Faceted secure multi execution

Citation for the original published paper (version of record):
Schmitz, T., Flanagan, C., Algehed, M. et al (2018). Faceted secure multi execution. Proceedings of the ACM Conference on Computer and Communications Security: 1617-1634.
http://dx.doi.org/10.1145/3243734.3243806

Faceted Secure Multi Execution

Thomas Schmitz
University of California, Santa Cruz
Santa Cruz, California
tschmitz@ucsc.edu

Maximilian Algehed
Chalmers University of Technology
Gothenburg, Sweden
algehed@chalmers.se

Cormac Flanagan
University of California, Santa Cruz
Santa Cruz, California
cormac@ucsc.edu

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

ABSTRACT

To enforce non-interference, both Secure Multi-Execution (SME) and Multiple Facets (MF) rely on the introduction of multi-executions. The attractiveness of these techniques is that they are precise: secure programs running under SME or MF do not change their behavior. Although MF was intended as an optimization for SME, it does provide a weaker security guarantee for termination leaks.

This paper presents Faceted Secure Multi Execution (FSME), a novel synthesis of MF and SME that combines the stronger security guarantees of SME with the optimizations of MF. The development of FSME required a unification of the ideas underlying MF and SME into a new multi-execution framework (Multef), which can be parameterized to provide MF, SME, or our new approach FSME, thus enabling an apples-to-apples comparison and benchmarking of all three approaches. Unlike the original work on MF and SME, Multef supports arbitrary (and possibly infinite) lattices necessary for decentralized labeling models—a feature needed in order to make possible the writing of applications where each principal can impose confidentiality and integrity requirements on data. We provide some micro-benchmarks for evaluating Multef and write a file hosting service, called ProtectedBox, whose functionality can be securely extended via third-party plugins.

CCS CONCEPTS

• **Security and privacy → Information flow control**; • **Software and its engineering → Functional languages**;

KEYWORDS

Multi-Executions; Decentralized Labels; Information-Flow Control; Haskell

ACM Reference Format:
Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. 2018. Faceted Secure Multi Execution. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243806

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243806

1 INTRODUCTION

Information-flow control (IFC) is a promising technology for systematically protecting confidentiality and integrity of data. In the last few years, there have been a proliferation of IFC techniques applied to a wide range of areas such as hardware [59], operating systems [36], programming languages [11], web browsers [51] and distributed systems [33]. Many of these techniques guarantee that secrets are not leaked by enforcing some notion of *non-interference* [23]. This security policy can be enforced either statically (e.g. via type-systems), dynamically (e.g. via runtime monitors), or by a combination of both [45]. Regardless of its dynamic or static nature, traditional IFC approaches might become conservative, thus rejecting secure programs due to imprecisions in the analysis of how information flows.

To mitigate (or even remove entirely) false alarms [43, 57], researchers have recently proposed IFC techniques based on multi-executions: *many copies of a given program (or parts of it) get executed while carefully adapting their semantics to avoid information leakage.* The price to pay is, however, a degradation in performance due to repeated computations. *Secure Multi-Execution* [16] (SME) and *Multiple Facets* [4] (MF) are two approaches based on this idea. On one hand, SME considers programs as black boxes. It executes a copy of the program for each security level while changing the input and output behavior to avoid leaks. MF, on the other hand, inspects the code of the program in order to perform multi-execution of instructions and multiplexing memory only when needed.

Although MF was intended as an optimization for SME, the mechanisms present different security guarantees for termination leaks [8]—i.e., leaks occurring by abnormal termination of programs. More specifically, MF guarantees *termination-insensitive* non-interference (TINI), while SME can remove termination leaks under the right scheduler [28]—thus ensuring *termination-sensitive non-interference* (TSNI).

Ngo et al. [39] have recently shown how to combine MF and SME for a simple while-language in order to ensure TSNI while enjoying some of the MF benefits in terms of minimizing multi-executions. The idea is very simple: run programs under a MF semantics until hitting a sensitive computation which seems to "take too much time to terminate"; in that case the evaluation should restart under a SME semantics, i.e., by spawning one thread for each security level. While a step in the right direction, that work

takes an *all-or-nothing approach*: either the program enjoys the resource-usage-savings of MF or falls into the computations and memory duplication of SME. Furthermore, their technique requires *a priori knowledge of all the points in the lattice*, something which is not feasible for decentralized lattices—lattices which are commonly used by practical IFC systems to allow principals to independently express their confidentiality and integrity requirements on data (e.g., [21, 22, 29, 33, 37, 38, 48, 51]).

From a foundational perspective, this work presents a novel (provably sound) combination of MF and SME called Faceted Secure Multi Execution (FSME), which provides a synthesis of both approaches. Our technique starts running under a MF semantics and spawns only *two* multi-executions when the current computation seems to diverge. However, such multi-executions start running under a MF semantics; so, it might never be necessary to spawn more multi-executions if computations "do not take much time to finish." It may seem a small detail, but it is precisely due to this choice that our approach enjoys the *best of both worlds*. The idea of spawning multi-execution *on-demand* when combining MF and SME is also novel. For that, we strongly rely on extending how MF and SME work when not all the points in the lattice are known—another foundational contribution.

Lastly, our work provides **Multef**, a unifying framework for multi-execution based IFC systems. Regardless the desired multi-execution semantics (i.e., MF, SME, or FSME), **Multef** behaves exactly the same except for a *single specific place*.

This work also contributes to the implementation and evaluation of multi-execution techniques. Despite many claims about MF being more performant than SME, these approaches have not been evaluated against each other besides qualitative informal [5] and theoretical results [8]. It is not clear how they compare quantitatively in terms of performance and memory usage. We believe that one of the main reasons for this is related to the considerable effort it takes to implement such multi-execution based systems [14]. In this light, we build **Multef** upon abstractions found in the functional programming (FP) language Haskell. Firstly, the use of a functional language helps to close the gap between our formal calculus and the implementation—it makes easier to see the correspondence between the semantics rules and their implementation. Secondly, and similar to other work [31, 44, 50], the special treatment of side-effects in Haskell makes it possible to provide **Multef** as a mere library. In that manner, security developers are relieved from building special IFC-aware languages from scratch or performing heavy modifications to the runtime—a major task on its own. Despite IFC libraries usually being small and elegant, it is possible to build non-trivial secure systems [22]. We demonstrate the flexibility of our framework by building a prototype file hosting service, called ProtectedBox, capable of enforcing robust privacy policies on users' files even while allowing untrusted apps to deliver extended features to the system.

It is our intention to establish **Multef** as a foundation for building multi-execution based systems. In summary, the contributions of this work are as follows.

► FSME, a novel combination of MF and SME which lets us enjoy the best of both worlds.

► An extension of SME to work on an "on-demand basis" together with extension of MF to work with the infinite lattice induced by decentralized label models like DC-labels [48].

► **Multef**, a unifying framework capable of providing MF, SME, and FSME.

► Mechanized soundness proofs of **Multef**'s security guarantees in the proof assistant Coq. The proof is parametric on the security lattice as well as the scheduler responsible to run multi-executions. The proof makes appropriated assumptions about these parameters—e.g., decidable label equality and fairness of the scheduler.

► An implementation of **Multef** in Haskell.

► Micro-benchmarks evaluating **Multef**'s performance when executing under a MF, SME, or FSME semantics.

► The implementation of a secure file hosting service called ProtectedBox.

The code, including Coq development and case study, for this paper is available online[1].

2 BACKGROUND

In this work, we assume that programs can access input and output file handles, which in practice may refer to files in a local or remote filesystem or to network sockets. Each input and output file has an associated security label l, and these labels are partially ordered by $\sqsubseteq$ and form a security lattice [15]. Concretely, data read from an input file i with label l_i should only influence data written to output file o (with label l_o) if $l_i \sqsubseteq l_o$; conversely, if $l_i \not\sqsubseteq l_o$ then such influences or information flows are not permitted and should be prevented by the enforcement mechanism. To simplify our discussion, we initially assume a security lattice with two labels low (L) and high (H), where $H \not\sqsubseteq L$ is the only disallowed flow.

We begin with a review of prior technology for ensuring dynamic information flow control via multi-execution. One prominent technology is SME [16], which we illustrate via the Haskell code below. The `when` instruction is simply an `if-then-else` where the `else` branch is just empty.

```
do input <- get highFile
   when heavyExpr (put lowFile (input+1))
```

SME will execute this program twice. One execution is for the high security label H, which can read from `highFile` (`get highFile`), but is prohibited from writing to `lowFile`, i.e., `put lowFile (input+1)` is ignored. The second execution is for the low label L and cannot read from `highFile`; instead some dummy value (e.g., 0) gets bound to variable `input`, and subsequently `input+1` (e.g., 1) is written to `lowFile`. By running the two executions concurrently, SME provides termination-sensitive non-inteference (TSNI). Moreover, SME is *precise*, i.e., it does not change the behavior of non-interfering programs (modulo some technicalities about the relative ordering of writes [43, 57]).

One of the main limitations of SME is performance. For the 2-point lattice, the boolean expression `heavyExpr` gets evaluated twice, even if it does not depend on the input. More generally, a system with n principals might have a powerset security lattice with 2^n labels, and so require 2^n executions.

[1] https://github.com/MaximilianAlgehed/Multef

To address these performance concerns, MF semantics, or also called multi-faceted execution, tries to avoid repeated redundant executions by running the evaluation of `heavyExpr` in the code above just once. More concretely, variable `input` is bound to the *faceted value* $\langle H\,?\,42:0\rangle$, which denotes that the high (secret) value of `input` is `42` while its corresponding low (public/dummy) value is `0`. As a result, the evaluation of `heavyExpr` is triggered only once, not twice—after all, it does not depend on secrets. The evaluation of `input+1` yields the faceted value $\langle H\,?\,43:1\rangle$, and `put` then writes the public facet, i.e., `1`, to the low file, thus avoiding the information leak.

MF provides both precision and non-interference guarantees. Unfortunately, since MF "intertwines" the low and high executions, a low output could block indefinitely on a divergent high computation, and so MF provides only termination-insensitive non-interference (TINI)—rather than the stronger and more desirable TSNI guarantee of SME.

To illustrate this limitation, consider the program below.

```
do secret <- get highFile
   when (secret == 42) diverge
   put lowFile 0
```

Here, `secret` is bound to $\langle H\,?\,42:0\rangle$, indicating that the value 42 read from `highFile` is considered private, with a corresponding public dummy value of 0. Consequently, the subsequent `when` instruction executes *both* the `then` branch (with side-effects and I/O effects visible to high observers) and the (empty) `else` branch (if it were not empty, like in a regular `if-then-else`, the side-effect and I/O actions would be visible to the low observers); after both branches terminate, the remainder of the program executes (with effects visible to both high and low observers). One consequence of this faceted semantics is that the *termination effect* of the high branch is now visible to low observers, which is why MF guarantees only TINI rather than TSNI.

In summary, both MF and SME are precise (i.e., they do not change the behavior of secure programs). On one hand, SME provides TSNI, but with some (perhaps significant) overhead. In contrast, MF addresses this overhead, but at the cost of a weaker security guarantee (TINI).

This work presents a new runtime monitor called FSME (Faceted Secure Multi Execution) that combines the advantages of MF and SME. Note that our approach improves over [39] in that it does not require to restart computations—instead, it gracefully transitions from MF into SME as needed by mid-computations, which in turn requires compatible representations of state and control in the two semantics. Developing the appropriate semantic machinery to unify MF and SME into FSME and to gracefully transition between them is a key contribution of this work.

3 A UNIFYING MULTI EXECUTION FRAMEWORK

We formalize our ideas in terms of a unifying operational semantic framework, called **Multef**, that can express all of SME, MF, and FSME. Our formal development targets an imperative language with mutable reference cells and reactive I/O. However, for ease of exposition, we present here only the core calculus with facets and mutable references; semantics for I/O is deferred to Appendix A. Following Haskell, we distinguish between pure and side-effecting computations.

3.1 Functional core

The functional core of **Multef** is standard, including variables, functions, function application, integers, addition, and conditionals. The language **Multef** is typed. For simplicity, the core types include just Int and function types $T \to T$. We say $t :: T$ to mean that t has type T.

3.2 Faceted values

The language includes *faceted values* $V :: \mathsf{Fac}\ T$, whose behavior can differ according to the security label of an observer. The constructor raw is used to encode concrete values within faceted ones, e.g., raw 42 should be thought of as simply 42. For instance, the faceted value $\langle H\,?\,42:0\rangle$ from Section 2 gets encoded as $\langle H\,?\,\mathsf{raw}\ 42:\mathsf{raw}\ 0\rangle$ in our semantics. For another example, $(\mathsf{raw}\ 0) :: \mathsf{Fac\ Int}$ should be thought of as a faceted value that behaves like 0 for all observers. In contrast,

$$\langle \mathsf{Alice}\,?\,\mathsf{raw}\ 42:\mathsf{raw}\ 0\rangle :: \mathsf{Fac\ Int}$$

is another value of type Fac Int that behaves like 42 for Alice (a label in the security lattice), but like 0 for observers who cannot see Alice's private data. Faceted values can be nested in a tree-like structure, so

$$\langle \mathsf{Alice}\,?\,\langle \mathsf{Bob}\,?\,\mathsf{raw}\ 42:\mathsf{raw}\ 1\rangle:\mathsf{raw}\ 0\rangle$$

behaves like 42 only for viewers who can see the secrets of both Alice and Bob.

To ensure security, programs are not allowed to directly manipulate the raw leaves of a faceted value. Instead, we provide a primitive called bind responsible to apply a computation to each of the *leaves* of the tree structure denoted by faceted values. For example, to add 1 to the faceted value shown above, we would write

$$\mathsf{bind}\ \langle \mathsf{Alice}\,?\,\langle \mathsf{Bob}\,?\,\mathsf{raw}\ 42:\mathsf{raw}\ 1\rangle:\mathsf{raw}\ 0\rangle\ (\lambda x.\ \mathsf{raw}\ (x+1))$$

which evaluates (in several steps) to

$$\langle \mathsf{Alice}\,?\,\langle \mathsf{Bob}\,?\,\mathsf{raw}\ 43:\mathsf{raw}\ 2\rangle:\mathsf{raw}\ 1\rangle.$$

Observe that the computation $(\lambda x.\ \mathsf{raw}\ (x+1))$ is applied to each leave of the faceted value to yield the result. Operationally, if $V :: \mathsf{Fac}\ T_1$ and $f :: T_1 \to \mathsf{Fac}\ T_2$, then bind $V\ f$

- extracts each raw leaf of type T_1 from the faceted tree V,
- applies f to this T_1 argument, producing a result of type Fac T_2, and
- joins these various results from f into a single faceted value of type Fac T_2, which is returned from bind.

