Title
Faceted Information Flow

Author
Schmitz, Thomas James

Publication Date
2019

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

FACETED INFORMATION FLOW

A dissertation submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Thomas Schmitz

June 2019

The Dissertation of Thomas Schmitz
is approved:

Cormac Flanagan, Chair

Luca de Alfaro

Owen Arden

Alejandro Russo

Lori Kletzer
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Faceted Information Flow

by

Thomas Schmitz

This thesis aims to make progress on the problem of using dynamic information flow control for computer security at the application level, specifically using Faceted Values. This technique involves augmenting program data values so that each one is a pair of two primitive values: one high-security version that is visible only to high-security observers, and one low-security version that is visible to everyone else. These augmented values are called faceted values, and the various versions are called facets. This technique allows very precise tracking of information flow through a program, allowing programmers to increase confidence in the security of their systems.

This thesis helps to increase the maturity of research on the "Faceted Values" technique, bringing it in line with research on the prior techniques "No Sensitive Upgrades" and "Secure Multi Execution." Specifically, we have formalized a new semantics (called Multef) and proved that it satisfies a strong ("termination sensitive") security property, we have implemented the technique as a Haskell library (called FIO) using two monads, and we have tested it in a prototype social network application (called FacetBook).

Chapter 1

Introduction

To motivate this thesis, we should look at all the buggy software out there with unmet security requirements. Consider social media (blog, forum, Facebook, etc.), banking software, search engines (issues with privacy), web mashups (webpages with advertisements, and generally any software that combines functionality of multiple services), and various kinds of shared database (Google Drive, conference management systems, etc.). Such software is often developed without any systematic methodology for guaranteeing security. This application level security contrasts with lower level system security (e.g. in operating systems and in hardware) because it is practical to sacrifice some runtime performance for improved confidence in security.

The word "security" means different things to different people. Here, we focus specifically on *information flow* security. Information flow *policies* [10, 4, 7] specify that a particular *restricted class of data* shall not flow to particular *restricted output channels*. Many software security requirements can be phrased in this way, and so we consider how such policies may be enforced.

1.1 Background

We begin with a brief review of some prior work on information flow control.

1.1.1 Security lattices

It is common to use *lattices* to specify information flow policies [4]. The security lattice is a set of *labels*, and we denote the lattice's partial order using the symbol $\sqsubseteq$. Each label in the lattice represents an information class. Information from one class may flow to another as long as the direction of flow is *upward* through the lattice; all other flows are prohibited.

1.1.2 Dynamic information flow control

After writing a policy (using a lattice), we would like to enforce the policy when running code. Most enforcement mechanisms are classified either as static or as dynamic. Static mechanisms (such as Jif [9]) analyze the program itself; dynamic mechanisms (such as LIO [14]) analyze the execution of the program.

Because they perform the analysis at runtime, dynamic mechanisms expose information flow violations later and exhibit worse runtime performance than static mechanisms do. On the other hand, dynamic techniques can offer better precision by exploiting observations about the program's runtime behavior. We focus on dynamic techniques for application level security, where the advantage of precision outweighs the disadvantage of performance.

Compared to other dynamic analysis techniques, dynamic information flow control is unusual because a single execution cannot constitute an information flow violation; rather, to exhibit a violation, we must compare at least two executions to one another. Therefore, dynamic techniques either enforce a conservative approximation of the desired policy or execute (parts of) the program multiple times.

1.1.3 No Sensitive Upgrades

No Sensitive Upgrades (NSU) [1, 14, 15] is a technique that involves labeling data values during program execution. Every value is either labeled H for restricted "high security" data or labeled L for unrestricted "low security" data. The labels allow tracking which values contain information about the restricted data. (For simplicity of exposition, we assume that the security lattice is $\{L, H\}$ with $L \sqsubset H$.) Dually, every output channel is either labeled L for restricted "low security" output channels or labeled H for unrestricted "high security" output channels.

During computation, the mechanism propagates labels from input values to output values. For example:

```
1    var x = 12345;   // A secret number
2    var y = x % 2;   // Get a bit of info about x
3    print(y);        // A public output channel
```

Given that the output channel `print` is labeled L, the above code fails on line 3 because the label H propagates from `x` to `y`, and `print` cannot accept arguments labeled H. This type of information propagation is an *explicit flow*.

Information can also propagate via the conditional presence or absence of side effects, such as assigning to variables, throwing exceptions, or printing to the console; we call these *implicit flows*. To track implicit flows, the NSU mechanism keeps a global *program counter label* (PC), which indicates whether the restricted data has influenced the current control flow path. For example:

```
var x = 12345;         // A secret number
```

```
2    if(x % 2 == 1) {       // PC label becomes H
3      print("I'm odd!");   // A public output channel
4    }                      // PC label becomes L
```

Program constants (such as the string `"I'm odd!"`) acquire their labels from the program counter, so in the above code, the label H propagates from `x` to the program counter on line 2, and then from the program counter to the constant string `"I'm odd!"` on line 3. Thus, the program fails on line 3, much like before. Note that this program would not fail if the secret number were even instead of odd; in the present discussion, we consider this behavior to be acceptably secure, although Section 1.2.2 describes enforcement mechanisms where attackers cannot infer information from mechanism failures.

Some implicit flows are harder to catch. We must analyze how information may be deduced when a side effect is *not* executed. It is infeasible to consider all skipped execution paths, but we can detect cases where the current execution path leaks information to other paths; in particular, we detect when a side effect *upgrades* the label of a value during a *sensitive* execution context (i.e., when the current program counter label is H). This example (adapted from [6]) illustrates the necessity of this check:

```
var x = 12345;     // A secret number
var y = 0;
var z = 1;
if(x % 2 == 1) {   // PC label becomes H
  y = 1;           // Sensitive upgrade occurs here
}
if(y == 0) {
  z = 0;
```

```
9    }
10   print(z);
```

Without the NSU mechanism, the above program would (indirectly) write the least significant bit of the secret number into the variable `z` and subsequently print it. Explicitly, if the value of `x` were `0`, then the program would print `0`; if the value of `x` were `1`, then the program would print `1`.

