_3, \bot\}$ and $\hat{\mu}$, where $\hat{\mu}(x_1) = (B_1 ? 10 : 0), \hat{\mu}(x_2) = (B_2 ? 5 : 0)$, and $\hat{\mu}(x_3) = (B_3 ? 7 : 0)$. After the execution of the instruction at line 2, the faceted memory is $\hat{\mu}' = \hat{\mu} (\text{winner} \mapsto 0, \text{test} \mapsto V)$, where $V = (B_1 ? 0 : (B_2 ? 0 : (B_3 ? \text{tt} : \text{tt})))$.

We consider the execution of the if instruction. For levels $pc_1 = \{B_3, \bot\}$, the evaluation of test is $\text{tt} : B_3(\hat{\mu}'(\text{test}) = \land(\hat{\mu}'(\text{test}) = \text{tt})$. Moreover, $pc_1 \neq pc$, therefore, the rule Off-S applies. The evaluation of the program is split to two: the first evaluation is with $P_1 = \text{winner} : 2$ and $pc_1 = \{B_3, \bot\}$; and the second evaluation is with $P_2 = \text{skip}$ and $pc_2 = \{T, B_1, B_2\}$. Each branch of the conditional will be evaluated only once. The evaluation of $P_1$ with $pc_1$ terminates with $\hat{\mu}_1''(\text{winner}) = 2$. The evaluation of $P_2$ with $pc_2$ terminates with $\hat{\mu}_2''(\text{winner}) = 0$. These two faceted memories are combined to $\hat{\mu}''$, where $\hat{\mu}''(\text{winner}) = (B_1 ? 0 : (B_2 ? 0 : 2))$ (as described in Examples 4.2 and 4.4).

In the example above, OGMF has only two sub-evaluations, while GMF has five, moreover OGMF combines faceted memories once, while GMF combines them four times. Therefore, OGMF is more resource-friendly than GMF.

### 4.2 Equivalence to SME-TINI and Security Guarantee

We first establish the relation between the standard semantics and the semantics of OGMF.

**Lemma 4.6.** $(P, \hat{\mu}) \circ P_O \circ \hat{\mu} \iff (P, I(\hat{\mu}) \circ I(\hat{\mu}))$ for all $l \in pc$.

We now prove the semantic equivalence result for OGMF and SME-TINI.

**Theorem 4.7.** OGMF and SME-TINI are equivalent.

As a consequence, OGMF and GMF are equivalent even though OGMF is optimized. In addition, OGMF is TINI.
\[
\llbracket (l \ ? \ v : v') , pc \rrbracket = \begin{cases}
  v & \text{if } v = v' \\
  (l \ ? \ v : v') & \text{otherwise}.
\end{cases}
\]

\[
\llbracket (l' ? v : v') , pc \rrbracket = \begin{cases}
  \llbracket (l' ? v : v') , pc \rrbracket & \text{if } l' \subseteq l, v = v', \text{where } pc' = pc \setminus \{ l \}, \\
  v & \text{if } l \parallel l', v = v', \text{cond}(V, pc) \text{ and } \text{glb}(l, l') \prec pc.
\end{cases}
\]

\[
\llbracket (l ? v : (l' ? v : v')) , pc \rrbracket = \begin{cases}
  \llbracket \text{glb}(l, l') ? v : v' , pc \rrbracket & \text{if } l \parallel l', v = v', \text{cond}(V, pc) \text{ and } \text{glb}(l, l') \parallel pc, \text{ where } pc' = (pc \setminus S) \cup \{ \text{glb}(l, l') \}, \\
  (l ? v : (l' ? v : v')) , pc & \text{otherwise, where } pc' = pc \setminus \{ l \}.
\end{cases}
\]

\[
\langle L, V \rangle = \begin{cases}
  (l(V) & \text{if } L = l, \\
  (l ? (T, V)) & \text{if } L = 1, T \neq [].
\end{cases}
\]

\section{A Termination Sensitive Version of\nMultiple Facets}

A termination sensitive model assumes that an attacker can observe\ntermination of evaluations. In [19], the model is explained\nfurther: an attacker at level \( l \) can observe the termination of\nevaluations at level \( l \) and lower. In the case of GMF and OGMF, an\nevaluation marked with \( pc \) is an evaluation at \( l \) if \( l \in pc \). Notice that\nan evaluation is at more than one level whenever \( pc \) is not a singleton.

As illustrated by Example 5.1, GMF and OGMF do not prevent\nthe influence of private data at higher levels to the termination of\nevaluations at lower levels. In other words, GMF and OGMF do not\nprevent leakage on termination channel [19].

\textbf{Example 5.1.} Suppose that \( L = \{ L, H \} \), where \( L \subseteq H \). We\nlook at the evaluation of \( \text{if } x \text{ then (while } t \text{ do skip) else skip} \) with\n\( pc = L \) and \( \mu(x) = (H ? t : H) \). When GMF or OGMF is used,\nthe evaluation is split into two: one is with \( pc_1 = \{ H \} \), the other\none is with \( pc_2 = \{ L \} \). The evaluation with \( pc_2 \) converges, while\nthe evaluation with \( pc_1 \) diverges since its executing program is\n\textbf{while } t \textbf{ do skip}. Therefore, the evaluation of the whole program\nwith \( pc = \{ L, H \} \) also diverges and hence, to an attacker at \( L \), the\nevaluation at \( L \) diverges. However, if the program is evaluated\nwith \( \mu'(x) = (H ? t : H) \), to the attacker at \( L \), the evaluation at \( L \)\nconverges. Based on observations on those two evaluations, an\nattacker at \( L \) can gain insight about the high facet of \( x \). In other\nwords, GMF and OGMF do not prevent the influence of data at \( H \)\nto the termination of the evaluation at \( L \).