3.3 FIO computations

So far, we can express side-effect-free computations on faceted values. To express programs that manipulate both faceted values and mutable reference cells, we introduce the FIO monad—a monad (e.g., [56]) is just a special-purposed data type designed to express computations with side-effects in pure functional languages like Haskell. In this light, the type FIO T characterizes side-effectful secure computations that yield a T value. Because of being a monad,

computations of type FIO T are built by two fundamental operations:

$$\mathsf{return} :: T \to \mathsf{FIO}\ T$$
$$(\gg\!=) :: \mathsf{FIO}\ T_1 \to (T_1 \to \mathsf{FIO}\ T_2) \to \mathsf{FIO}\ T_2$$

The operation return x produces a computation that returns the value of x without causing side-effects. The function $(\gg\!=)$—called *FIO-bind* to distinguish it from the analogous bind operation on faceted values—is used to sequence FIO computations and their associated side-effects. Specifically, $fio \gg\!= f$ executes *fio*, takes its *result* and passes it to the function f, which then returns a second computation to run. Some languages, like Haskell, provide syntactic sugar for monadic computations known as **do**-notation. For instance, the program $fio \gg\!= \lambda x.\mathsf{return}\ (x+1)$, which adds 1 to the value produced by computation *fio*, can be written as

$$\begin{array}{l}\textbf{do}\ x \leftarrow fio\\ \quad \mathsf{return}\ (x+1)\end{array}$$

which gives a more "imperative" feeling to programs.

3.4 Building side-effectful computations based on faceted values

In most programs, side-effects may occur conditionally based on values in the program. For example, the following code snippet performs two different side-effects depending on whether $x :: \mathsf{Int}$ is positive. Let us imagine that, for instance, code $\mathsf{effect}_0 :: \mathsf{FIO}\ ()$ writes 0 to a reference, while $\mathsf{effect}_1 :: \mathsf{FIO}\ ()$ writes 1 instead:

$$\mathsf{if}\ (x > 0)\ \mathsf{effect}_0\ \mathsf{effect}_1 :: \mathsf{FIO}\ ()$$

If computations have side-effects which must depend on *faceted* values, then their type will be of the form Fac (FIO T) for some type T, i.e., a faceted value whose tree-like structure stores side-effectful computations at its leaves—thus expressing that different FIO T computations should be visible to different security levels. In this case, we rely on the special operator

$$\mathsf{run} :: \mathsf{Fac}\ (\mathsf{FIO}\ T) \to \mathsf{FIO}\ (\mathsf{Fac}\ T)$$

to enable interaction[2] between Fac and FIO. Intuitively, run takes all the side-effectful actions inside the tree-like structure of the argument and somehow (e.g., by sequentialising) executes them and collects the results in another tree-like faceted value. For instance, if we change the previous snippet so that the writes should depend on $fx :: \mathsf{Fac\ Int}$, then it becomes

$$\begin{array}{l}p = \mathsf{bind}\ fx\ (\lambda x.\ \mathsf{raw}\ (\mathsf{if}\ (x > 0)\\ \qquad\qquad\qquad\qquad \mathsf{effect}_0\\ \qquad\qquad\qquad\qquad \mathsf{effect}_1\ )) :: \ \mathsf{Fac}\ (\mathsf{FIO}\ ())\end{array}$$

The function $(\lambda x. \ldots) :: \mathsf{Int} \to \mathsf{Fac}\ (\mathsf{FIO}\ ())$ is run for each integer in fx, and so $(\mathsf{bind}\ fx\ (\lambda x. \ldots) :: \mathsf{Fac}\ (\mathsf{FIO}\ ()))$ results in a faceted tree of FIO computations—we use ellipses here to denote the corresponding code above. The primitive run in

$$\mathsf{run}\ (p) :: \mathsf{FIO}\ (\mathsf{Fac}\ ())$$

then controls the sequential or concurrent execution of these various FIO computations, and thus encapsulates the key design choices regarding the different multi-execution approaches that we consider. In our framework, *the semantics of this operation is the one that determines if we consider MF, SME, or FSME when launching multi-executions*. We proceed now to add the operations related to building and executing side-effectful computations.

[2] This run operator enables interaction between the two monads FIO and Fac in the manner proposed by Jones and Duponcheel [25] as the *swap* construction.

3.5 Supported multi-executions approaches

Before we dive into the technicalities of our semantics, we provide some examples to illustrate the different multi-executions semantics that **Multef** considers. Let us consider the following code fragment:

$$\begin{array}{ll}p = & \textbf{do}\ fx \leftarrow \mathsf{get}\ highFile\\ & \quad \mathsf{run}\ (\mathsf{bind}\ fx\ (\lambda x.\ \mathsf{raw}\ (\mathsf{put}\ highFile\ (x+1))))\\ & \quad \mathsf{run}\ (\mathsf{bind}\ fx\ (\lambda x.\ \mathsf{raw}\ (divergeIf42\ x)))\\ & \quad \mathsf{put}\ lowFile\ 0\end{array}$$

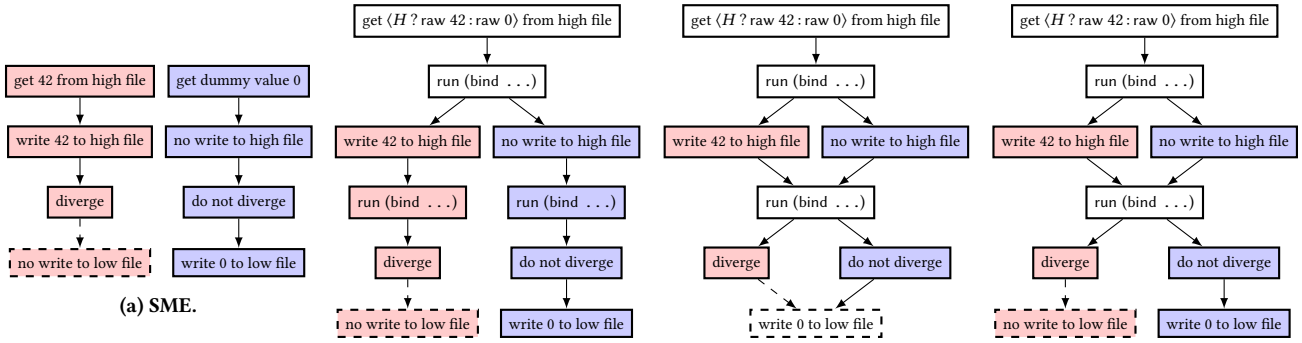
This program $p :: \mathsf{FIO}\ ()$ works as follows. It reads a secret value from a sensitive file—let us assume that the file has stored the number 42. Hence, primitive get returns the faceted integer $fx = \langle H\ ?\ \mathsf{raw}\ 42 : \mathsf{raw}\ 0\rangle$, thus protecting the secret 42. In the next line, run and bind are used to extract the raw $x :: \mathsf{Int}$ from the secret, increment it, and write it into a high file. Similarly, the next line calls the function divergeIf42 which loops when the value given as an argument is 42. Finally, the last instruction writes 0 to a public file. We use this example to illustrate some of the challenges in ensuring TSNI.

SME. The original formulation of SME [16] would run two versions of the program, as shown in Figure 1a. The left high execution can read and write high files, but cannot write to low files. Conversely, the right low execution never sees any secret data; it reads dummy values from high files, but it can write to low files. As the figure shows, SME duplicates both memory and code. The divergence of the high execution does not block the public write in the low execution, thus satisfying TSNI.

Demand-driven SME. Our demand-driven optimization of SME is shown in Figure 1b, where the high and low executions are not *forked until the first call to run*, which then forks two copies of the entire continuation, again satisfying TSNI. As with the main thread, every forked multi-execution will not spawn others until reaching another run.

MF. Figure 1c illustrates how MF processes the example, where run forks two (high and low) subcomputations, and then waits for them to terminate before executing the continuation. This approach is potentially more efficient, but at the cost of violating TSNI, since the divergent high computation now blocks the subsequent public write.

FMSE. Finally, Figure 1d illustrates our novel combination of MF and SME to obtain the best of both worlds, i.e., TSNI security and MF efficiency. Here, run forks two subcomputations, and if both subcomputations terminate within a given time bound (as in the first call to run), then the continuation is run just once, as in MF. However, if the time bound is exceeded (as in the second call of run), then the continuation is executed twice, thus satisfying TSNI. The newly spawned computations will not fork others until reaching run and the time bound has been exceeded again—this is a novelty with respect to previous combination of MF and SME [39] and it



(a) SME. (b) Demand-driven SME. (c) MF. (d) FSME.

Figure 1: Control flow diagrams. Dashed boxes denote code that is not executed due to earlier divergence. Red means $pc = \{H\}$ (high view), blue means $pc = \{\overline{H}\}$ (low view), and white means $pc = \{\}$ (i.e., instructions common to both views).

proves crucial to get good performance in our implementation (see Sections 9 and 10). Furthermore, when it comes to non-termination, FSME guarantees that the thread which hits divergence under a branch does not stop others from making progress. Fully stopping progress in programs can only occur when looping under an empty *pc*—which is secure since it denotes divergence based on public information.

Note that the TSNI guarantee holds for any finite timeout. Larger timeouts may lead to fewer forked continuations and so better performance. Various policies can be used to set the timeout. One plausible option is to set the timeout for the private subcomputation at (say) twice the time required for the public subcomputation.

Multef supports all these variations in multi-execution semantics just by changing the semantics of run, as we explain below.

3.6 Formal semantics

To illustrate the possible semantics for run, we formalize a **Multef** evaluation relation $t \longrightarrow_{pc} t'$ that captures MF and SME, as well as other forking strategies like FSME. Here, *pc* is the *program counter label*, which is a set of constraints called *branches*, each of the form k or $\overline{k}$. If $k \in pc$, then the computation can see only the high-confidentiality facet V_H of any faceted value $\langle k \,?\, V_H : V_L \rangle$. Conversely, if $\overline{k} \in pc$, the computation should only see V_L. If neither k nor $\overline{k}$ are in *pc*, then the computation processes both facets V_H and V_L.

Conceptually, *pc* describes which security labels $l \in Lattice$ are represented by the current computation. We formalize this intuition by the following function *views*, which maps a *pc* to the corresponding set of labels:

$$views(pc) = \{l \in Lattice \mid (\forall k \in pc.\ k \sqsubseteq l) \wedge (\forall \overline{k} \in pc.\ k \not\sqsubseteq l)\}$$

For example, $views(\{k_1, k_2, \overline{k}_3, \overline{k}_4\})$ only includes lattice elements in the upward closure of k_1 and k_2 and not in the upward closure of k_3 or k_4. The most interesting rules for $t \longrightarrow_{pc} t'$ are summarized in Figure 2—see Appendix A for the rest of the semantic rules.

Forking on-demand. The rules for run V form the core of our evaluation strategy, and depend on the structure of the faceted computation tree $V :: \mathsf{Fac}\,(\mathsf{FIO}\,T)$. If V is a faceted value $\langle k \,?\, t_1 : t_2 \rangle$, then in general rule [F-RUN-FACET-3] creates two new threads, denoted by the syntax $\langle\!\langle k \,?\, \mathsf{run}\ t_1 : \mathsf{run}\ t_2 \rangle\!\rangle$, which will proceed to evaluate t_1 and t_2, respectively. Subsequently, the rule [F-THREAD-1] permits evaluation of t_1, with k added to the *pc*, indicating that side-effects of the computation t_1 should only be visible at security levels in $views(pc \cup \{k\})$. Conversely, the rule [F-THREAD-2] permits evaluation of t_2, with $\overline{k}$ added to *pc*. Both rules may be applicable at the same time (our semantics is nondeterministic), which allows for t_1 and t_2 to be evaluated in any order. A concrete scheduler can choose to use either [F-THREAD-1] or [F-THREAD-2] first, and may interleave them to achieve concurrency.

Observe that adding a new branch constraint to the *pc* may entail $views(pc)$ is empty, which means that the current computation is not visible to any observer. Rules [F-RUN-FACET-1] and [F-RUN-FACET-2] are optimizations to avoid unnecessary creation of such "invisible" threads.

MF semantics. Once each FIO computation run t_i for $i \in \{1, 2\}$ terminates to return V_i, rule [F-MERGE] joins the two threads back together into a single terminated FIO computation return $\langle k \,?\, V_1 : V_2 \rangle$. The rules described so far perform MF-like computation by blocking the continuation of run until both sub-threads terminate.

SME semantics. Alternatively, to permit SME-like computation, rule [F-FORK-CONTINUATION] allows the *continuation* (the enclosing evaluation context E) to be copied into each sub-thread, yielding $\langle\!\langle k \,?\, E[t_1] : E[t_2] \rangle\!\rangle$. Consequently, the evaluation of the continuation E in the low thread $E[t_2]$ is not blocked by a divergent high computation t_1 in the high thread. This enables a stronger termination-sensitive security guarantee, but at the cost of evaluating E twice.

FSME semantics. Since **Multef** supports both MF and SME, it is now possible to express our novel approach, Faceted Secure Multi Execution (FSME), which combines the benefits of both. Under most circumstances, FSME proceeds exactly like MF. However, if say the low subcomputation t_2 returns but t_1 exceeds a policy-specified timeout, then the rule [F-FORK-CONTINUATION] is applied to fork the enclosing continuation E, thus allowing the low view to proceed without blocking on the high view.

Note that our semantics is non-deterministic, enabling different evaluation strategies to provide MF, SME, and FSME-like behavior. Although we consider a call-by-name semantics, we expect

Multef syntax

$$t ::= x \mid \lambda x.t \mid t\ t \mid n \mid t + t \mid \text{if } t\ t\ t \mid V \mid \text{return } t \mid t \gg\!= t \mid \text{run } t \mid a \mid \text{new } t \mid \text{read } t \mid \text{write } t\ t \mid \text{get } i \mid \text{put } o\ t \mid \langle\!\langle k \,?\, t : t \rangle\!\rangle$$
$$V ::= \text{raw } t \mid \langle k \,?\, V_H : V_L \rangle \mid \text{bind } t\ t$$

$\boxed{t \longrightarrow_{pc} t}$

$$\text{run } \langle k \,?\, t_1 : t_2 \rangle \longrightarrow_{pc} \begin{cases} \text{run } t_1 & \text{if } views(pc \cup \{\overline{k}\}) = \emptyset & \text{[F-RUN-FACET-1]} \\ \text{run } t_2 & \text{if } views(pc \cup \{k\}) = \emptyset & \text{[F-RUN-FACET-2]} \\ \langle\!\langle k \,?\, \text{run } t_1 : \text{run } t_2 \rangle\!\rangle & \text{otherwise.} & \text{[F-RUN-FACET-3]} \end{cases}$$

$$\langle\!\langle k \,?\, t_1 : t_2 \rangle\!\rangle \longrightarrow_{pc} \langle\!\langle k \,?\, t_1' : t_2 \rangle\!\rangle \quad \text{if } \overline{k} \notin pc \text{ and } t_1 \longrightarrow_{pc \cup \{k\}} t_1' \quad \text{[F-THREAD-1]}$$

$$\langle\!\langle k \,?\, t_1 : t_2 \rangle\!\rangle \longrightarrow_{pc} \langle\!\langle k \,?\, t_1 : t_2' \rangle\!\rangle \quad \text{if } k \notin pc \text{ and } t_2 \longrightarrow_{pc \cup \{\overline{k}\}} t_2' \quad \text{[F-THREAD-2]}$$

$$\langle\!\langle k \,?\, \text{return } V_1 : \text{return } V_2 \rangle\!\rangle \longrightarrow_{pc} \text{return } \langle k \,?\, V_1 : V_2 \rangle \quad \text{[F-MERGE]}$$

$$E[\langle\!\langle k \,?\, t_1 : t_2 \rangle\!\rangle] \longrightarrow_{pc} \langle\!\langle k \,?\, E[t_1] : E[t_2] \rangle\!\rangle \quad \text{[F-FORK-CONTINUATION]}$$

Figure 2: Syntax and selected rules from the Multef semantics.

our results to extend to strict languages by the introduction of explicit suspensions—a well-known technique to encode call-by-name operations in call-by-value semantics.