However, the NSU mechanism detects the sensitive upgrade on line `5`, and the program fails. Again, note that the program would not fail if the secret number were even instead of odd.

There are multiple ways to implement the mechanism to fail after detecting a sensitive upgrade. The classic choice is to diverge (i.e., to go into an infinite loop), thus preventing any use of leaked information. Another option (proposed by [6]) is to suppress the side effect (i.e., updating the value of `y`) and continue execution, though this may yield unexpected results.

This enforcement mechanism yields false positives: not all programs with sensitive upgrades are actually insecure. For example:

```
1    var x = 12345;       // A secret number
2    var y;
3    if(x % 2 == 1) {     // PC label becomes H
4      y = "I'm odd!";    // Sensitive upgrade occurs here
5    } else {
6      y = "I'm even!";
7    }
```

The above program is secure because it prints nothing to the public output channel, but the mechanism fails on line `4` due to the sensitive upgrade.

Due to such false positives, NSU lacks a desirable property called *precision*; we say that an enforcement mechanism is *precise* if it does not alter the behavior of any *secure* programs, which are programs that already satisfy the desired policy.

1.1.4 Secure Multi Execution

The Secure Multi Execution (SME) mechanism [5] runs the program twice:

- The *high execution* runs in a sandbox where it is *legal* to read the restricted (H) data but *illegal* to write to the restricted (L) output channel.
- The *low execution* runs in a sandbox where it is *illegal* to read the restricted (H) data but *legal* to write to the restricted (L) output channel.

This technique is clearly secure because it is explicitly impossible for the restricted (H) data to flow to the restricted (L) output channel.

SME is also precise—it does not alter the behavior of any secure programs. In particular, the low execution produces the correct output on the restricted channel; the high execution produces correct output on all other output channels. This is in contrast with NSU, which fails on some secure programs.

SME can easily support internal program effects (e.g., assigning to variables and throwing exceptions) because the effects are local to one execution and do not need to propagate to the second execution. On the other hand, externally visible effects (e.g., printing to the console) are clearly duplicated. To cope with this, we must partition outside observers into low security observers and high security observers, and we must arrange the execution environment so that the low security observers see only the effects of the first execution, while the high security observers see only those of the second.

Another problem is that the performance overhead can be quite large. Executing the program twice takes about twice as much time. When using SME to enforce multiple information flow policies on a single program, the runtime overhead is exponential in the number of policies. However, we expect that much of this computation is redundant, which leads us to the next technique.

1.1.5 Faceted Values

The Faceted Values (FV) mechanism augments the program data values so that each one is a pair of two primitive values: one high security version that is visible only to high security observers, and one low security version that is visible to everyone else. These augmented values are called *faceted* values, and the various versions are called *facets*. If the two facets of a value are identical, then we can optimize the representation by collapsing the pair to a single primitive value.

When a faceted value is used during program execution, then the execution must *bifurcate* into two separate executions, one for each facet. When the sub-executions complete, the mechanism combines their results into a new faceted value and continues the remainder of the program as a single execution.

To properly handle side effects, we need a *program counter data structure* (typically called *pc*), which tracks whether the execution has bifurcated, and if so, which of the two sub-executions is currently running. By tracking this information, the mechanism can correctly decide whether to perform the effect for low observers or for high observers. If the execution has not bifurcated, then both effects occur.

FV includes special support for mutable reference cells. Rather than maintaining two stores separately, the mechanism puts faceted values into the unique global store. When updating a reference cell, the *pc* dictates which one (or both) of the two facets should change.

Overall, this technique resembles SME because parts of the program execute twice. However, the performance characteristics differ because some parts execute only once.

FV also resembles NSU. If we use a special “undefined” token for the public facet of every faceted value, then the two mechanisms behave analogously until a sensitive upgrade occurs. At this point, FV continues execution by updating the public facets as necessary (thus deviating from the public-facets-must-be-“undefined” discipline), whereas NSU conservatively aborts the program.

1.1.6 Richer lattices

As described so far, each of the three techniques supports just a two-element security lattice. It is easy to generalize NSU and SME to support an arbitrary security lattice with n elements: for NSU, we can use n different labels instead of just two; for SME, we can execute the program n times instead of just twice.

It is known [2] that FV generalizes easily to support power set lattices, simply by orthogonally composing multiple copies of the two-element lattice mechanism. We have shown [11] that the technique can also support arbitrary lattices. This result is intuitive because FV has the same semantics as SME, which itself supports arbitrary lattices.

When using FV with a two-element lattice, there are three legal values for the pc:

- $pc = \mathsf{HL}$, which means that the execution has not bifurcated and thus we are currently simulating both views at the same time;
- $pc = \mathsf{H}$, which means that we are currently simulating only the high security view; and
- $pc = \mathsf{L}$, which means that we are currently simulating only the low security view.

The salient information contained in *pc* is the set of views currently being simulated. There is one view for each lattice element, so the generalized *pc* data structure should denote a set of views (lattice elements) containing the ones currently being simulated.

Rather than representing a faceted value as a pair of primitive values (one for each of the two views), we can instead represent a faceted value as a function that maps lattice elements to primitive values (so again we have one for each view). In the semantics (and in the Haskell prototype implementation), we represent these functions as binary decision trees with lattice elements at the nodes and primitive values at the leaves.

1.2 Structure of the dissertation

This thesis is organized into three self-contained chapters. Each chapter focuses exclusively on one topic, and they can be read in any order.