Therefore, we propose Termination Sensitive Multiple Facets (TSMF),\na version of MF that takes into account the termination sensitive\nmodel. TSMF is a generalization of a version of MF presented in [8,\nAppendix A]. The basic idea of TSMF is that when an if instruction\nis evaluated, TSMF performs a bounded evaluation of the\ninstruction by using OGMF. If the OGMF evaluation does not terminate\nwithin the given time bound, then the instruction is evaluated in\nstead using SME semantics with a low-prio scheduler [14]. The\nsecurity guarantees offered by TSMF are the same as SME with\nthe same low-prio scheduler [19]. The semantics of TSMF and the\nproofs about its security guarantees can be found in [23].

\section{Related Work}

SME. Devriese and Piessens introduce the idea of Secure Multi-\nExecution [14]. Since then, many researchers have developed different\naspects of this approach. Close to our work, Kashyap et al. [19]\ndiscuss how schedulers might affect security guarantees (i.e., TSNI\nand TINI) based on the chosen scheduler and the lattice ordering.\nThey show several schedulers and classify them according to the\nstrength of security guarantees and according to fairness properties.\nThis work complement theirs by providing a similar analysis but\nfor an interplay of MF and SME semantics. SME [14] has many\nimplementations: as a library in Haskell [18], as an experimental\nweb browser based on Firefox [13], as a static program transformation\nfor both Python and JavaScript [4], and as an adaptation to reactive\nsystems [6]. In the work above, SME preserves the semantics of\nsecure programs up to interleaving of events. To remedy that, Zanarini\net al. [37] carefully leverage SME to design a precise monitor which\nexactly preserves semantics of secure programs up to termination.\nSeveral other works [10, 26, 33] expand SME and introduce\ndeclassification. In this work, we focus on semantics guarantees up\nto interleaving of events—as in the SME original formulation.

MF. Austin and Flanagan introduce MF semantics [3]—a technique\noften referred as an optimization for SME. However, as shown by\nBielova and Rezk [7], they do not provide the same security guaran-\ntees (i.e., TINI vs. TSNI) and differ in their treatment of default\nvalues. This work provides yet another look into a comparison\nbetween both techniques to show their differences, while introduc-\ning novel value-based optimizations to MF. Another work by the\nsame authors [9] compare and contrast five dynamic techniques,\nincluding MF and SME, to mainly reason about the preservation of\nsemantics of secure programs, a property known as transparency.\nIn this work, we show that GMF and OGMF enjoy the same trans-\nparency guarantees as SME-TINI (Theorems 3.10 and 4.7).

Tools. Most information flow control tools provide TINI, e.g., Jif [21],\nFlowCampl [25], Laminar [27], Paragon [11], and JSFlow [16].\nSimilarly, termination leaks are often ignored in security tools coming\nfrom the operating system research community, e.g., Asbestos [15],\nHiStar [38], and Flume [20]. A few exceptions to this trend are the\nsecurity libraries LIO [31] and MAC [28], which provide\nTSNI for concurrent programs.

Decentralized label models. The decentralized label model (DLM),\nallows one to express the interests of mutually-distrusting prin-\ncipals without a central authority [22]. The set of labels forms a\npre-order where the order relationship does not require to know\nall the points in the relationship to determine the result of compar-\ning two labels—bearing in mind that there might be an infinite\nnumber of labels due to the dynamic creation of principals at run-\ntime. In a similar spirit, DC-labels [32, 34] provides a decentralized\nlabel format which allows one to express rich policies dictated
by mutually-distrusting principals as propositional logic formulas (without negation). In this work, we require to know all the points in the chosen lattice in order to optimize MF as shown by OGMF. Extending our techniques to DLM or DC-labels is an interesting direction for future work.

7 CONCLUSION AND PERSPECTIVES

This work contributes to develop techniques to secure programs using dynamic information flow—a promising approach to secure existing JavaScript code. We especially focus on proposing a technique that achieves a smaller number of executions than MF (and hence smaller memory footprint) without diminishing security guarantees. We further extend our MF-based technique to work with arbitrary finite lattices (GMF) based on the observation that off-the-shelf lattices with principals are not always the most convenient ones to use. Knowing all the points in the lattice allows for further optimizations: spawning multi-executions could be done on a value-based basis (OGMF) rather than on security levels—as in original MF. Finally, we propose a hybrid approach which present an interesting balance between the number of executions and security guarantees: it behaves as OGMF as long as it can and switches to SME when termination leaks could occur (TSMF). In other words, TSMF prioritizes resource usage as long as there are no risks for termination leaks. We expect that these insights will help inform future development of multi-execution-based techniques. In fact, an intriguing question is what it would take for our optimizations (or future ones) to work on potentially infinite lattices like the DLM or DC-labels—an interesting direction for future work.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for feedback that helped to improve this paper. This research has been partially supported by the ANR projects AJACS ANR-14-CE28-0008, CISC ANR-17-CE25-0014-01, the National Science Foundation under grants 17-CE25-0014-01, the National Science Foundation under grants CCF-1337278 and CCF-1421016, and Swedish research agencies Vetenskapsrådet andSSF Cyber Security projects WebSec: Securing Web-driven Systems and Octopi: Secure Programming for the Internet of Things.

REFERENCES