Side Effects. We extend the operational semantics to support both mutable reference cells and I/O by extending the evaluation relation from terms $t \longrightarrow_{pc} t'$ to states $\sigma \longrightarrow_{pc} \sigma'$, where each state has the form (t, M, P, I, O). The memory M maps reference addresses a to faceted values. Note that reference cells always contain faceted data, as they may be updated by computations that should only be visible at certain security levels. The output buffer O contains an integer sequence $O(o)$ for each output channel o, which is extended by put $o\ n$. The input buffer I also contains an integer sequence $I(i)$ for each input channel i, but these input buffers are not modified during execution; instead, we maintain a buffer pointer $P(i)$ (pointing into $I(i)$) that is incremented as necessary during each get i operation. Since computations at different security levels may advance at different rates, the buffer pointer $P(i)$ can be a faceted tree with integer leaves.

$$\begin{array}{rrl} M \in \mathit{Memory} & = & \mathit{Address} \to \mathit{FacetedValue} \\ p \in \mathit{BufferPointer} & ::= & n \mid \langle k \,?\, p : p \rangle \\ P \in \mathit{BufferPointers} & = & \mathit{InputHandle} \to \mathit{BufferPointer} \\ I \in \mathit{InputBuffer} & = & \mathit{InputHandle} \to \mathbb{Z}^* \\ O \in \mathit{OutputBuffer} & = & \mathit{OutputHandle} \to \mathbb{Z}^* \\ \sigma \in \mathit{State} & ::= & (t, M, P, I, O) \end{array}$$

The previously described rules extend in a natural manner from terms to states. Figure 3 shows the rules to allocate, read, and write reference cells, making sure that values written to the memory M appropriately reflect the current program counter label pc, using the following notation to construct a faceted value from a pc:

$$\begin{array}{lcl} \langle\!\langle \bullet \,?\, \bullet : \bullet \rangle\!\rangle & : & \mathit{PC} \to \mathit{FacetedValue} \to \mathit{FacetedValue} \\ & & \quad \to \mathit{FacetedValue} \\ \langle\!\langle \{\} \,?\, V_1 : V_2 \rangle\!\rangle & = & V_1 \\ \langle\!\langle pc \cup \{k\} \,?\, V_1 : V_2 \rangle\!\rangle & = & \langle k \,?\, \langle\!\langle pc \,?\, V_1 : V_2 \rangle\!\rangle : V_2 \rangle \\ \langle\!\langle pc \cup \{\overline{k}\} \,?\, V_1 : V_2 \rangle\!\rangle & = & \langle k \,?\, V_2 : \langle\!\langle pc \,?\, V_1 : V_2 \rangle\!\rangle \rangle \end{array}$$

Appendix A contains a full definition of our operational semantics, including various rules (such as for I/O) that we do not have space to include here.

4 TERMINATION INSENSITIVE SECURITY GUARANTEES

As a starting point for reasoning about the correctness properties of our faceted framework, we first develop a corresponding "standard" semantics $\xrightarrow{\text{std}}$ for **Multef** that does not perform any faceted evaluation. This semantics works over non-faceted states σ that do not include faceted values $\langle k \,?\, V : V \rangle$, faceted input buffer pointers $\langle k \,?\, p : p \rangle$, or concurrent faceted threads $\langle\!\langle k \,?\, t : t \rangle\!\rangle$. Many of the rules are identical to the corresponding $\longrightarrow_{pc}$ rules; Figure 4 illustrates some modified rules that avoid introducing facets for reference cells.

For any faceted state σ and label l, we can generate a corresponding non-faceted state, denoted $\sigma{\downarrow_l}$, that is the view of σ seen by an observer at level l. This *projection* operation $\sigma{\downarrow_l}$ is defined in Figure 5. We say σ and σ' are l-equivalent (written $\sigma \approx_l \sigma'$) if their l-projections are identical (i.e., $\sigma{\downarrow_l} = \sigma'{\downarrow_l}$).

We now show that each faceted framework step $\sigma \longrightarrow_{pc} \sigma'$ corresponds to either zero or one standard evaluation steps of $\sigma{\downarrow_l}$, provided that $l \in views(pc)$. For example, $\sigma \longrightarrow_{pc} \sigma'$ could evaluate a high thread t_1 inside $\sigma = (\langle\!\langle H \,?\, t_1 : t_2 \rangle\!\rangle, \ldots)$, resulting in $\sigma{\downarrow_H} \xrightarrow{\text{std}} \sigma'{\downarrow_H}$ and $\sigma{\downarrow_L} = \sigma'{\downarrow_L}$. Moreover, if $\sigma \longrightarrow_{pc}$ is stuck, then the projected state $\sigma{\downarrow_l} \xrightarrow{\text{std}}$ is also stuck, again provided that $l \in views(pc)$. Finally, a faceted step $\sigma \longrightarrow_{pc} \sigma'$ does not change any of the state components M, P, I, O seen by a viewer at any level $l \notin views(pc)$.

Theorem 1 (Projection).

(1) If $\sigma \longrightarrow_{pc} \sigma'$ and $l \in views(pc)$, then either $\sigma \approx_l \sigma'$ or $\sigma{\downarrow_l} \xrightarrow{std} \sigma'{\downarrow_l}$.

(2) If $\sigma \not\longrightarrow_{pc}$ and $l \in views(pc)$, then $\sigma{\downarrow_l} \xrightarrow{std}$.

(3) If $(t, M, P, I, O) \longrightarrow_{pc} (t', M', P', I', O')$ and $l \notin views(pc)$, then $M \approx_l M'$ and $P \approx_l P'$ and $I \approx_l I'$ and $O \approx_l O'$.

Based on this projection theorem, we show that our framework satisfies termination-insensitive non-interference. Essentially, if σ_1 and σ_2 are l-equivalent states, then running both states to termination will produce l-equivalent final states, that is, evaluation does not leak information that should be kept hidden from l. Here we use $\sigma_i' \not\longrightarrow_{\emptyset}$ to denote that state σ_i' cannot be evaluated further, and run both computations with the empty $pc = \emptyset$, so the faceted framework simulates standard evaluation for all views.

$\boxed{\sigma \longrightarrow_{pc} \sigma}$

$$(\text{new } V, M, P, I, O) \longrightarrow_{pc} (\text{return } a, M[a := \langle\!\langle pc \,?\, V : \text{raw } 0 \rangle\!\rangle], P, I, O) \quad \text{if } a \notin \text{dom}(M) \quad [\text{F-NEW}]$$

$$(\text{read } a, M, P, I, O) \longrightarrow_{pc} (\text{return } M(a), M, P, I, O) \quad [\text{F-READ}]$$

$$(\text{write } a\ V, M, P, I, O) \longrightarrow_{pc} (\text{return } V, M', P, I, O) \quad \text{if } M' = M[a := \langle\!\langle pc \,?\, V : M(a) \rangle\!\rangle] \quad [\text{F-WRITE}]$$

Figure 3: Rules for references.

$\boxed{\sigma \xrightarrow{\text{std}} \sigma}$

$$(\text{new } F, M, P, I, O) \xrightarrow{\text{std}} (\text{return } a, M[a := F], P, I, O) \quad \text{if } a \notin \text{dom}(M) \quad [\text{S-NEW}]$$

$$(\text{write } a\ F, M, P, I, O) \xrightarrow{\text{std}} (\text{return } F, M[a := F], P, I, O) \quad [\text{S-WRITE}]$$

Figure 4: Selected rules of the standard semantics.

$\boxed{t\downarrow_l = t}$

$$\langle k \,?\, F_1 : F_2 \rangle\downarrow_l = \begin{cases} F_1\downarrow_l & k \sqsubseteq l \\ F_2\downarrow_l & \text{otherwise} \end{cases}$$

$$\langle\!\langle k \,?\, t_1 : t_2 \rangle\!\rangle\downarrow_l = \begin{cases} t_1\downarrow_l & k \sqsubseteq l \\ t_2\downarrow_l & \text{otherwise} \end{cases}$$

$$(\text{put } o\ t)\downarrow_l = \begin{cases} \text{put } o\ (t\downarrow_l) & l_o = l \\ \text{return } (t\downarrow_l) & \text{otherwise} \end{cases}$$

$t\downarrow_l$ is homomorphic otherwise

$\boxed{M\downarrow_l = M}$ $\quad M\downarrow_l = \lambda a.M(a)\downarrow_l$

$\boxed{p\downarrow_l = p}$ $\quad n\downarrow_l = n$

$$\langle k \,?\, p_1 : p_2 \rangle\downarrow_l = \begin{cases} p_1\downarrow_l & k \sqsubseteq l \\ p_2\downarrow_l & \text{otherwise} \end{cases}$$

$\boxed{P\downarrow_l = P}$ $\quad P\downarrow_l = \lambda i.P(i)\downarrow_l$

$\boxed{I\downarrow_l = I}$ $\quad I\downarrow_l = \lambda i. \begin{cases} I(i) & l_i \sqsubseteq l \\ \epsilon & \text{otherwise} \end{cases}$

$\boxed{O\downarrow_l = O}$ $\quad O\downarrow_l = \lambda o. \begin{cases} O(o) & l_o = l \\ \epsilon & \text{otherwise} \end{cases}$

$\boxed{\sigma\downarrow_l = \sigma}$ $\quad (t, M, P, I, O)\downarrow_l = (t\downarrow_l, M\downarrow_l, P\downarrow_l, I\downarrow_l, O\downarrow_l)$

Figure 5: Projection functions.

Theorem 2 (Termination-Insensitive Non-Interference). *If* $\sigma_1 \approx_l \sigma_2$ *and* $\sigma_1 \longrightarrow_\emptyset^* \sigma_1' \not\longrightarrow_\emptyset$ *and* $\sigma_2 \longrightarrow_\emptyset^* \sigma_2' \not\longrightarrow_\emptyset$ *then* $\sigma_1' \approx_l \sigma_2'$.

5 FAIR SCHEDULING

The semantics $\sigma \longrightarrow_{pc} \sigma'$ is non-deterministic, and so requires a *fair scheduler* in order to guarantee the desired termination-sensitive security properties. To illustrate this requirement, consider the term t:

$$\langle\!\langle k \,?\, \mathit{diverge} : \text{return (raw 2)} \rangle\!\rangle \gg\!= \lambda\_.t_2$$

where t_2 = put *publicFile* 3 and *diverge* is a computation that diverges based on the value of some secret. A scheduler that prioritized evaluation of the divergent high thread *diverge* via [F-THREAD-1] could forever block the low output on *publicFile*—which produces a termination leak since the attacker would never see the output 3 performed by t_2. Alternatively, the semantics does permit the low thread to make progress, by using [F-FORK-CONTINUATION] to lift the continuation $(\lambda\_.t_2)$ inside each forked thread, and subsequently executing the continuation twice, at both security levels (in a manner reminiscent of SME) and finally executing the low write t_2 without blocking on *diverge*.

$$\begin{aligned} t &= \langle\!\langle k \,?\, \mathit{diverge} : \text{return (raw 2)} \rangle\!\rangle \gg\!= \lambda\_.t_2 \\ &\longrightarrow_\emptyset \langle\!\langle k \,?\, \mathit{diverge} \gg\!= (\lambda\_.t_2) : \text{return (raw 2)} \gg\!= \lambda\_.t_2 \rangle\!\rangle \\ &\longrightarrow_\emptyset \langle\!\langle k \,?\, \mathit{diverge} \gg\!= (\lambda\_.t_2) : (\lambda\_.t_2)\ (\text{raw 2}) \rangle\!\rangle \\ &\longrightarrow_\emptyset \langle\!\langle k \,?\, \mathit{diverge} \gg\!= (\lambda\_.t_2) : t_2 \rangle\!\rangle \end{aligned}$$

We introduce a fairness requirement to ensure that the implementation does not indefinitely choose high executions when low executions are available—thus avoiding possible termination leaks. A *fair state* $\Sigma = (\sigma, s)$ consists of a state σ plus additional scheduling information s.

$$\begin{aligned} \Sigma \in \mathit{FairState} &::= (\sigma, s) \\ s \in \mathit{SchedulingInfo} & \end{aligned}$$

We leave the scheduling information s abstract and assume only a *fair evaluation relation*

$$(\sigma, s) \xrightarrow{\text{fair}} (\sigma', s')$$

satisfying the properties

- Validity: If $(\sigma, s) \xrightarrow{\text{fair}} (\sigma', s')$ then $\sigma \longrightarrow_\emptyset \sigma'$.
- Blocking: If $(\sigma, s) \not\xrightarrow{\text{fair}}$ then $\sigma \not\longrightarrow_\emptyset$.
- Fairness: $\forall \sigma, s, l. \exists n \in \mathbb{N}$. if σ can l-step, then any n-step fair evaluation sequence $(\sigma, s) \xrightarrow{\text{fair}}{}^n (\sigma', s')$ includes an l-step.

The fairness condition says that, given a fair state (σ, s) and a label l, if the projected state $\sigma\downarrow_l$ seen by a viewer at level l can make progress, then there exists some step limit $n \in \mathbb{N}$ such that any n-step fair evaluation $(\sigma, s) \xrightarrow{\text{fair}}{}^n (\sigma', s')$ will include progress seen by a viewer at level l. This is the essential requirement that stops low outputs from being blocked indefinitely on high computations. The fair evaluation relation will typically be deterministic.

6 TERMINATION SENSITIVE SECURITY GUARANTEES

We next prove a stronger termination-sensitive non-interference result, based on the fair scheduling semantics. First, given any fair state (σ, s) where the l-projection $\sigma\downarrow_l$ can perform a standard step, then the fair semantics will eventually perform a corresponding

step. That is, no view l is ever blocked indefinitely by the fair semantics.