1.2.1 Overview of Chapter 2: Faceted Dynamic Information Flow via Control and Data Monads

We have implemented FV as a Haskell library called `FIO` [12]. The library design includes two monads: one (called `FIO`) for encapsulating side effects (as is typical in Haskell), and surprisingly a second one (called `Faceted`) for encapsulating the faceted values. `FIO` resembles Haskell's built-in `IO` monad, but offers only a subset of the functionality—namely, mutable reference cells and file I/O, for which we have designed suitably secure algorithms with proofs of noninterference [2]. `Fac` forms a monad because faceted values support the three necessary operations:

- `return :: a -> Faceted a` creates a faceted value where all viewers see the same facet;

- `fmap :: (a -> b) -> Faceted a -> Faceted b` changes the facets of a faceted value by applying a function uniformly to each facet, preserving the required property that information from one facet cannot influence another facet; and

- `join :: Faceted (Faceted a) -> Faceted a` reinterprets a faceted value with faceted values in its facets by aggregating all of the facets of the latter faceted values into a single faceted value, preserving the labels protecting each facet.

To enable interactions between the two monads, the library provides a *prod* [8] function:

```
prod :: Faceted (FIO (Faceted a)) -> FIO (Faceted a)
```

This function enables the execution of computations that depend on faceted information—in other words, faceted computations. The resulting execution bifurcates if necessary when running those computations.

1.2.2 Overview of Chapter 3: Faceted Secure Multi Execution

Each mechanism mentioned so far satisfies a formal correctness criterion called *termination insensitive noninterference* (TINI). This criterion states:

- If we execute a program with two different but *indistinguishable* inputs and thusly obtain two outputs, then the two outputs should also be indistinguishable.

Here, we say that two values are indistinguishable when their *censored* versions are equal. The censored version of a value is obtained just by replacing its restricted data with non-informative default data.

The above criterion is called *termination insensitive* because divergent program executions are exempted from consideration, as they do not produce outputs. Each mechanism described so far [1, 5, 2] guarantees TINI.

Alternatively, if we rephrase the criterion so that divergence is considered a possible program output, then the new criterion is called *termination sensitive* noninterference (TSNI). Explicitly, TSNI specifies that if the program diverges on one input, then it should also diverge on the other (indistinguishable) input:

- If we execute a program with two different but indistinguishable inputs, then the two resulting behaviors (either convergence to a specific value or divergence) should also be indistinguishable.

TSNI is strictly stronger than TINI, which means that fewer programs satisfy the TSNI criterion.

Previous work [13, 5] on both NSU and SME has adapted them to offer TSNI. This work aims to adapt FV likewise.

The extension of NSU guarantees TSNI, but at the cost of a new programming model involving concurrency. Many existing programs do not work as written because they are not written as concurrent programs.

SME can also guarantee TSNI, namely by running the multiple executions concurrently.

To extend FV to offer TSNI, we developed a new technique called Faceted Secure Multi Execution (FSME), which runs the two parts concurrently when the mechanism bifurcates. However, it becomes tricky to *join* the subcomputations, which means waiting for both subcomputations to complete before executing their shared continuation. Joining may be unsafe because then the subsequent low continuation would depend on the termination of the current high subcomputation.

We propose that when a subcomputation completes, it should wait at most T seconds for the other subcomputation to complete, where T is a configurable parameter of the system. If the other subcomputation completes before T seconds have elapsed, then they join and the continuation will execute as normal; otherwise, if the T seconds expire, then the two subcomputations will not join at all: instead, each thread will execute the continuation when ready to do so. In the latter case, the continuation will execute a total of two times instead of just one time.

When using $T = 0$, the mechanism is identical to SME (modulo lazy spawning) because the system will eventually have spawned one thread for each lattice element, and none of these threads will ever join together (we call this variation *demand-driven* SME). On the other hand, when $T = \infty$, the mechanism is identical to (TINI) FV because every bifurcation will join before executing the continuation.

The resulting system (with positive finite T) satisfies the TSNI criterion and enjoys most of the performance advantages of FV. We have produced a Haskell library and a formal semantics with a proof of TSNI. To validate its usefulness, we have developed ProtectedBox, a secure file hosting API that supports third party plugins, which add functionality without compromising security. We developed three plugins for ProtectedBox:

- The *comments* plugin allows users to add comments to the files in the cloud.
- The *tarball* plugin allows users to create archive files that aggregate the contents of multiple other files.
- The *checksum* plugin computes a checksum for each file in the cloud.

In this experimental application using these plugins, we verified that FSME does not noticeably degrade performance.

1.2.3 Overview of Chapter 4: FacetBook

To validate the usefulness of FV in general and of our Haskell library `FIO` in particular, we have created a prototype social networking website called FacetBook.

Social networking websites inherently involve interactions among many people, so the information flow security requirements are complex. Since FV can handle complex requirements, a social networking application is a good choice to illustrate its usefulness. Recent research (and recent events) point to the importance both of social media itself [3] and of social media security particularly [16], so it interested us to investigate how FV can improve the state of the art.

We created two implementations of FacetBook: one ordinary Haskell implementation and a second Haskell implementation using the `FIO` library. The first implementation has fewer lines of code in total, but the second implementation has fewer lines of code in the *trusted computing base* (TCB), which is the part of the code that must be carefully examined in order to convince oneself that the code meets the security requirements. Smaller TCBs are easier to audit, and so by building an application with a reduced TCB, we illustrate that FV helps to improve confidence in security.

1.3 Future directions

With the theoretical underpinnings lain out in this thesis, future work can focus on the practical issues of implementing specific applications. In the worst case, the number of bifurcations during faceted execution equals the size of the security lattice (or is unbounded in the case of infinite lattices), so we anticipate optimization techniques for limiting the necessary number of bifurcations. In other work [11], we present some theoretical optimization ideas where at most one bifurcation occurs at each conditional control structure. Additional engineering effort can reduce the performance overhead of managing large numbers of bifurcations.

Debugging tools specific to faceted execution would help programmers understand when and why their code bifurcates. This understanding will help auditors in verifying the correctness of the security policy and will help programmers to optimize their code by manually reducing bifurcations.

Bibliography

[1] Thomas H Austin and Cormac Flanagan. "Efficient purely-dynamic information flow analysis". In: *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)* (2009).

[2] Thomas H Austin and Cormac Flanagan. "Multiple facets for dynamic information flow". In: *Proc. ACM Symp. on Principles of Programming Languages (POPL)* (2012).