Theorem 3 (Fair Projection).
If $\sigma\downarrow_l \xrightarrow{std} \sigma_1$ *then* $\exists\sigma_2, s_2.\ (\sigma, s) \xrightarrow{fair}{}^{*} (\sigma_2, s_2)$ *and* $\sigma_2\downarrow_l = \sigma_1$.

The fair semantics satisfies TSNI: given two l-equivalent states $\sigma_1 \approx_l \sigma_2$, if σ_1 evaluates to σ_1' via the fair semantics, then σ_2 must also evaluate to a corresponding l-equivalent state σ_2' (and in particular σ_2 cannot diverge before doing so).

Theorem 4 (Termination-Sensitive Non-Interference).
If $\sigma_1 \approx_l \sigma_2$ *and* $(\sigma_1, s_1) \xrightarrow{fair}{}^{*} (\sigma_1', s_1')$ *then*
$\exists\sigma_2', s_2'.\ (\sigma_2, s_2) \xrightarrow{fair}{}^{*} (\sigma_2', s_2')$ *and* $\sigma_1' \approx_l \sigma_2'$.

Recently, Ngo et al. [41] call *indirect termination sensitive non-interference* (ITSNI) to security conditions (like ours) where the termination behavior of sensitive programs is not exposed via public inputs and outputs despite their divergence. In this work, however, we refer to our security condition as TSNI since it is a more widely accepted term.[3]

The fair semantics is also *transparent*, in that it does not perturb the behavior of non-interfering programs. We consider a *program* to be any term t without facets (i.e., without any secrets). We say a program t is *non-interfering* if running t with two l-equivalent inputs $I_1 \approx_l I_2$ gives l-equivalent behavior, i.e. if

$$(t, \emptyset, \lambda i.0, I_1, \lambda o.\epsilon) \xrightarrow{\text{std}}{}^{*} \sigma_1$$

then there is some $\sigma_2 \approx_l \sigma_1$ such that

$$(t, \emptyset, \lambda i.0, I_2, \lambda o.\epsilon) \xrightarrow{\text{std}}{}^{*} \sigma_2$$

Here, $(t, \emptyset, \lambda i.0, I_1, \lambda o.\epsilon)$ is the initial state for running t with the empty memory, 0-initialized buffer pointers, input I_1, and empty output buffers.

For such programs that are non-interfering under the standard semantics, the fair faceted semantics does not change behavior.

Theorem 5 (Transparency).
Consider any standard run $\sigma = (t, \emptyset, \lambda i.0, I, \lambda o.\epsilon) \xrightarrow{std}{}^{*} \sigma'$ *of a non-interfering program* t. *For all* $l \in$ *Lattice, the fair semantics generates a corresponding run*

$$(\sigma, s) \xrightarrow{fair}{}^{*} (\sigma'', s'')$$

with $\sigma' \approx_l \sigma''$. *In particular, all* l*-visible output buffers in* σ' *and* σ'' *are identical.*

7 DECENTRALIZED LABELS

In our framework, the semantic rule for run determines when multi-executions are necessary. To recap briefly, this rule has the following side conditions (recall Figure 2) for a given *pc* and label k.

$$\begin{aligned} views(pc \cup \{\overline{k}\}) &= \emptyset \\ views(pc \cup \{k\}) &= \emptyset \end{aligned}$$

[3]More precisely, our security condition is progress-sensitive non-interference[35]: it ensures that information is not leaked via termination even in the presence of outputs.

Recall that the definition of $views(pc)$ hinges on quantifying over all labels in the lattice. The definition of $views(pc)$ in Section 3 is:

$$views(pc) = \{l \in Lattice \mid (\forall k \in pc.\ k \sqsubseteq l) \wedge (\forall \overline{k} \in pc.\ k \not\sqsubseteq l)\}$$

Where l ranges over labels in the lattice. The reader may be worried that this definition means that our calculus is not applicable to infinite, decentralised, lattices, a severe restriction to real-world applicability would it be the case. In this section, we show that the condition $views(pc) = \emptyset$ is decidable given that the lattice has a decidable ordering relation ($\sqsubseteq$) and computable join ($\sqcup$)—a novelty with respect to previous work (e.g., [4, 39]) that assume either finite lattices or lattices with just a confidentiality component.

We introduce the notion of a *candidate label* for a given *pc*, defined as

$$l_c(pc) = \bigsqcup \{k \mid k \in pc\}$$

which is the smallest label that must be in $views(pc)$. To check if $views(pc)$ is non-empty, we simply check that for any negated label $\overline{k} \in pc$, k does not flow into this candidate label.

Theorem 6 (Emptiness Check).

$$\forall pc.\ views(pc) \neq \emptyset \Leftrightarrow \forall \overline{k} \in pc.\ k \not\sqsubseteq l_c(pc)$$

This theorem gives us a decision procedure for finite PCs when the lattice has decidable ($\sqsubseteq$) and computable ($\sqcup$): it guarantees that we are not limited in our choice of lattice when instantiating **Multef**. One consequence of this result is that **Multef** can use practical decentralised label models like DC-labels [49] and DLM [34].

7.1 Disjunction Category Labels

Disjunction Category (DC) Labels is a decentralized labeling scheme whereby labels are represented as pairs of finite monotonic propositional logical formulas, i.e., logical formulas without negation or implication. The atoms in the formulae represent actors in the system. Each label consists of two such formulas, one expressing a confidentiality and the other an integrity requirement.

A DC label, then, is a tuple $\langle C, I\rangle$, where C stands for confidentiality and I for integrity. When it comes to confidentiality, conjunctions represent the multiple interest of principals to protect the data, while disjunctions denote groups wherein any member may learn the information. For instance, the formula `Alice` $\wedge$ `Bob` indicates that information is sensitive to both principals and requires their joint consensus to observe it. In contrast, `Alice` $\vee$ `Bob` reflects that data can be observed either by one of the principals. Dually, when it comes to integrity, conjunctions of principals represent groups of principals where members are independently responsible for the information. As a example, the formula `Alice` $\wedge$ `Bob` means that Alice is completely responsible for the data, and so is Bob. Conversely, disjunctions of principals represent groups that *collectively* take responsibility for the information, i.e., no single principal takes full responsibility. For example, the formula `Alice` $\vee$ `Bob` means that Alice and Bob collectively are responsible for the data—both may have contributed to or influenced it. This notion of labels is general enough to encode the label models used in many IFC operating systems (e.g., Asbestos [21], HiStar[58], and Flume [29]) as well as a subset of DLM [34].

DC Labels form a lattice where the definition of the ordering (can-flow-to) relation $\sqsubseteq$ is as follows.

$$\frac{C_1 \vdash C_0 \quad I_0 \vdash I_1}{\langle C_0, I_0\rangle \sqsubseteq \langle C_1, I_1\rangle}$$

The sequent $A \vdash B$ should be read "given the assumption A, we can prove B using the rules of propositional logic." As an example, let us consider the DC label $L_1 = \langle \text{Bob}, \text{Bob} \vee \text{Alice}\rangle$, where data is confidential to Bob but he does not assume full responsibility for it, and label $L_2 = \langle \text{Bob} \wedge \text{Alice}, \text{Bob}\rangle$ where data is confidential to both principals but Bob assumes responsibility for it. Can data label with L_1 flow into entities label with L_2, i.e., $L_1 \sqsubseteq L_2$? When it comes to confidentiality, it holds that Alice $\wedge$ Bob $\vdash$ Bob. However, Alice $\vee$ Bob $\nvdash$ Bob; otherwise Bob would assume full responsibility for information that he has not completely vouched for, wherefore $L_1 \not\sqsubseteq L_2$. Note that for any pair of labels ℓ and ℓ' the statement $\ell \sqsubseteq \ell'$ is decidable using standard techniques like SAT solvers or BDDs [1, 20].

The join ($\sqcup$) of two labels is also easily constructed by taking the conjunction of the confidentiality components and the disjuction of the integrity components.

$$\langle C_0, I_0\rangle \sqcup \langle C_1, I_1\rangle = \langle C_0 \wedge C_1, I_0 \vee I_1\rangle$$

With computable join ($\sqcup$) and decidable ordering ($\sqsubseteq$) we obtain a full decision procedure for emptyness of view of finite PCs under DC-labels, thus **Multef** can naturally support expressive DC-labels.

8 IMPLEMENTATION

In this section, we give an overview of the implementation of **Multef**. Particularly, we describe some technical problems to overcome in order to deliver **Multef** as a Haskell library. Our implementation supports references and I/O, and is easily extended with any effects that can be accommodated by our formal results. **Multef** can be used as a basis to implement IFC-secure plugins and applications.

8.1 Basic structures

We begin by representing labels and program counters as data types in Haskell.

```
data Label -- Kept abstract for this presentation
data Branch = Private Label | Public Label
type PC     = [Branch]
```

We use the syntax `[a]` for denoting the type of lists of elements of type `a` and `x:xs` to denote the insertion `x` at the head of the list `xs`. The decision procedure described in Section 7.1 for deciding if a view is empty is named but kept abstract in the interest of brevity.

```
isEmptyViews :: PC -> Bool
```

Faceted values are implemented as the following data type [26].

```
data Fac a where
  Raw  :: a -> Fac a
  Bind :: Fac a -> (a -> Fac b) -> Fac b
  Q    :: Label -> Fac a -> Fac a -> Fac a
```

The constructors `Raw`, `Bind`, and `Q` (for question mark) correspond to the constructors raw, bind, and $\langle \bullet ? \bullet : \bullet \rangle$ in our calculus, respectively. With faceted values in place, we proceed to provide the FIO operations in our calculus.

```
data FIO a where
  Return  :: a -> FIO a
  (:>>=:) :: FIO a -> (a -> FIO b) -> FIO b
  Run     :: Fac (FIO a) -> FIO (Fac a)
  -- Primitives for references and I/O
  ...
```

Similarly to `Fac`, the constructors of `FIO` denote different operations used to build terms of type FIO—a standard approach taken when representing domain-specific languages (DSLs) in Haskell [52]. For brevity, we focus only on constructors representing return, `:>>=:`, and run, and we refer the interested reader to Appendix B for further details.

8.2 Executor commonalities

Our goal is to implement three executors for programs of type `FIO a` so that, by changing the executor, we can execute programs under MF, MF-par, SME, or FSME. Ideally, we want our executors to have the same type and to "factor out" their common behavior as much as possible. With this in mind, we propose the following type for the executors: `FIO a -> PC -> IO (a, PC)`, i.e., it takes a `FIO`-program and an initial *pc* (`PC`), and returns a (possibly) side-effectful program which produces a result of type `a` and a final *pc* (`IO (a,PC)`). In Haskell, the special data type `IO r` denotes programs that might perform side-effects (e.g., writing to a file) and return values of type `r`.

We start by defining the executor `execute` as a base implementation of all the commonalities across the multi-executions techniques.

```
execute :: FIO a -> PC -> IO (a, PC)
-- Def. monadic FIO primitives
execute (Return a)       = return (a, pc)
execute (fio :>>=: rest) = do
   (a, pc) <- execute fio pc
   execute (rest a) pc
-- Def. for references and I/O
...
```

The code skeleton above shows how to execute the monadic FIO-primitives in a manner that is common to all the multi-execution techniques—we omit those for references and I/O for brevity and simplicity. More precisely, `Return` simply maps to the return in `IO` (i.e., `return (a,pc)`). The bind operator (`:>>=:`) is defined as expected: it reduces the given `fio` computation and passes its result of type `a` to `rest` and executes the resulting FIO computation (i.e., `execute (rest a) pc`). According to Figure 2, the behavior of many FIO-operations are common to all the multi-executions techniques supported by our calculus. It is easy to show that the cases in the definition of `execute` corresponds to the semantic rules in Appendix A, Figure 12. For instance, `execute (Return t :>>=: rest)` is equivalent to `execute (rest t)`—thus matching the rule [F-BIND-FIO] in Figure 10. The interesting part of implementing `execute` arises from evaluating `Run`, the constructor responsible of introducing multi-executions. For `Run`, it is not possible (as expected) to have a common code for all the different multi-execution techniques.

8.3 MF executor

We show here the behavior of `Run` in the MF executor.

```
execute (Run (Q k priv publ)) pc
  | isEmptyViews (Public  k : pc) -> execute (Run priv) pc
  | isEmptyViews (Private k : pc) -> execute (Run publ) pc
  | otherwise          -> do
     (priv', _) <- execute (Run priv) (Private k : pc)
     (publ', _) <- execute (Run publ) (Public  k : pc)
     return (Q k priv' publ', pc)
```

As in our formal calculus, the definition consists of three cases divided by the symbol `|`. The first cases are triggered when `pc` can observe only the private (see rule [F-RUN-FACET-1]) or public facet (see rule [F-RUN-FACET-2]), respectively. When it comes to the `otherwise` case, the MF executor *sequentially* evaluates the private and public facets, respectively—observe the recursive calls with the *pc*s `Private k : pc` and `Public k : pc`, respectively. The resulting faceted value, `Q k priv' publ'` (aka $\langle k \,?\, priv' : publ' \rangle$), is constructed with the result of these evaluations. This implementation corresponds to the applications of rules [F-THREAD-1], then [F-THREAD-2], and finally [F-MERGE] in our calculus.

MF-par executor. We also implement a slight variation of the MF executor above called the *MF-par* executor. This executor essentially runs the private and public sub-computations in parallel, which then gives different performance characteristics. Observe that this variation is supported by our formal framework in Section 3.

8.4 Continuations and SME

We now turn to trying to implement our SME executor for the same representation of programs used above. However, we run into a problem, it is impossible to make the executor correspond to the calculus. The key observation is that when spawning the new thread, we not only want to execute the instruction `Run priv` under the *pc* `Private k : pc` but also the rest of the program! Imagine we wish to execute the program `Run (Q k priv pub) :>>=: rest`. If we just execute `fork (execute (Run priv) (Private k : pc))` under the `otherwise` guard, we will end up not running `rest` for the private view. The problem lies in the interaction between `:>>=:` and `Run`. More precisely, when evaluating `Run`, the executor has no access to the "rest of the program." Note that evaluation contexts denote the rest of the program, so this problem does not exist in our formal semantics and only materialises in practise.

There are two possible solutions to the problem outlined above, the first is to change the type of the executors to reflect the need for keeping track of the "rest of the program" via continuations. Unfortunately, the new type quickly becomes cluttered.

Instead, we choose a simpler approach: to remove the troublesome `(:>>=:)` construct without loosing any expressive power in our language. For that, we apply a known technique for domain-specific languages (DSL) [12] for deriving alternate implementations of APIs. In a nutshell, what we will do is to replace the constructor `Run` with a new one called `RunBind` such that its semantics is determined by the equation `RunBind fac rest` ≡ `(Run fac) :>>=: rest`. We change our implementation of `FIO` as follows.

```
data FIO a where
  Return  :: a -> FIO a
  RunBind :: Fac (FIO a) -> (Fac a -> FIO b) -> FIO b
  -- Primitives for references and I/O
  ...
```

The type form of `RunBind` arises from its semantics definition. We can now *soundly derive* an implementation of a bind function `(>>=) :: FIO a -> (a -> FIO b) -> FIO b` by simply applying `RunBind`'s semantics. In other words, whatever `FIO`-program was built before using the constructor `:>>=:`, it can be obtained with function `(>>=)` without changing its semantics—see Appendix B for details.

With this new representation, we can write the behavior of `RunBind` for SME.

```
execute (RunBind (Q k priv pub) rest) pc
  ...
  | otherwise -> do
    fork (execute (RunBind priv rest) (Private k : pc))
    execute (RunBind pub rest) (Public k : pc)
```

Observe that `rest` contains "the rest of the program", which then gets evaluated twice as expected, i.e., once for each view. The MF executor is also easily adjusted to accomodate this new representation—see Appendix B for the details.

8.5 FSME executor

Implementing the FSME executor requires careful thought. It involves setting a timeout that, when triggered, causes the execution to be split into two separate executions. The splitting, however, needs to be done in a safe manner, e.g., not in the middle of an output. To achieve that, when hitting the *otherwise* guard, our executor spawns a thread to compute the private facet, send the result to a pre-determined location, and wait for what to do next. In contrast, the thread for the public facet sets a timeout to check if the result of the private facet arrived on time. If that is the case, then the thread for the public facet indicates to the private one to terminate; otherwise, it sends a signal to compute the "rest of the program" in the separate thread. The notion of the continuation in the constructor `RunBind` turns out to be essential to implementing this approach. Unfortunately, explaining the implementation of this executor requires explaining some synchronisation and concurrency primitives in Haskell. For the sake of brevity, we refer to the interested reader to Appendix C for the details.

9 EVALUATION

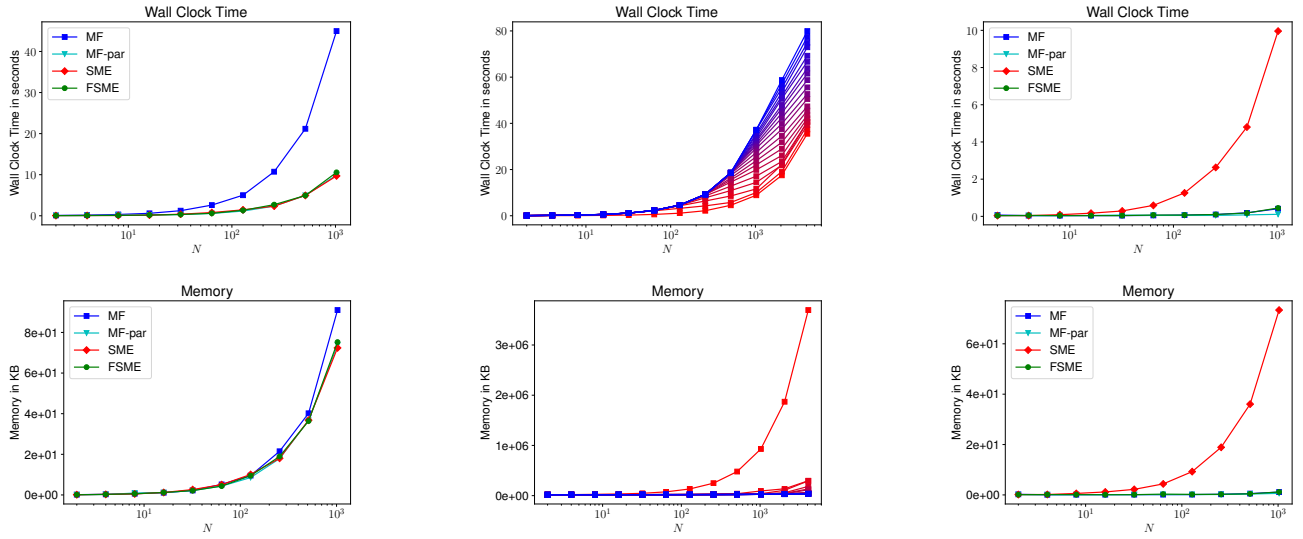
We next evaluate the performance of our four executors (MF, MF-par, SME, and FSME) on several micro-benchmarks. Suppose we have n principals/actors, which we formalize as n incomparable labels $l_1, \ldots, l_n \in Lattice$. Let $s_i = \langle l_i \,?\, \ldots : \ldots \rangle$ be a string secret to label l_i. Then the concatenation of these n strings generates a faceted tree s with height n and 2^n leaves. Computations over s thus may generate $N = 2^n$ subcomputations over the leaves, and so we use s as a suitable faceted value to stress the implementation of `RunBind`'s `otherwise` guard.

We now define an expensive function on faceted values.

```
benchmark1 :: Int -> Fac String -> FIO (Fac String)
benchmark1 n fac =
  RunBind (Bind fac
                (\s -> Raw (Return (hashes n s))))
          Return
```

This function takes a faceted value and computes nested hashes on all its leaves. Function `hashes n s` computes `n` nested SHA256 hashes of the string `s`.

Figure 6a shows the performance characteristics for our executors when executing `(benchmark1 100000 s)`. The measurements were taken on a 2.8GHz 4 core `Intel Core i7-7700HQ` processor. Note that the MF-par, SME, and FSME executors run roughly 4 times



(a) 10^5 **rounds of SHA256 for a faceted value with** N **leaves**

(b) 10^5 **rounds of SHA256 for a faceted value with** N **leaves using different timeouts in FSME. Red is a shorter timeout and blue is a longer timeout.**

(c) 10^5 **rounds of SHA256 after branching on a faceted value with** N **leaves (FSME coincides with MF)**

Figure 6: Time and memory consumption for different micro-benchmarks

faster than MF, due to parallelism. Interestingly, the memory consumption, measured in peak resident set size, is significantly larger for MF-par and SME than for MF. This is a result of SME spawning additional threads which need to be represented in the Haskell runtime, whereas the MF executor only keeps the current task in memory.

The performance of FSME sits between MF and SME, obtaining the best of both worlds. Figure 6a shows that FSME gains speedup while keeping memory consumption close to MF most of the time. What we observe is that the timeout mechanism implemented by FSME is triggered early enough to obtain only a few threads. From that point on, the program is run in parallel; however, within the threads, the execution continues mainly under a MF semantics, i.e., the timeout mechanisms subsequently does not get triggered frequently. These results were obtained with a timeout of 1.5 seconds.

We also ran the same benchmark described above for timeouts varying from 0 to 20 seconds, going from full SME closer to MF. Figure 6b shows the result of this experiment. The graphs go from red, indicating a low timeout (SME-like semantics), to blue indicating a large timeout (MF-like semantics). Interestingly, imposing any non-zero timeout, 1 second in the example, drastically reduces memory consumption. This is also the case for even smaller timeouts, like 0.25 and 0.1 seconds.

It is worth noting that while variations in timeout impact performance, the security implications of the timeout are not as severe. Regardless of the length of the timeout, non-terminating computations will always encounter it. However, if we take terminating computations, and we take a sufficiently long timeout, we can run everything just as MF.

The performance of SME versus MF seen so far may give the impression that SME is always faster than MF at the cost of an increased memory footprint. However, Figure 6c shows evidence of the contrary. For this benchmark, instead of taking the hash of the faceted value, we take the hash of a constant value after branching on a faceted one.

```
benchmark2 :: Int -> Fac () -> FIO (String)
benchmark2 n fac =
  RunBind (Bind fac
                (\() -> Raw (Return ()))
           (\f -> Return (hashes n "hello"))
```

In this benchmark, SME is exponentially slower than MF. The reason for this is that every time that `benchmark2` branches on a faceted value, it duplicates the continuation (`\f -> Return (hashes n "hello")`). As a result, the expensive computation (`hashes n "hello"`) executes many times even though it does not depend on the faceted value. MF, MF-par, and FSME, on the other hand, run all the inexpensive computation first (i.e., `Raw (Return ())`), i.e., once for every leaf in the faceted value, and subsequently executes the hashing function only once.

10 PROTECTEDBOX

In order to demonstrate the viability of our framework for building practical IFC systems, we have implemented a prototype service called ProtectedBox. ProtectedBox is a essentially an API for the cloud storage solution Dropbox [18] that makes possible to securely write and execute (mutually distrust) third-party plugins on users' files. Plugins are written in **Multef** extended with I/O primitives specific to the Dropbox API [19].

10.1 Labeling policy

File owners specifies how information can be shared with different plugins. Initially, every file in `User`'s folders are labeled as

{⟨User, User⟩}, thus indicating that the files are confidential to the principal (or source of authority) User and that User is responsible for its content. We consider plugins as another source of authority. In this light, a given plugin named Plugin is considered a principal whose initial PC corresponds to {⟨⊥, Plugin⟩}—so the plugin does not have any confidentiality requirements a priori. Below, we describe three plugins that we implemented for ProtectedBox as well as the labeling discipline that they follow.

► Comments: this plugin allows the user to add comments to a file. The comments are stored in a different file with label ⟨User, User ∨ Comments⟩. This indicates that the content of the comments is confidential to the user, but might have been affected by either the user or the plugin.

► Tarball: this plugin creates a tarball of several files. The tarball is labeled with the least upper bound of all the files in the tarball joined with ⟨⊥, Tarball⟩ to indicate that the plugin may have influenced the contents of the files, i.e., the tarball gets the label $(\bigsqcup l_f) \sqcup \langle \bot, \text{Tarball} \rangle$.

► Checksum: This plugin computes the SHA256 hash of a file and saves it to another file. The file created by the plugin is labeled as $l_f \sqcup \langle \bot, \text{Checksum} \rangle$. This means that the checksum is as confidential as the file it comes from but that Checksum might have influenced its content.

Plugins are restricted from arbitrarily querying information about folders (e.g., list of files) and files (e.g., update time, etc.) in order to avoid leaks of information via many different covert channels [30]. Instead, they have access to the following file-specific API and, of course, the primitives of our framework.

```
-- Interact with user files
createFile :: Label  -> String -> String -> FIO ()
writeFile  :: String -> String -> FIO ()
readFile   :: String -> FIO (Faceted (Maybe String))
```

A read operation on a file with label l returns the faceted value $\langle l \,?\, \texttt{contents} : \bot \rangle$. Similarly, writes to a file with label l only happens if $l \in \mathit{views}(pc)$, similarly to the semantics of put. The same goes for creating files, a file can only be created if its label is in the view of the *PC*.

10.2 Performance

We have evaluated the performance overheads associated with our executors in ProtectedBox. We have five different FIO executors, MF, MF-par, SME, FSME, and STD. The latter is analogous to $\xrightarrow{\text{std}}$ in that it never introduces faceted values, only deals with raw values, and provides no security guarantees.

Figure 7 shows the performance characteristics when running the Tarball plugin on up to 30 files. As can be seen from the figure, our secure executors (MF, MF-par, SME, FSME) do not introduce extraneous overheads over the unsecure STD executor. All executors had the same memory footprint in this experiment. The total memory overhead was small, measured in a few hundred KB at most. This benchmark provides evidence that, in the case of non-malicious plug-ins, the performance is similar for the different multi-execution approaches. Malicious code, however, may stress the system in ways like what is shown in Section 9.

The performance is dominated by network overheads. For this reason it is important that the safe executors do not introduce large

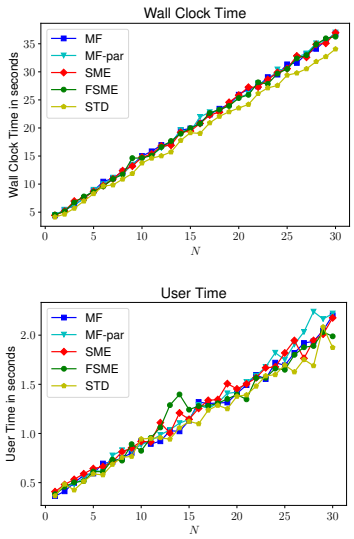


Figure 7: Time for different executors in the Tarball benchmark, where N is the number of files.

numbers of sequential requests. The code under test in Figure 7 does not display such weakness. It is possible to construct programs similar to the first benchmark in Section 9 which introduce an exponential number of network requests, these programs degrade performance differently under MF, MF-par, SME and FSME in a way similar to the results in Section 9. However, due to throttling from the Dropbox API we have been unable to thoroughly evaluate scenarios of this kind in ProtectedBox, but tentative experiments suggest that the effect exist.

11 RELATED WORK

SME. The idea of utilizing multi-executions to secure programs has been independently proposed by many researchers. Capizzi et al. [10] propose running two copies of the same program, so called *shadow executions*: one for public and other for handling private data, respectively. Cristiá and Mata independently formalize a similar system at the operating system level [13]. Devriese and Piessens [16] coin the term SME and are the first to formalise the soundness and precision guarantees of the approach. Different from our approach, the original formulation of SME is *black-box*, i.e, language independent, which makes it possible to deploy it for complex languages like JavaScript. Jaskelioff and Russo [24] present an implementation of SME in Haskell in less than 150 lines of code. Barthe et al. [6] propose a program that inlines SME into JavaScript-like programs—so that it is not necessary to modify the runtime system to obtain multi-executions. We believe that our contributions could be used to extend the approaches above to work on decentralized labels as well as obtaining multi-executions "on-demand." When it comes to applications, the web has been the chosen domain to test SME ideas [7] and their implementations, e.g., FlowFox [14]. The implementation accompanying [7] handles SME for a specific infinite lattice with levels L (public or bottom), H (secret or top), and $M(d)$ for every incomparable web domain d. When receiving an event from an unseen domain, the enforcement creates a copy of the browser's state which gets initialized with the L-state—which is only suitable under the considered lattice.

Instead, our work allows for more general infinite lattices and initialization of multi-executions' states without loosing soundness or transparency guarantees. SME has also been successfully applied to the map-reduce programming model [40]. When it comes to security guarantees, secure programs interpreted under SME produce the same outputs as if they were run under a standard semantics *modulo the relative ordering of observable events from different security levels*. The work in [28] explores how different scheduling policies affect the security guarantees provided by SME, i.e., TINI or TSNI. In [43, 57], the authors combine scheduling techniques with monitoring approaches to guarantee that interleaving of events gets preserved for secure programs. The authors of [43, 53] provide means for declassification. While our framework does not present means for declassification, we state as future work adapting such techniques for a functional language.