[3] Shelley Boulianne. "Social media use and participation: A meta-analysis of current research". In: *Information, Communication & Society* 18.5 (2015), pp. 524–538.

[4] Dorothy E Denning. "A lattice model of secure information flow". In: *Communications of the ACM* 19.5 (1976), pp. 236–243.

[5] Dominique Devriese and Frank Piessens. "Noninterference through secure multi-execution". In: *Security and Privacy (SP), 2010 IEEE Symposium on.* IEEE. 2010, pp. 109–124.

[6] Jeffrey Stewart Fenton. "Memoryless subsystems". In: *The Computer Journal* 17.2 (1974), pp. 143–147.

[7] Daniel Hedin and Andrei Sabelfeld. *A Perspective on Information-Flow Control.* 2012.

[8] Mark P Jones and Luc Duponcheel. *Composing monads.* Tech. rep. Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale University, 1993.

[9] Andrew C Myers. "JFlow: Practical mostly-static information flow control". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM. 1999, pp. 228–241.

[10] Andrew C Myers and Barbara Liskov. "Protecting privacy using the decentralized label model". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9.4 (2000), pp. 410–442.

[11] Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. “A Better Facet of Dynamic Information Flow Control”. In: *WWW’18 Companion: The 2018 Web Conference Companion.* 2018, pp. 1–9.

[12] Thomas Schmitz, Dustin Rhodes, Thomas H Austin, Kenneth Knowles, and Cormac Flanagan. “Faceted dynamic information flow via control and data monads”. In: *International Conference on Principles of Security and Trust.* Springer. 2016, pp. 3–23.

[13] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C Mitchell, and David Mazieres. “Addressing covert termination and timing channels in concurrent information flow systems”. In: *The 17th ACM SIGPLAN International Conference on Functional Programming (ICFP).* 2012.

[14] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. *Flexible dynamic information flow control in Haskell.* Vol. 46. 12. ACM, 2011.

[15] Stephan Arthur Zdancewic. “Programming languages for information security”. PhD thesis. Cornell University, 2002.

[16] Zhiyong Zhang and Brij B Gupta. “Social media security and trustworthiness: overview and new direction”. In: *Future Generation Computer Systems* (2016).

Chapter 2

Faceted Dynamic Information Flow via Control and Data Monads

Abstract An application that fails to ensure information flow security may leak sensitive data such as passwords, credit card numbers, or medical records. News stories of such failures abound. Austin and Flanagan[2] introduce faceted values – values that present different behavior according to the privilege of the observer – as a dynamic approach to enforce information flow policies for an untyped, imperative λ-calculus.

We implement faceted values as a Haskell library, elucidating their relationship to types and monadic imperative programming. In contrast to previous work, our approach does not require modification to the language runtime. In addition to pure faceted values, our library supports faceted mutable reference cells and secure facet-aware socket-like communication. This library guarantees information flow security, independent of any vulnerabilities or bugs in application code. The

library uses a *control* monad in the traditional way for encapsulating effects, but it also uniquely uses a second *data* monad to structure faceted values. To illustrate a non-trivial use of the library, we present a bi-monadic interpreter for a small language that illustrates the interplay of the control and data monads.

2.1 Introduction

When writing a program that manipulates sensitive data, the programmer must prevent misuse of that data, intentional or accidental. For example, when one enters a password on a web form, the password should be communicated to the site, but not written to disk. Unfortunately, enforcing these kinds of *information flow* policies is problematic. Developers primarily focus on correct functionality; security properties are prioritized only after an attempted exploit.

Just as memory-safe languages relieve developers from reasoning about memory management (and the host of bugs resulting from its *mis*management), information flow analysis enforces security properties in a systemic fashion. Information flow controls require a developer to mark sensitive information, but otherwise automatically prevent any "leaks" of this data. Formally, we call this property *noninterference*; that is, public outputs do not depend on private inputs[1].

Secure multi-execution [9, 16, 23] is a relatively recent and popular information flow enforcement technique. A program execution is split into two versions: the "high" execution has access to sensitive information, but may only write to private channels; the "low" execution may write to public channels, but cannot access any sensitive information. This elegant approach ensures noninterference.

[1] We refer to sensitive values as "private" and non-sensitive values as "public", as confidentiality is generally given more attention in the literature on information flow analysis. However, the same mechanism can also enforce integrity properties, such as that trusted outputs are not influenced by untrusted inputs.

Faceted evaluation is a technique for simulating secure multi-execution with a single process, using special *faceted values* that contain both a public view and a private view of the data. With this approach, a single execution can provide many of the same guarantees that secure multi-execution provides, while achieving better performance.

This paper extends the ideas of faceted values from an untyped variant of the λ-calculus [2] to Haskell and describes the implementation of faceted values as a Haskell library. This approach provides a number of benefits and insights.

First, whereas prior work on faceted values required the development of a new language semantics, we show how to incorporate faceted values within an existing language via library support.

Second, faceted values fit surprisingly well (but with some subtleties) into Haskell's monadic structure. As might be expected, we use an `IO`-like monad called `FIO` to support imperative updates and I/O operations. We also use a second type constructor `Faceted` to describe faceted values; for example, the faceted value $\langle k \ ? \ 3 : 4 \rangle$ has type `Faceted Int`. Somewhat surprisingly, `Faceted` turns out to also be a monad, with natural definitions of the corresponding operations that satisfy the monad axioms [33]. These two monads, `FIO` and `Faceted`, naturally interoperate via an associated product function [17] that supports switching from the `FIO` monad to the `Faceted` monad when necessary (as described in more detail below).

This library guarantees the traditional information flow security property of termination-insensitive noninterference, independent of any bugs, vulnerabilities, or malicious code in the client application.

Finally we present an application of this library in the form of an interpreter for the imperative λ-calculus with I/O. This interpreter validates the expressiveness of the `Faceted` library; it also illustrates how the `FIO` and `Faceted` monads flow along control paths and data paths respectively.

In summary, this paper contributes the following:

- We present the first formulation of faceted values and computations in a typed context.
- We show how to integrate faceted values into a language as a library, rather than by modifying the runtime environment.
- We clarify the relationship between explicit flows in pure calculations (via the `Faceted` monad) and implicit flows in impure computations (via the `FIO` monad).
- Finally, we present an interpreter for an imperative λ-calculus with dynamic information flow. The security of the implementation is guaranteed by our library. Notably, this interpreter uses the impure monad (`FIO`) in the traditional way to structure computational effects, and uses the pure faceted monad (`Faceted`) to structure values.

2.2 Review of Information Flow and Faceted Values

In traditional information flow systems, information is tagged with a label to mark it as confidential to particular parties. For instance, if we need to restrict `pin` to *bank*, we might write:

$\texttt{pin = 4321}^{bank}$

	$\texttt{x} = \langle k \ ? \ \texttt{True} : \bot \rangle$			
`do`	*Naive*	*NSU*	*Fenton*	*Faceted Evaluation*
`y <- newIORef True`	$\texttt{y} = \texttt{True}$	$\texttt{y} = \texttt{True}$	$\texttt{y} = \texttt{True}$	$\texttt{y} = \texttt{True}$
`z <- newIORef True`	$\texttt{z} = \texttt{True}$	$\texttt{z} = \texttt{True}$	$\texttt{z} = \texttt{True}$	$\texttt{z} = \texttt{True}$
`vx <- readIORef x`	—	—	—	—
`when vx`	$pc = \{k\}$	$pc = \{k\}$	$pc = \{k\}$	$pc = \{k\}$
`  (writeIORef y False)`	$\texttt{y} = \langle k \ ? \ \texttt{False} : \bot \rangle$	*stuck*	*ignored*	$\texttt{y} = \langle k \ ? \ \texttt{False} : \texttt{True} \rangle$
`vy <- readIORef y`	—		—	—
`when vy`	—		—	$pc = \{\bar{k}\}$
`  (writeIORef z False)`	—		—	$\texttt{z} = \langle k \ ? \ \texttt{True} : \texttt{False} \rangle$
`readIORef z`	—		—	—
Result:	`True`	*stuck*	`False`	$\langle k \ ? \ \texttt{True} : \texttt{False} \rangle$

Figure 2.1: A computation with implicit flows.

To protect this value, we must prevent unauthorized viewers from observing it, directly or indirectly. In particular, we must defend against *explicit flows* where a confidential value is directly assigned to a public variable, and *implicit flows* where an observer may deduce a confidential value by reasoning about the program's control flow. The following code shows an explicit flow from `pin` to the variable `x`.

```
pin = 4321^bank
x = pin + 1
```

Taint tracking – in languages such as Perl and Ruby – suffices to track straightforward explicit flows; in contrast, implicit flows are more subtle. Continuing our example, consider the following code, which uses a mutable `IORef`.

```
do above2K ← newIORef False
   if (pin > 2000)
     then writeIORef above2K True
     else return ()
```

This code illustrates a simple implicit flow. After it runs, the value of `above2K` will reflect information about `pin`, even though the code never directly assigns the value of `pin` to `above2K`. There are several proposed strategies for handling these types of flows:

1. Allow the update, but mark `above2K` as sensitive because it was changed in a sensitive context. This strategy can help for auditing information flows "in the wild" [15], but it fails to guarantee noninterference, as shown in the *Naive* column of Figure 2.1 (note that the naive computation results in `True` when `x` is `True`).

2. Disallow the update to `above2K` within the context of the sensitive conditional `pin`. When enforced at runtime, this technique becomes the *no-sensitive-*

upgrade strategy [35, 1] illustrated in the *NSU* column of Figure 2.1. Note that while this technique maintains noninterference, it also terminates the program prematurely.

3. Ignore the update to `above2K` in a sensitive context, an approach first used by Fenton [11]. This strategy guarantees noninterference by sacrificing correctness (the program's result may not be internally consistent). We show this strategy in the *Fenton* column of Figure 2.1.

Faceted values introduce a third aspect to sensitive data. In addition to the sensitive value and its label, the following faceted value includes a default public view of '0000'.

`pin =` $\langle bank\ ?\ 4321 : 0000\rangle$

Then, when we run the previous program with this faceted `pin`, the value of `above2K` is $\langle bank\ ?\ \texttt{True} : \texttt{False}\rangle$. The bank sees the sensitive value `True`, but an unauthorized viewer instead sees the default value `False`, giving a consistent picture to the unauthorized viewer while still protecting sensitive data.

Label-based information flow systems reason about multiple principals by joining labels together (e.g. $\texttt{3}^A + \texttt{4}^B = \texttt{7}^{AB}$). In a similar manner, faceted evaluation nests faceted values to represent multiple principals, essentially constructing a tree[2] mapping permissions to values:

$$\langle k_1\ ?\ 3 : 0\rangle + \langle k_2\ ?\ 4 : 0\rangle = \langle k_1\ ?\ \langle k_2\ ?\ 7 : 3\rangle : \langle k_2\ ?\ 4 : 0\rangle\rangle$$

Figure 2.1, adapted from Austin and Flanagan [2], demonstrates a classic

[2]Alternatively, a faceted value can be interpreted as a function mapping sets of labels to values, and the syntax above as merely a compact representation.

code snippet first introduced by Fenton [11]. The example uses two conditional statements to evade some information flow controls. When this code runs, the private value `x` leaks into the public variable `z`. We represent the input `x`, a confidential boolean value, in faceted notation as $\langle k \ ? \ \texttt{False} : \bot \rangle$ for false and $\langle k \ ? \ \texttt{True} : \bot \rangle$ for true, where $\bot$ means roughly 'undefined'. Boolean reference cells `y` and `z` are initialized to `True`; by default, they are public to maximize the permissiveness of these values.