MF. Austin and Flanagan introduce MF semantics [5], where authors refer to it as an optimization for SME. Schmitz et al. [46] show an implementation of MF in Haskell—part of that design inspired ours. Bielova and Rezk [8] later show that SME and MF are actually different: they differ on the provided security guarantees (i.e., TINI vs. TSNI) and the treatment of default values. They propose an *all-or-nothing* combination of MF and SME using a non-decidable semantics—which takes decisions based on the termination behavior of commands. Their enforcement run programs under a MF semantics but switches to SME (with a low priority scheduler) when commands inside a branch do not terminate. In the same all-or-nothing spirit, Ngo et al. [39] combine MF and SME techniques for a simple while-language, where timeouts are set to determine when to switch to SME. These works and ours share similar goals, but the underlying mechanisms are entirely different. One obvious difference is that we use a monad-based operational semantics vs. a while-like language. From the enforcement perspective, our technique uses a decidable semantics (unlike [8]) and spawns multi-executions on-demand while [39] does not, thus duplicating memory and execution of code. Furthermore, their switching mechanism between MF and SME requires knowledge of all points in the lattice, something which is not feasible in decentralized lattices like DC-labels (or DLM). Different from that work, **Multef** supports decentralized labeling models and it does not spawn as many threads as security labels when providing termination-sensitive guarantees. Schoepe et al. [47] investigate how to apply MF semantics to encode taint analysis.

IFC libraries. Many IFC security libraries exists for Haskell. They can enforce non-interference statically [2, 32, 44, 54], dynamically [50], or as a combination of both [9, 17]. Many of these libraries utilize the concept of monads to control the side-effects that programs are allowed to perform. Differently from them, our work (library) uses monads to adapt programs semantics to MF, SME, or FSME.

12 CONCLUSIONS

MF and SME are two promising approaches to dynamic IFC that provide complementary benefits—MF provides better performance, whereas SME provides stronger termination-sensitive security guarantees. This paper provides the unifying framework **Multef**, a synthesis of both prior approaches in the form of both a unifying formal semantics and a corresponding Haskell IFC library. Using **Multef**, we have developed Faceted Secure Multi Execution, which combines the performance benefits and termination-sensitive guarantees of MF and SME, respectively. In addition, our work supports decentralized labels, necessary in many realistic settings.

We believe the our mechanically-verified semantics and IFC library provide a solid foundation for the future development of extensions as well as realistic applications with strong IFC-based security guarantees. We envision as future work to extend **Multef** to support exceptions and timing-sensitive guarantees. Specifically, we expect to need some mechanism for propagating exceptions across threads for MF- and FSME-based multi-executions. On the other hand, when it comes to timing guarantees, we believe it is possible to leverage some existing results to make FSME robust against timing leaks—perhaps by assuming a specific scheduler [28], or perhaps by padding the sensitive computations by the chosen timeout [3].

ACKNOWLEDGMENTS

We would like to thank Tamara Rezk and Nataliia Bielova for initial discussions on this work as well as the anonymous reviewers for their helpful comments. This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and WebSec (Ref. RIT17-0011), the Swedish research agency Vetenskapsrådet, and NSF Grants 1337278 and 1421016.

REFERENCES

[1] Sheldon B. Akers. 1978. Binary decision diagrams. *IEEE Transactions on computers* 6 (1978), 509–516.
[2] Maximilian Algehed and Alejandro Russo. 2017. Encoding DCC in Haskell. In *Proc. of the 2017 Workshop on Programming Languages and Analysis for Security (PLAS '17)*. ACM.
[3] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. 2010. Predictive black-box mitigation of timing channels. In *Proc. of the 17th ACM conference on Computer and Communications Security*. ACM.
[4] Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*. 165–178.
[5] Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '12)*. ACM.
[6] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. 2012. Secure multi-execution through static program transformation. In *Formal Techniques for Distributed Systems (FMOODS/FORTE 2012)*.
[7] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. 2011. Reactive non-interference for a browser model. In *Proceedings of the 5th International Conference on Network and System Security (NSS 2011),*.
[8] Nataliia Bielova and Tamara Rezk. 2016. Spot the Difference: Secure Multi-execution and Multiple Facets. In *European Symposium on Research in Computer Security*. 501–519.
[9] P. Buiras, D. Vytiniotis, and A. Russo. 2015. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM.
[10] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. 2008. Preventing Information Leaks through Shadow Executions. In *Proc. of the Annual Computer Security Applications Conference (ACSAC '08)*. IEEE Computer Society.
[11] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. 2017. Nonmalleable Information Flow Control. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*. 1875–1891.
[12] Koen Claessen. 2004. Parallel Parsing Processes. *Journal of Funcitonal Programming* 14, 6 (2004), 741–757.
[13] Maximiliano Cristiá and Pablo Mata. 2009. Runtime Enforcement of Noninterference by Duplicating Processes and their Memories. In *Workshop de Seguridad Informática WSEGI 2009, Argentina (38 JAIIO)*.
[14] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: a web browser with flexible and precise information flow control.

In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12)*. ACM, New York, NY, USA.
[15] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (July 1977), 504–513. https://doi.org/10.1145/359636.359712
[16] D. Devriese and F. Piessens. 2010. Noninterference through Secure Multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society.
[17] D. Devriese and F. Piessens. 2011. Information flow enforcement in monadic libraries. In *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '11)*. ACM.
[18] Dropbox. [n. d.]. Dropbox. https://www.dropbox.com. ([n. d.]).
[19] Dropbox. [n. d.]. Dropbox HTTP API. https://www.dropbox.com/developers/documentation/http/overview. ([n. d.]).
[20] Niklas Eén and Niklas Sörensson. 2003. An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*. Springer, 502–518.
[21] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. 2005. Labels and event processes in the Asbestos operating system. In *Proc. of the twentieth ACM symp. on Operating systems principles (SOSP '05)*. ACM.
[22] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. 2012. Hails: Protecting Data Privacy in Untrusted Web Applications. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI*.
[23] J.A. Goguen and J. Meseguer. 1982. Security policies and security models. In *Proc of IEEE Symposium on Security and Privacy*. IEEE Computer Society.
[24] M. Jaskelioff and A. Russo. 2011. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics (LNCS)*. Springer-Verlag.
[25] Mark P Jones and Luc Duponcheel. 1993. *Composing monads*. Technical Report. Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale University.
[26] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *Proc. of the ACM SIGPLAN International Conf. on Functional Programming, ICFP*.
[27] Simon Peyton Jones (Ed.). 2003. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England.
[28] V. Kashyap, B. Wiedermann, and B. Hardekopf. 2011. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In *Proc. of IEEE Symposium on Sec. and Privacy*. IEEE.
[29] Maxwell N. Krohn, Alexander Yip, Micah Z. Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Tappan Morris. 2007. Information flow control for standard OS abstractions. In *Proc. of the 21st ACM Symposium on Operating Systems Principles*. 321–334.
[30] B. W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16, 10 (Oct. 1973).
[31] P. Li and S. Zdancewic. 2006. Encoding Information Flow in Haskell. In *Proc. of the IEEE Workshop on Computer Security Foundations (CSFW '06)*. IEEE Computer Society.
[32] P. Li and S. Zdancewic. 2010. Arrows for secure information flow. *Theoretical Computer Science* 411, 19 (2010), 1974–1994.
[33] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. 2009. Fabric: A Platform for Secure Distributed Computation and Storage. In *Proc. of the ACM SIGOPS Symposium on Operating Systems Principles*. ACM.
[34] B. Montagu, B.C. Pierce, and R. Pollack. 2013. A Theory of Information-Flow Labels. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*.
[35] Scott Moore, Aslan Askarov, and Stephen Chong. 2012. Precise enforcement of progress-sensitive security. In *the ACM Conference on Computer and Communications Security, CCS'12*.
[36] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From General Purpose to a Proof of Information Flow Enforcement. *2012 IEEE Symposium on Security and Privacy* 0 (2013).
[37] Andrew C Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9, 4 (2000), 410–442.
[38] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. 2001. Jif: Java Information Flow. (2001). http://www.cs.cornell.edu/jif.
[39] Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. 2018. A better facet of dynamic information flow control. In *The Web Conference. Research track: Security and privacy of the Web. (WWW'18)*.
[40] Minh Ngo, Fabio Massacci, and Olga Gadyatskaya. 2013. MAP-REDUCE Runtime Enforcement of Information Flow Policies. *CoRR* (2013). http://arxiv.org/abs/1305.2136
[41] M. Ngo, F. Piessens, and T. Rezk. 2018. Impossibility of Precise and Sound Termination-Sensitive Security Enforcements. In *IEEE Symposium on Security and Privacy (SP)*.
[42] S. Peyton Jones, A. Gordon, and S. Finne. 1996. Concurrent Haskell. In *Proc. of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96)*. ACM.
[43] Willard Rafnsson and Andrei Sabelfeld. 2013. Secure multi-execution: fine-grained, declassification-aware, and transparent. (Feb. 2013). Submitted.
[44] A. Russo, K. Claessen, and J. Hughes. 2008. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN symposium on Haskell (HASKELL '08)*. ACM.
[45] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proc. of the 2010 23rd IEEE Computer Security Foundations Symp. (CSF '10)*. IEEE Computer Society, 186–199.
[46] Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. 2016. Faceted Dynamic Information Flow via Control and Data Monads. In *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*.
[47] Daniel Schoepe, Musard Balliu, Frank Piessens, and Andrei Sabelfeld. 2016. Let's Face It: Faceted Values for Taint Tracking. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*.
[48] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. 2011. Disjunction Category Labels. In *Proc. of the Nordic Conference on Information Security Technology for Applications (NORDSEC '11)*. Springer-Verlag.
[49] Deian Stefan, Alejandro Russo, David Mazières, and John C Mitchell. 2011. Disjunction category labels. In *Nordic conference on secure IT systems*. Springer.
[50] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*.
[51] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. 2014. Protecting Users by Confining JavaScript with COWL. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association.
[52] Wouter Swierstra and Thorsten Altenkirch. 2007. Beauty in the Beast: A Functional Semantics of the Awkward Squad. In *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell*. 25–36.
[53] Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. 2014. Stateful declassification policies for event-driven programs. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE.
[54] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. 2017. MAC A Verified Static Information-Flow Control Library. *Journal of Logical and Algebraic Methods in Programming* (2017). https://doi.org/10.1016/j.jlamp.2017.12.003
[55] Philip Wadler. 1993. Monads for functional programming. In *Program design calculi*. Springer, 233–264.
[56] Philip Wadler. 1995. Monads for functional programming. In *International School on Advanced Functional Programming*. Springer, 24–52.
[57] Dante Zanarini, Mauro Jaskelioff, and Alejandro Russo. 2013. Precise Enforcement of Confidentiality for Reactive Systems.. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE, 18–32.
[58] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in HiStar. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation*. USENIX.
[59] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*. 503–516.

A SEMANTICS AND PROOF SKETCHES

This appendix presents the full syntax, type system, and semantics of our language as well as our security guarantee results and proof sketches. The full syntax can be found in Figures 8 and 9 along with the semantics in Figure 10 and the type system in Figure 11. Figure 12 shows the full standard semantics. Next we go through our security guarantees as well as their respective proof sketches.

Theorem 2 (Termination-Insensitive Non-Interference). *If* $\sigma_1 \approx_l \sigma_2$ *and* $\sigma_1 \longrightarrow_\emptyset^* \sigma_1' \not\longrightarrow_\emptyset$ *and* $\sigma_2 \longrightarrow_\emptyset^* \sigma_2' \not\longrightarrow_\emptyset$ *then* $\sigma_1' \approx_l \sigma_2'$.

Proof sketch

By repeated application of Projection 1, and by using Projection 2, we have $\sigma_1\downarrow_l \xrightarrow{\text{std}}{}^* \sigma_1'\downarrow_l \not\xrightarrow{\text{std}}$ and $\sigma_2\downarrow_l \xrightarrow{\text{std}}{}^* \sigma_2'\downarrow_l \not\xrightarrow{\text{std}}$. Since $\xrightarrow{\text{std}}$ is deterministic and $\sigma_1 \approx_l \sigma_2$, therefore $\sigma_1' \approx_l \sigma_2'$, as desired. □

Theorem 3 (Fair Projection).
If $\sigma\downarrow_l \xrightarrow{std} \sigma_1$ *then* $\exists\sigma_2, s_2.\ (\sigma, s) \xrightarrow{fair}{}^{*} (\sigma_2, s_2)$ *and* $\sigma_2\downarrow_l = \sigma_1$.

Proof sketch
By strong induction on measure(l, σ), which is roughly defined as the sum of 2^{depth} of each occurence of $\langle \bullet\ ?\ \bullet : \bullet \rangle$ or $\langle\!\langle \bullet\ ?\ \bullet : \bullet \rangle\!\rangle$ in the program, ignoring subterms that are not visible to l and ignoring the right hand subterms of any occurrences of bind. This number represents an upper bound on the number of invisible (to l) steps that σ can take. Also, do induction on the number n mentioned in the definition of Fairness. □

Theorem 4 (Termination-Sensitive Non-Interference).
If $\sigma_1 \approx_l \sigma_2$ *and* $(\sigma_1, s_1) \xrightarrow{fair}{}^{*} (\sigma_1', s_1')$ *then* $\exists\sigma_2', s_2'.\ (\sigma_2, s_2) \xrightarrow{fair}{}^{*} (\sigma_2', s_2')$ *and* $\sigma_1' \approx_l \sigma_2'$.

Proof sketch
By Scheduler Validity, we have $\sigma_1 \longrightarrow^{*}_{\emptyset} \sigma_1'$. By Projection 1, we have $\sigma_1\downarrow_l \xrightarrow{\text{std}}{}^{*} \sigma_1'\downarrow_l$. Now because $\sigma_1 \approx_l \sigma_2$, we have $\sigma_2\downarrow_l \xrightarrow{\text{std}}{}^{*} \sigma_1'\downarrow_l$. By Fair Projection, we have $\exists\sigma_2', s_2'.(\sigma_2, s_2) \xrightarrow{\text{fair}}{}^{*} (\sigma_2', s_2')$ and $\sigma_2' \approx_l \sigma_1'$, as desired. □

Definition 1 (Non-interfering). *We say that a program (i.e., a non-faceted term)* t *is non-interfering when the following is the case. For all* l, I_1, I_2, σ_1, *if* $I_1 \approx_l I_2$ *and* $(t, \emptyset, \lambda i.0, I_1, \lambda o.\epsilon) \xrightarrow{std}{}^{*} \sigma_1$ *then there exists* σ_2 *such that* $(t, \emptyset, \lambda i.0, I_2, \lambda o.\epsilon) \xrightarrow{std}{}^{*} \sigma_2$ *and* $\sigma_2 \approx_l \sigma_1$. □

$$
\begin{array}{rll}
n \in \mathbb{Z} & & \\
k, l \in \mathit{Lattice} & & \\
b \in \mathit{Branch} & ::= k \mid \overline{k} & \\
pc \in \mathit{PC} & = 2^{\mathit{Branch}} & \\
V \in \mathit{FacetedValue} & ::= \mathsf{raw}\ t & \\
 & \mid \langle k\ ?\ V : V \rangle & \\
 & \mid \mathsf{bind}\ t\ t & \\
x \in \mathit{Var} & & \\
t \in \mathit{Term} & ::= x & \\
 & \mid \lambda x.t \mid t\ t & \\
 & \mid a & \\
 & \mid n \mid t + t & \\
 & \mid \mathsf{if}\ t\ t\ t & \\
 & \mid V & \\
 & \mid \mathsf{return}\ t \mid t \gg\!= t & \\
 & \mid \mathsf{new}\ t \mid \mathsf{read}\ t \mid \mathsf{write}\ t\ t & \\
 & \mid \mathsf{get}\ i \mid \mathsf{put}\ o\ t & \\
 & \mid \mathsf{run}\ t & \\
 & \mid \langle\!\langle k\ ?\ t : t \rangle\!\rangle & \\
T \in \mathit{Type} & ::= \mathsf{Int} & \\
 & \mid T \to T & \\
 & \mid \mathsf{Fac}\ T & \\
 & \mid \mathsf{FIO}\ T & \\
 & \mid \mathsf{FIORef}\ T & \\
\Gamma \in \mathit{VarTypes} & = \mathit{Var} \to \mathit{Type} &
\end{array}
$$

Figure 8: Full syntax (part I).

$$
\begin{array}{rll}
a \in \mathit{Address} & & \\
i \in \mathit{InputHandle} & & \\
o \in \mathit{OutputHandle} & & \\
l_i \in \mathit{Lattice} & \text{is the label of the channel } i & \\
l_o \in \mathit{Lattice} & \text{is the label of the channel } o & \\
v \in \mathit{Value} & ::= V & \\
 & \mid \lambda x.t & \\
 & \mid n & \\
 & \mid a & \\
 & \mid \mathsf{return}\ v & \\
E \in \mathit{Context} & ::= \bullet\ t & \\
 & \mid \mathsf{bind}\ \bullet\ t & \\
 & \mid \bullet + t \mid v + \bullet & \\
 & \mid \mathsf{if}\ \bullet\ t\ t & \\
 & \mid \bullet \gg\!= t & \\
 & \mid \mathsf{run}\ \bullet \mid \mathsf{run}\ (\mathsf{bind}\ \bullet\ t) & \\
 & \mid \mathsf{new}\ \bullet \mid \mathsf{read}\ \bullet \mid \mathsf{write}\ \bullet\ t \mid \mathsf{write}\ a\ \bullet & \\
 & \mid \mathsf{put}\ o\ \bullet & \\
 & \mid \mathsf{return}\ \bullet & \\
M \in \mathit{Memory} & = \mathit{Address} \to \mathit{FacetedValue} & \\
p \in \mathit{BufferPointer} & ::= n \mid \langle k\ ?\ p : p \rangle & \\
P \in \mathit{BufferPointers} & = \mathit{InputHandle} \to \mathit{BufferPointer} & \\
ns \in \mathit{Sequence} & = \mathbb{Z}^{*} & \\
I \in \mathit{InputBuffer} & = \mathit{InputHandle} \to \mathit{Sequence} & \\
O \in \mathit{OutputBuffer} & = \mathit{OutputHandle} \to \mathit{Sequence} & \\
\sigma \in \mathit{State} & ::= (t, M, P, I, O) & \\
\Delta \in \mathit{MemoryTypes} & = \mathit{Address} \to \mathit{Type} &
\end{array}
$$

Figure 9: Full syntax (part II).

Theorem 5 (Transparency).
If t *is non-interfering and* $\sigma = (t, \emptyset, \lambda i.0, I, \lambda o.\epsilon) \xrightarrow{std}{}^{*} \sigma'$ *then there exists* σ'', s'' *such that* $(\sigma, s) \xrightarrow{fair}{}^{*} (\sigma'', s'')$ *and* $\sigma' \approx_l \sigma''$.

Proof sketch
Since $t = t\downarrow_l$ is non-interfering, we have σ''' such that $\sigma\downarrow_l \xrightarrow{\text{std}}{}^{*} \sigma'''$ and $\sigma' \approx_l \sigma'''$. By repeated application of Fair Projection, we have σ'' and s'' such that $(\sigma, s) \xrightarrow{\text{fair}}{}^{*} (\sigma'', s'')$ and $\sigma''\downarrow_l = \sigma'''$. Finally, $\sigma'\downarrow_l = \sigma'''\downarrow_l = \sigma''\downarrow_l\downarrow_l = \sigma''\downarrow_l$, as desired. □

Theorem 6 (Emptiness check).

$$\forall pc.\ \mathit{views}(pc) \neq \emptyset \Leftrightarrow \forall \overline{k} \in pc.\ k \not\sqsubseteq l_c(pc)$$

Proof Sketch Right-to-left holds trivially by the definition of the candidate label. In the other direction we have that for any $l \in \mathit{views}(pc)$, it is the case that $l_c(pc) \sqsubseteq l$ by simple properties of the join, i.e., as $l_c(pc)$ computes the least upper bound of the positive labels in pc, and $\forall \overline{k} \in pc.\ k \not\sqsubseteq l$ by (6). We will prove the theorem by contradiction. Assume that $\neg(\forall \overline{k} \in pc.\ k \not\sqsubseteq l_c(pc))$, we then have $\exists \overline{k} \in pc.\ k \sqsubseteq l_c(pc)$. Let us take k_0 to be the witness of this existential quantification. We obtain, by transitivity of $(\sqsubseteq)$, $k_0 \sqsubseteq l_c(pc) \sqsubseteq l$, but $l \in \mathit{views}(pc)$ which implies that $k_0 \not\sqsubseteq l$, contradiction. □

B IMPLEMENTATION

DC labels, see section 7.1, are represented as a Haskell data type:

$\boxed{\sigma \longrightarrow_{pc} \sigma}$

$$(E[t], M, P, I, O) \longrightarrow_{pc} (E[t'], M', P', I', O') \quad \text{if } (t, M, P, I, O) \longrightarrow_{pc} (t', M', P', I', O') \quad \text{[F-CONTEXT]}$$

$$\begin{array}{lll}
((\lambda x.t_1)\ t_2, M, P, I, O) & \longrightarrow_{pc} (t_1[x := t_2], M, P, I, O) & \text{[F-APP]} \\
(n_1 + n_2, M, P, I, O) & \longrightarrow_{pc} (n, M, P, I, O) \quad \text{if } n = n_1 + n_2 & \text{[F-PLUS]} \\
(\text{if } n\ t_1\ t_2, M, P, I, O) & \longrightarrow_{pc} (t_1, M, P, I, O) \quad \text{if } n \neq 0 & \text{[F-IF-1]} \\
(\text{if } n\ t_1\ t_2, M, P, I, O) & \longrightarrow_{pc} (t_2, M, P, I, O) \quad \text{if } n = 0 & \text{[F-IF-2]} \\
((\text{return } t_1) \gg\!= t_2, M, P, I, O) & \longrightarrow_{pc} (t_2\ t_1, M, P, I, O) & \text{[F-BIND-FIO]} \\
(\text{run } (\text{raw } t), M, P, I, O) & \longrightarrow_{pc} (t \gg\!= \lambda x.\text{return } (\text{raw } x), M, P, I, O) & \text{[F-RUN-RAW]} \\
(\text{run } \langle k\ ?\ t_1 : t_2\rangle, M, P, I, O) & \longrightarrow_{pc} \begin{cases} (\text{run } t_1, M, P, I, O) & \text{if } \mathit{views}(pc \cup \{\overline{k}\}) = \emptyset \\ (\text{run } t_2, M, P, I, O) & \text{if } \mathit{views}(pc \cup \{k\}) = \emptyset \\ (\langle\!\langle k\ ?\ \text{run } t_1 : \text{run } t_2\rangle\!\rangle, M, P, I, O) & \text{otherwise.} \end{cases} & \begin{array}{l}\text{[F-RUN-FACET-1]} \\ \text{[F-RUN-FACET-2]} \\ \text{[F-RUN-FACET-3]}\end{array} \\
(\text{run } (\text{bind } (\text{raw } t_1)\ t_2), M, P, I, O) & \longrightarrow_{pc} (\text{run } (t_2\ t_1), M, P, I, O) & \text{[F-BIND-FAC-1]} \\
(\text{run } (\text{bind } \langle k\ ?\ V_1 : V_2\rangle\ t), M, P, I, O) & \longrightarrow_{pc} (\text{run } \langle k\ ?\ \text{bind } V_1\ t : \text{bind } V_2\ t\rangle, M, P, I, O) & \text{[F-BIND-FAC-2]} \\
(\text{run } (\text{bind } (\text{bind } t_1\ t_2)\ t_3), M, P, I, O) & \longrightarrow_{pc} (\text{run } (\text{bind } t_1\ (\lambda x.\text{bind } (t_2\ x), M, P, I, O)\ t_3)) & \text{[F-BIND-FAC-3]} \\
(E[\langle\!\langle k\ ?\ t_1 : t_2\rangle\!\rangle], M, P, I, O) & \longrightarrow_{pc} (\langle\!\langle k\ ?\ E[t_1] : E[t_2]\rangle\!\rangle, M, P, I, O) & \text{[F-FORK-CONTINUATION]} \\
(\langle\!\langle k\ ?\ \text{return } V_1 : \text{return } V_2\rangle\!\rangle, M, P, I, O) & \longrightarrow_{pc} (\text{return } \langle k\ ?\ V_1 : V_2\rangle, M, P, I, O) & \text{[F-MERGE]} \\
(\langle\!\langle k\ ?\ t_1 : t_2\rangle\!\rangle, M, P, I, O) & \longrightarrow_{pc} (\langle\!\langle k\ ?\ t_1' : t_2\rangle\!\rangle, M, P, I, O) \quad \text{if } \overline{k} \notin pc \text{ and } t_1 \longrightarrow_{pc \cup \{k\}} t_1' & \text{[F-THREAD-1]} \\
(\langle\!\langle k\ ?\ t_1 : t_2\rangle\!\rangle, M, P, I, O) & \longrightarrow_{pc} (\langle\!\langle k\ ?\ t_1 : t_2'\rangle\!\rangle, M, P, I, O) \quad \text{if } k \notin pc \text{ and } t_2 \longrightarrow_{pc \cup \{k\}} t_2' & \text{[F-THREAD-2]}
\end{array}$$

$$\begin{array}{lll}
(\text{new } V, M, P, I, O) & \longrightarrow_{pc} (\text{return } a, M[a := \langle\!\langle pc\ ?\ V : \text{raw } 0\rangle\!\rangle], P, I, O) \quad \text{if } a \notin \text{dom}(M) & \text{[F-NEW]} \\
(\text{read } a, M, P, I, O) & \longrightarrow_{pc} (\text{return } M(a), M, P, I, O) & \text{[F-READ]} \\
(\text{write } a\ V, M, P, I, O) & \longrightarrow_{pc} (\text{return } V, M', P, I, O) \quad \text{if } M' = M[a := \langle\!\langle pc\ ?\ V : M(a)\rangle\!\rangle] & \text{[F-WRITE]} \\
(\text{get } i, M, P, I, O) & \longrightarrow_{pc} (\text{return } \langle l_i\ ?\ V : \text{raw } 0\rangle, M, P[i := p'], I, O) \quad \text{if } (V, p') = \text{fac_get}(pc, P(i), I(i)) & \text{[F-GET]} \\
(\text{put } o\ n, M, P, I, O) & \longrightarrow_{pc} \begin{cases} (\text{return } n, M, P, I, O[o := O(o) +\!\!+ n]) & \text{if } l_o \in \mathit{views}(pc) \\ (\text{return } n, M, P, I, O) & \text{if } l_o \notin \mathit{views}(pc) \end{cases} & \begin{array}{l}\text{[F-PUT-1]} \\ \text{[F-PUT-2]}\end{array}
\end{array}$$

$\boxed{(V, p) = \text{fac\_get}(pc, p, ns)}$

$$\frac{(V_1, p_1') = \text{fac\_get}(pc \setminus \{k, \overline{k}\}, p_1, ns) \qquad (V_2, p_2') = \text{fac\_get}(pc \setminus \{k, \overline{k}\}, p_2, ns) \qquad k \in pc}{(\langle k\ ?\ V_1 : V_2\rangle, \langle k\ ?\ p_1' : p_2\rangle) = \text{fac\_get}(pc, \langle k\ ?\ p_1 : p_2\rangle, ns)} \quad \text{[R-FACET-1]}$$

$$\frac{(V_1, p_1') = \text{fac\_get}(pc \setminus \{k, \overline{k}\}, p_1, ns) \qquad (V_2, p_2') = \text{fac\_get}(pc \setminus \{k, \overline{k}\}, p_2, ns) \qquad \overline{k} \in pc}{(\langle k\ ?\ V_1 : V_2\rangle, \langle k\ ?\ p_1 : p_2'\rangle) = \text{fac\_get}(pc, \langle k\ ?\ p_1 : p_2\rangle, ns)} \quad \text{[R-FACET-2]}$$

$$\frac{(V_1, p_1') = \text{fac\_get}(pc \setminus \{k, \overline{k}\}, p_1, ns) \qquad (V_2, p_2') = \text{fac\_get}(pc \setminus \{k, \overline{k}\}, p_2, ns) \qquad k, \overline{k} \notin pc}{(\langle k\ ?\ V_1 : V_2\rangle, \langle k\ ?\ p_1' : p_2'\rangle) = \text{fac\_get}(pc, \langle k\ ?\ p_1 : p_2\rangle, ns)} \quad \text{[R-FACET-3]}$$

$$\frac{ns_{n_1} = n_2}{(\text{raw } n_2, \langle\!\langle pc\ ?\ n_1 + 1 : n_1\rangle\!\rangle) = \text{fac\_get}(pc, n_1, ns)} \quad \text{[R-RAW]}$$

$$\frac{n \geq \text{length}(I(i))}{(\text{raw } (-1), \langle\!\langle pc\ ?\ n + 1 : n\rangle\!\rangle) = \text{fac\_get}(pc, n, ns)} \quad \text{[R-RAW-EOF]}$$

Figure 10: Full semantics.

```
data Form = T | F | And Form Form | Or Form Form | Atomic String
data Label = Label Form Form
```

Where `Label (Atomic "a" `Or` Atomic "b") (Atomic "b")` denotes a DC label $\langle a \vee b, b\rangle$. Similarly, faceted values are represented as a Generalised Algebraic Data Type:

```
data Fac a where
  Raw  :: a -> Fac a
  Bind :: Fac a -> (a -> Fac b) -> Fac b
  Q    :: Label -> Fac a -> Fac a -> Fac a
```