When the input `x` is $\langle k \ ? \ \texttt{False} : \bot \rangle$, the value for `y` remains unchanged because the first `when` statement is not run. Then in the second `when` statement, `y` is still public, and thus `z` also remains public because it depends only on `y`. Since no private information is involved in the update to `z`, all information flow strategies return the public value `False` as their final result.

The case where the input `x` is $\langle k \ ? \ \texttt{True} : \bot \rangle$ is more interesting, as illustrated in Figure 2.1. Note that if the final value appears as `True` to public observers, then the private value `x` has leaked. The strategies differ in the way they handle the update to `y` in the first conditional statement. Since this update depends upon the value of `x`, we must be careful to avoid the potential implicit flow from `x` to `y`. We now compare how each approach handles this update.

In the *Naive* column of Figure 2.1, the strategy tracks the influence of `x` by applying the label k to `y`. Regardless, `y` is false during the second conditional, so `z` retains its public `True` value. Thus, under Naive information flow control, the result of this code sample is a public copy of `x`, violating noninterference.

The *No-Sensitive-Upgrade* approach instead terminates execution on this update, guaranteeing termination-insensitive noninterference, but at the cost of potentially rejecting valid programs. Stefan et al. implement this strategy in the elegant LIO library for Haskell [31]. Our work shares the motivations of LIO, but extends beyond the No-Sensitive-Upgrade strategy to support faceted values, thus enabling correct execution of more programs.

The *Fenton* strategy forbids the update to `y`, but allows execution to continue. This approach avoids abnormal termination, but it may return inaccurate results, as shown in Figure 2.1.

Faceted evaluation solves this dilemma by simulating different executions of this program, allowing it to provide accurate results and avoid rejecting valid programs. In the *Faceted Evaluation* column, we see that the update to `y` results in the creation of a new faceted value $\langle k \ ?\ \texttt{False} : \texttt{True} \rangle$. Any viewer authorized to see k-sensitive data[3] can see the real value of `y`; unauthorized viewers instead see `True`, thus hiding the value of `x`. In the second conditional assignment, the runtime updates `z` in a similar manner and produces the final result $\langle\, k\,?\,\texttt{True}\,{:}\,\texttt{False}\,\rangle$. In contexts with the k security label, this value will behave as `True`; in other contexts, it will behave as `False`. This code therefore provides noninterference, avoids abnormal termination, and provides accurate results to authorized users.

2.3 Library Overview

We implement faceted computation in Haskell as a library that enforces information flow security dynamically, using abstract data types to prevent buggy or malicious programs from circumventing dynamic protections. In contrast, the original formulation [2] added faceted values pervasively to the semantics of a dynamically-typed, imperative λ-calculus. Because of the encapsulation offered by Haskell's type system, we do not need to modify the language semantics. Our library is available at `https://github.com/haskell-facets/haskell-faceted`.

Our library is conceptually divided into the following components:

[3] That is, authorized to see data marked as sensitive to principal k.

```
type Label = String

data Faceted a

public  :: a → Faceted a
faceted :: Label → Faceted a → Faceted a → Faceted a
bottom  :: Faceted a

instance Monad Faceted
```

Figure 2.2: Interface for the pure fragment of the `Faceted` library.

- Pure faceted values of type a (represented by the type `Faceted` a).
- Imperative faceted computations (represented by the type `FIO` a), which can operate on:
  - faceted reference cells (represented by the type `FioRef` a), and
  - facet-enabled file handles / sockets (represented by the type `FHandle`).

2.3.1 Pure Faceted Values: `Faceted a`

Figure 2.2 shows the public interface for the pure fragment of our library. This fragment tracks explicit data flow information in pure computations.

Our implementation presumes that security labels are strings, though leaving the type of labels abstract is straightforward.

A value of type `Faceted` a represents multiple values, or *facets*, of type a. To maintain security, the facets should not be directly observable; therefore, the data type is abstract.

The function `public` injects any type a into the type `Faceted` a. It accepts a value v of type a and returns a faceted value that behaves just like v for any observer.

The function `faceted` constructs a value of type `Faceted` a from a label k and two other faceted values *priv* and *pub*, each of type `Faceted` a. To any viewer authorized to see k, the result behaves as *priv*; to all other observers, the result behaves as *pub* (and so on, recursively).

The value `bottom` (abbreviated $\perp$) is a member of `Faceted` a for any a, and represents a lack of a value. `bottom` is used when a default value is necessary, such as in a public facet. Any computation based on `bottom` results in `bottom`.

From `faceted`, we can define various derived constructors for creating faceted values with minimal effort. For example:

```
makePrivate :: Label → a → Faceted a
makePrivate k v = faceted k (public v) bottom


makeFacets :: Label → a → a → Faceted a
makeFacets k priv pub = faceted k (public priv) (public pub)
```

The `Monad` instance for `Faceted` conveniently propagates security labels as appropriate. For example, the following code uses Haskell's `do` syntax to multiply two values of type `Faceted Int`.

```
do x ← makeFacets "k" 7 1  -- <"k" ? 7 : 1>
   y ← makeFacets "l" 6 1  -- <"l" ? 6 : 1>
   return (x ∗ y)          -- <"k" ? <"l" ? 42 : 7> : <"l" ? 6 : 1»
```

Here, x is an `Int` that is extracted from `(faceted "k" 7 1)`, either `7` or `1`. The `Faceted` monad instance automatically executes the remainder of the `do` block twice (once for each possible value of x) before collecting the various results into a faceted value. The situation is similar for `y`, so the final faceted value is a tree with four leaves.

2.3.2 Faceted Reference Cells: `FIO a` and `FioRef a`

For the pure language of Section 2.3.1, information flow analysis is straightforward because all dependencies between values are explicit; there are no *implicit flows*. An implicit flow occurs when a value is computed based on side effects that depend on private data, as in the following example, where `x` is an `IORef` with initial value `0`.

```
do if secret == 42                          -- working in IO monad
     then writeIORef x 1
     else writeIORef x 2
   readIORef x
```

The return value will be `1` if and only if `secret == 42`.