Where `Q l priv pub` represents the faceted value $\langle l\ ?\ priv : pub\rangle$. We represent FIO references using Haskell's mutable `IORef` references.

```
data Ref a = Ref (IORef (Fac a))
```

Channels are represented using file handles and mutable references:

$$[\text{T-VAR}]\ \Gamma, \Delta \vdash x :: \Gamma(x) \qquad [\text{T-LAM}]\ \frac{\Gamma[x := T_1], \Delta \vdash t_2 :: T_2}{\Gamma, \Delta \vdash \lambda x.t_2 :: T_1 \to T_2} \qquad [\text{T-APP}]\ \frac{\Gamma, \Delta \vdash t_0 :: T_1 \to T_2 \qquad \Gamma, \Delta \vdash t_1 :: T_1}{\Gamma, \Delta \vdash t_0\ t_1 :: T_2}$$

$$[\text{T-ADDR}]\ \frac{}{\Gamma, \Delta \vdash a :: \Delta(a)} \qquad [\text{T-INT}]\ \frac{}{\Gamma, \Delta \vdash n :: \mathsf{Int}}$$

$$[\text{T-PLUS}]\ \frac{\Gamma, \Delta \vdash t_1 :: \mathsf{Int} \qquad \Gamma, \Delta \vdash t_2 :: \mathsf{Int}}{\Gamma, \Delta \vdash t_1 + t_2 :: \mathsf{Int}} \qquad [\text{T-IF}]\ \frac{\Gamma, \Delta \vdash t_0 :: \mathsf{Int} \qquad \Gamma, \Delta \vdash t_1 :: T \qquad \Gamma, \Delta \vdash t_2 :: T}{\Gamma, \Delta \vdash \mathsf{if}\ t_0\ t_1\ t_2 :: T}$$

$$[\text{T-RAW}]\ \frac{\Gamma, \Delta \vdash t :: T}{\Gamma, \Delta \vdash \mathsf{raw}\ t :: \mathsf{Fac}\ T}$$

$$[\text{T-FACET}]\ \frac{\Gamma, \Delta \vdash V_1 :: \mathsf{Fac}\ T \qquad \Gamma, \Delta \vdash V_2 :: \mathsf{Fac}\ T}{\Gamma, \Delta \vdash \langle k\ ?\ V_1 : V_2 \rangle :: \mathsf{Fac}\ T} \qquad [\text{T-BIND-FAC}]\ \frac{\Gamma, \Delta \vdash t_1 :: \mathsf{Fac}\ T_1 \qquad \Gamma, \Delta \vdash t_2 :: T_1 \to \mathsf{Fac}\ T_2}{\Gamma, \Delta \vdash \mathsf{bind}\ t_1\ t_2 :: \mathsf{Fac}\ T_2}$$

$$[\text{T-RETURN}]\ \frac{\Gamma, \Delta \vdash t :: T}{\Gamma, \Delta \vdash \mathsf{return}\ t :: \mathsf{FIO}\ T}$$

$$[\text{T-BIND-FIO}]\ \frac{\Gamma, \Delta \vdash t_1 :: \mathsf{FIO}\ T_1 \qquad \Gamma, \Delta \vdash t_2 :: T_1 \to \mathsf{FIO}\ T_2}{\Gamma, \Delta \vdash t_1 \gg\!= t_2 :: \mathsf{FIO}\ T_2} \qquad [\text{T-NEW}]\ \frac{\Gamma, \Delta \vdash t :: \mathsf{Fac}\ T}{\Gamma, \Delta \vdash \mathsf{new}\ t :: \mathsf{FIO}\ (\mathsf{FIORef}\ T)} \qquad [\text{T-READ}]\ \frac{\Gamma, \Delta \vdash t :: \mathsf{FIORef}\ T}{\Gamma, \Delta \vdash \mathsf{read}\ t :: \mathsf{FIO}\ (\mathsf{Fac}\ T)}$$

$$[\text{T-WRITE}]\ \frac{\Gamma, \Delta \vdash t_1 :: \mathsf{FIORef}\ T \qquad \Gamma, \Delta \vdash t_2 :: \mathsf{Fac}\ T}{\Gamma, \Delta \vdash \mathsf{write}\ t_1\ t_2 :: \mathsf{FIO}\ (\mathsf{Fac}\ T)} \qquad [\text{T-GET}]\ \frac{}{\Gamma, \Delta \vdash \mathsf{get}\ i :: \mathsf{FIO}\ (\mathsf{Fac}\ \mathsf{Int})} \qquad [\text{T-PUT}]\ \frac{\Gamma, \Delta \vdash t :: \mathsf{Int}}{\Gamma, \Delta \vdash \mathsf{put}\ o\ t :: \mathsf{FIO}\ \mathsf{Int}} \qquad [\text{T-RUN}]\ \frac{\Gamma, \Delta \vdash t :: \mathsf{Fac}\ (\mathsf{FIO}\ T)}{\Gamma, \Delta \vdash \mathsf{run}\ t :: \mathsf{FIO}\ (\mathsf{Fac}\ T)}$$

$$[\text{T-THREADS}]\ \frac{\Gamma, \Delta \vdash t_1 :: \mathsf{FIO}\ T \qquad \Gamma, \Delta \vdash t_2 :: \mathsf{FIO}\ T}{\Gamma, \Delta \vdash \langle\!\langle k\ ?\ t_1 : t_2 \rangle\!\rangle :: \mathsf{FIO}\ T}$$

Figure 11: Typing rules $\boxed{\Gamma, \Delta \vdash t :: T}$

$\boxed{\sigma \xrightarrow{\text{std}} \sigma}$ Same rules: [F-CONTEXT], [F-APP], [F-PLUS], [F-IF-1], [F-IF-2], [F-BIND-FIO], [F-RUN-RAW], [F-BIND-FAC-1], [F-BIND-FAC-2], [F-READ]

$$(\text{new } F, M, P, I, O) \xrightarrow{\text{std}} (\text{return } a, M[a := F], P, I, O) \quad \text{if } a \notin \text{dom}(M) \quad [\text{S-NEW}]$$

$$(\text{write } a\ F, M, P, I, O) \xrightarrow{\text{std}} (\text{return } F, M[a := F], P, I, O) \quad [\text{S-WRITE}]$$

$$(\text{get } i, M, P, I, O) \xrightarrow{\text{std}} (\text{return } (\text{raw } n), M, P[i := P(i) + 1], I, O) \quad \text{if } n = I(i)_{P(i)} \quad [\text{S-GET}]$$

$$(\text{get } i, M, P, I, O) \xrightarrow{\text{std}} (\text{return } (\text{raw } (-1)), M, P, I, O) \quad \text{if } P(i) \geq \text{length}(I(i)) \quad [\text{S-GET-EOF}]$$

$$(\text{put } o\ n, M, P, I, O) \xrightarrow{\text{std}} (\text{return } n, M, P, I, O[o := O(o) +\!\!+ n]) \quad [\text{S-PUT}]$$

Figure 12: Full standard semantics.

```
data Ch = Ch { label :: Label, iH :: Handle
             , iPtr :: IORef (Fac Int), oH :: Handle }
```

FIO computations are represented as a deep embedding in a continuation-passing style. Representing the computation as a concrete data type allows us to implement multiple different executors for the same syntax.

```
data FIO a where
  RunBind :: Fac (FIO a) -> (Fac a -> FIO b) -> FIO b
  New     :: Fac a -> (Ref a -> FIO b) -> FIO b
  Read    :: Ref a -> (Fac a -> FIO b) -> FIO b
  Write   :: Ref a -> Fac a -> (() -> FIO b) -> FIO b
  Get     :: Ch -> (Fac Char -> FIO b) -> FIO b
  Put     :: Ch -> Char -> (() -> FIO a) -> FIO a
  Return  :: a -> FIO a
```

We proceed to implement the interface for side-effectful operations based on `FIO` constructors as follows:

```
newFIORef :: Fac a -> FIO (Ref a)
newFIORef f = New f Return

readFIORef :: Ref a -> FIO (Fac a)
readFIORef r = Read r Return
```

The other operations are implemented analogously.

Note that the primitives `Read`, `New` and `Write` support continuations, as motivated in Section 8.4. Based on these continuation-based primitives, we implement non-continuation-based wrappers that have the expected type matching Figure 11.

The `return` and `(>>=)` constructs are implemented as derived operations (they are usually provided as parts of the standard `Monad` interface) [27, 55].

```
(>>=) :: FIO a -> (a -> FIO b) -> FIO b
Return a    >>= k = k a
RunBind f c >>= k = RunBind f (\a -> c a >>= k)
New f c     >>= k = New f (\a -> c a >>= k)
Read r c    >>= k = Read r (\a -> c a >>= k)
...
```

The program counter (PC) is implemented as a list of branches.

```
data Branch = Private Label | Public Label
type PC = [Branch]
```

The decision procedure from section 7 is implemented as pure Haskell function making use a library for BDDs:

```
isEmptyViews :: PC -> Bool
isEmptyViews pc =
  let lc = foldr lub (Label T F) [ k | Private k <- pc ]
  in not (and [ canFlowTo k lc | Public k <- pc ])
```

We have implemented two different executors for FIO, `mf`, `sme`. All the executors have the same type, `FIO a -> PC -> IO (a, PC)`, a function from an FIO computation and a program counter to a result and a new program counter in the `IO` monad. The definition of `mf` is straight forward:

```
mf :: FIO a -> PC -> IO (a,PC)
mf (Return a) pc = return (a, pc)
mf (New fac k) pc = do ref <- newIORef fac
                       mf (k (Ref ref)) pc
mf (Read (Ref ref) k) pc = do fac <- readIORef ref
                              mf (k fac) pc
mf (Write (Ref ref) fac k) pc = do
  atomicModifyIORef' ref $
    \old_fac -> (pcF pc fac old_fac, ())
  mf (k ()) pc
mf (Get i k) pc = do ptr <- readIORef (iPtr i)
                     (val, ptr') <- fac_get pc (iH i) ptr
                     writeIORef (iPtr i) ptr'
                     mf (k val) pc
mf (Put o v k) pc
  | label o `inViews` pc = do hPutChar (oH o) v
                              mf k pc
  | otherwise            = mf k pc

mf (RunBind (Raw fio) k) pc = do (a, pc') <- mf fio pc
                                 mf (k (Raw a)) pc
mf (RunBind (Bind (Raw fio) c) k) pc =  mf (RunBind (c fio) k) pc
mf (RunBind (Bind (Bind t0 c0) c1) k) pc =
  mf (RunBind (Bind t0 (\x -> Bind (c0 x) c1)) k) pc
mf (RunBind (Q l priv pub) k) pc
  | isEmptyViews (Public  l : pc) = mf (RunBind priv k) pc
  | isEmptyViews (Private l : pc) = mf (RunBind pub  k) pc
  | otherwise = do
      (a1,_) <- mf (RunBind priv return) (Private l : pc)
      (a2,_) <- mf (RunBind pub  return) (Public  l : pc)
      mf (k (Q l a1 a2)) pc
```

The function `pcF` used in the case for `Write` implements the notation $\langle\!\langle pc\,?\,priv : pub \rangle\!\rangle$ from Section 3.

In the case for `Return` we just return the value and the current PC. For `New` we create a new `IORef` and run the continuation `k` with that `IORef` wrapped in a `Ref` constructor. Similarly for `Read`, read the value of the reference and run the continuation. The case for `Write` is more interesting, when we are a value to a reference we need to update the current faceted value to reflect that the update is done with the current PC. Writes are executed atomically; while this is not important for the defintion of `mf` (which is sequential), it matters in concurrent executors like `sme` below. The two cases for `Run` depend on the faceted value being branched over. If the value is a leaf (`Raw fio`), we execute the FIO computation at the leaf and continue with the continutation. If the value is a branch (`Q l priv pub`), we check the branching conditions described in Section 3 and execute one of three cases. The first two cases simply pick the private or public branches depending on if the specific branching condition is satisfied. The third case is more interesting, we run both the public and the private branches with different PCs, each containing either `Private l` or `Public l`. Note that this is a literal translation of the

The definition of `sme` is identical except for the clause for `Get`, where we use a lock to ensure that the file pointers are not concurrently updated, and the final clause of the definition for `Run`:

```
sme (RunBind (Q l priv pub) k) pc
  ...
  | otherwise = do
      forkIO . void $ sme (RunBind priv k) (Private l : pc)
      sme (RunBind pub k) (Public l : pc)
```

Instead of first running the private branch and then the public, we fork the private branch to run in parallel and continue with the public branch. Note that the use of `forkIO . void` is a technicality, the type of `forkIO` requires a computation of type `IO ()` as argumentand `void` as type `IO a -> IO ()`.

C FSME (SWITCHING) EXECUTOR

The rule [F-FORK-CONTINUATION] in the semantics models switching from a single thread of execution to multiple threads. In this appendix we show how the rule can be implemented in a switching executor. The only difference between the executor we develop here and the `sme` and `mf` variants are in the implementation of the case for `RunBind` which needs to run both the private and the public computations. The idea of this executor is to run the private computation assuming it is going to terminate. If the private computation does not terminate we start running the public computation in parallel with the private and continue by doing SME. The way this is achieved by our executor, which can be seen below, is by executing the the private computation in a separate, lightweight, thread. The thread running the private computation communicates the result of the computation to the main thread when finished. It then waits for the main thread to tell it to either terminate or continue running the continuation. The main thread waits for the result of the private computation for a bounded amount of time. If the main thread receives the result of the computation on time, then it continues running in the fashion of MF. If the main thread does not receive the result on time, then it signals the thread running the private computation to run its continuation, and the execution continues in the fashion of SME.

The necessary communication is achieved using the `MVar` data structure. A value of type `MVar a` [42] is a concurrent datastructure which is either empty or contains a term of type `a`. An empty `MVar` is created using `newEmptyMVar :: IO (MVar a)`. The function `readMVar` empties a full `MVar` and returns its content or blocks otherwise. The function `putMVar :: a -> MVar a -> IO ()` fills an empty `MVar` or blocks otherwise.

```
fsme (RunBind (Q k priv pub) f) pc =
  ...
  | otherwise = do
      privResult <- newEmptyMVar
      privCont   <- newEmptyMVar
      fork $ do -- Private facet behavior
        (priv', pc') <- fsme (RunBind priv Return) (Private k : pc)
        putMVar privResult  priv'
        -- Wait for what to do next
        switchSME <- readMVar privCont
        when switchSME $ void (fsme (k priv') pc')
      -- Public facet behavior
      onTime <- timeout waitTime (readMVar privResult)
      case onTime of
         Just priv' -> do -- No need to switch to SME
           putMVar switchSME False
           fsme (RunBind publ (\publ' -> f (Q p priv' publ')))
                (Public p : pc)
         Nothing -> do -- Switching to SME
           putMVar switchSME True
           fsme (RunBind publ f) (Public p : pc)
```