Suppose we opt to protect the confidentiality of `secret` by setting `secret` = `makePrivate` k `42`. The type of `secret` is now `Faceted Int`. Then our example can be reformulated:

```
do n ← secret                               -- working in Faceted monad
   return $ do if n == 42                   -- working in IO monad
                 then writeIORef x 1
                 else writeIORef x 2
               readIORef x
```

The outer `do` begins a computation in the `Faceted` monad, with the value `42` bound to `n`. This expression has type `Faceted (IO Int)`, so it cannot be "run" as part of a Haskell program. Thus, the pure fragment of our library described so far prevents *all* implicit flows, even those that are safe.

Guided by the types, we seek a way to convert a value of type `Faceted` (`IO` a) to a value of type `IO` (`Faceted` a). The latter could then be run to yield a value of type `Faceted` a, where the facets account for any implicit flows.

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

data FIO a

instance Monad FIO

runFIO :: FIO a → PC → IO a
prod :: Faceted (FIO (Faceted a)) → FIO (Faceted a)

data FioRef a
newFioRef   :: Faceted a → FIO (FioRef (Faceted a))
readFioRef  :: FioRef (Faceted a) → FIO (Faceted a)
writeFioRef :: FioRef (Faceted a) → Faceted a → FIO (Faceted ())
```

Figure 2.3: Interface for `FIO` and `FioRef`.

Faceted `IO` computations take place in the `FIO` monad (the name is short for "Faceted I/O"). Figure 2.3 shows the public interface for this fragment of the library. When faceted data influences control flow, the result of a computation implicitly depends on the observed facets; the implementation of `FIO` transparently tracks this information flow.

The `Monad` instance for `FIO` allows sequencing computations in the usual way, so `FIO` acts as a (limited) drop-in replacement for `IO`. If `fio1` and `fio2` each have type `FIO Int`, then the following expression also has type `FIO Int`.

```
do x ← fio1
   y ← fio2
   return (x * y)
```

The function `runFIO` converts a value of type `FIO` a to a value of type `IO` a. The side effects in this `IO` computation will respect the information flow policy.

`runFIO` takes one additional argument: an initial value for a data structure called *pc* (for "program counter label"), which is used for tracking the branching of the computation. To guarantee security, it may be necessary to execute parts of the program multiple times – once for observers who may view k-sensitive data, and again for observers who may not. During the former branch of computation, the *pc* will contain the value `Private` k; during the latter branch, it will contain `Public` k.

The *pc* argument to `runFIO` allows controlling the set of observers whose viewpoints are considered during faceted computation. The empty *pc*, denoted `[]`, will force simulation of all possible viewpoints.

A value of type `FioRef` a (short for "facet-aware `IORef`") is a mutable reference cell where initialization, reading, and writing are all `FIO` computations that operate on `Faceted` values and that account for implicit flows accordingly.

Figure 2.3 presents the public interface to `FioRef` a, which parallels that of conventional reference cells of type `IORef` a.

To write side-effecting code that depends on a faceted value, the `Faceted` and `FIO` monads must be used together. The library function `prod` enables this interaction.

Using these library functions, our running example finally looks as follows.

```
do x ← newFioRef (public 0)                    -- working in FIO monad
   prod $ do v ← secret                        -- working in Faceted monad
             return $ if v == 42
                        then writeFioRef x (public 1)
                        else writeFioRef x (public 2)
   readFioRef x
```

As hinted earlier, the inner `do` block has type `Faceted (FIO (Faceted ()))` and so cannot compose with the other actions in the outer `do` block. To rectify this, the function `prod` is enclosing the inner `do` block, converting it to type `FIO`

```
data FHandle

type View = [Label]

openFileFio :: View → FilePath → IOMode → FIO FHandle
closeFio :: FHandle → FIO ()

getCharFio :: FHandle → FIO (Faceted Char)
putCharFio :: FHandle → Char → FIO ()
```

Figure 2.4: Interface for `FHandle`.

`(Faceted ())`.

In this example, the value read from `x` will be `faceted` k `1 0`, which correctly accounts for the influence from `secret`. In section 2.4, we will explain the machinery that implements this secure behavior.

2.3.3 Faceted I/O: `FHandle`

Faceted I/O differs from reference cells in that the network and file system, which we collectively refer to as the *environment*, lie outside the purview of our programming language. The environment has no knowledge of facets and cannot be retrofitted. Additionally, there are other programs able to read from and write to the file system. We assume that the environment appropriately restricts other users of the file handles, and we provide facilities within Haskell to express and enforce the relevant information flow policy.

Figure 2.4 shows the core of the public interface for facet-aware file handles, type `FHandle`.

We support policies that associate with each file handle h a set of labels $view_h$ of type `View`. This view indicates the confidentiality for data read from and written to h. Intuitively, if a view contains a label k, then that view is allowed to see data that is confidential to k.

The function `openFileFio` accepts a view $view_h$ along with a file path and mode and returns a (computation that returns a) facet-aware handle h protected by the policy $view_h$.

When writing to h via `putCharFio`, the view $view_h$ describes the confidentiality assured by the external environment for data written to h. In other words, we trust that the external world will protect the data with those labels in $view_h$.

When reading from a handle h via `getCharFio`, we treat $view_h$ as the confidentiality expected by the external world for data read from h. In other words, we certify that we protect the data received from h. For example, in the following computation, the character read from `h` is observable only to views that include labels `"k"` and `"l"`.

```
do h ← openFileFio ["k", "l"] "/tmp/socket.0" ReadMode
   getCharFio h
```

2.4 Formal Semantics

In this section, we formalize the behavior of the Haskell library as an operational semantics and prove that it guarantees termination-insensitive noninterference.

Figures 2.5 and 2.6 show the formal syntax. The syntactic class t represents Haskell programs, k is a label, and σ is a "store" mapping addresses a to values, and mapping file handles h to strings of characters ch.

For ease of understanding, we separate the set of values into three syntactic classes. *FacetedValue* contains values in the `Faceted` monad; *FioAction* contains computations in the impure `FIO` monad; and *Value* contains both of these, as well as ordinary values: closures, characters, labels, addresses, and handles.

$$
\begin{array}{lcllll}
ch & \in & \mathit{Character} & & & \\
k & \in & \mathit{Label} & & & \\
t & \in & \mathit{Term} & ::= & x & \\
 & & & \mid & \lambda x.t & \\
 & & & \mid & t\ t & \\
 & & & \mid & ch & \text{Character} \\
 & & & \mid & F & \text{Faceted values} \\
 & & & \mid & \texttt{faceted}\ k\ t\ t & \\
 & & & \mid & \texttt{return}^{\mathsf{Fac}}\ t & \\
 & & & \mid & \texttt{bind}^{\mathsf{Fac}}\ t\ t & \\
 & & & \mid & A & \text{FIO actions} \\
 & & & \mid & \texttt{prod}\ t & \\
 & & & \mid & () & \text{Unit value} \\
F & \in & \mathit{FacetedValue} & ::= & \texttt{public}\ t \mid \texttt{faceted}\ k\ F\ F \mid \texttt{bottom} & \\
A & \in & \mathit{FioAction} & ::= & \texttt{return}^{\mathsf{FIO}}\ t \mid \texttt{bind}^{\mathsf{FIO}}\ t\ t \mid \texttt{prod}\ F & \\
 & & & \mid & \texttt{newFioRef}\ t \mid \texttt{readFioRef}\ t \mid \texttt{writeFioRef}\ t\ t & \\
 & & & \mid & \texttt{getCharFio}\ t \mid \texttt{putCharFio}\ t\ t &
\end{array}
$$

Figure 2.5: Source syntax.

$$
\begin{array}{lclll}
a & \in & \mathit{Address} & & \\
h & \in & \mathit{Handle} & & \\
t & \in & \mathit{Term} & ::= & \ldots \mid a \mid h \\
v & \in & \mathit{Value} & ::= & F \mid A \mid \lambda x.t \mid ch \mid a \mid h \mid () \\
E & \in & \mathit{EvalContext} & ::= & \bullet\ t \mid \texttt{bind}^{\mathsf{Fac}}\ \bullet\ t \mid \texttt{faceted}\ k\ \bullet\ t \mid \texttt{faceted}\ k\ F\ \bullet \\
 & & & \mid & \texttt{prod}\ \bullet \\
\sigma & \in & \mathit{Store} & = & (\mathit{Address} \rightarrow \mathit{Term}) \cup (\mathit{Handle} \rightarrow \mathit{String})
\end{array}
$$

Figure 2.6: Runtime syntax.

$\boxed{t \Downarrow v}$ **Pure evaluation.**

$$\frac{}{v \Downarrow v} \quad \text{[E-VAL]}$$

$$\frac{t[x := t_1] \Downarrow v}{(\lambda x.t)\ t_1 \Downarrow v} \quad \text{[E-APP]}$$

$$\frac{\begin{array}{c} t \text{ not a value} \\ t \Downarrow v_1 \\ E[v_1] \Downarrow v_2 \end{array}}{E[t] \Downarrow v_2} \quad \text{[E-CTXT]}$$

$$\frac{}{\texttt{return}^{\textsf{Fac}}\ t \Downarrow \texttt{public}\ t} \quad \text{[E-RET]}$$

$$\frac{t_2\ t_1 \Downarrow F}{\texttt{bind}^{\textsf{Fac}}\ (\texttt{public}\ t_1)\ t_2 \Downarrow F} \quad \text{[E-BIND-P]}$$

$$\frac{\begin{array}{c} \texttt{bind}^{\textsf{Fac}}\ F_1\ t_3 \Downarrow F_1' \\ \texttt{bind}^{\textsf{Fac}}\ F_2\ t_3 \Downarrow F_2' \\ F = \texttt{faceted}\ k\ F_1'\ F_2' \end{array}}{\texttt{bind}^{\textsf{Fac}}\ (\texttt{faceted}\ k\ F_1\ F_2)\ t_3 \Downarrow F} \quad \text{[E-BIND-F]}$$

$$\frac{}{\texttt{bind}^{\textsf{Fac}}\ \texttt{bottom}\ t \Downarrow \texttt{bottom}} \quad \text{[E-BIND-B]}$$

Figure 2.7: Semantics (part 1).

We define the operational semantics with two big-step evaluation judgments.

- $t \Downarrow v$ means that the pure Haskell expression t evaluates to the value v.
- $\sigma, A \Downarrow_{pc}^{\textsf{FIO}} \sigma', v$ means that the Haskell program "`main = runFIO` A pc" changes the store from σ to σ' and yields the result v.

Figure 2.7 depicts the pure derivation rules. These rules describe a call-by-name λ-calculus with opaque constants and two library functions: $\texttt{return}^{\textsf{Fac}}$ and $\texttt{bind}^{\textsf{Fac}}$. These monad operators for `Faceted` are particularly simple because it is a free

$\boxed{\sigma, A \Downarrow_{pc}^{\mathsf{FIO}} \sigma, t}$ **Impure faceted computation.**

$$\frac{}{\sigma, \texttt{return}^{\mathsf{FIO}}\ t \Downarrow_{pc}^{\mathsf{FIO}} \sigma, t} \quad [\text{F-RET}]$$

$$\frac{\begin{array}{c} t_1 \Downarrow A_1 \\ \sigma_0, A_1 \Downarrow_{pc}^{\mathsf{FIO}} \sigma_1, t_3 \\ t_2\ t_3 \Downarrow A_2 \\ \sigma_1, A_2 \Downarrow_{pc}^{\mathsf{FIO}} \sigma_2, t_4 \end{array}}{\sigma_0, \texttt{bind}^{\mathsf{FIO}}\ t_1\ t_2 \Downarrow_{pc}^{\mathsf{FIO}} \sigma_2, t_4} \quad [\text{F-BIND}]$$

$$\frac{\begin{array}{c} t \Downarrow A \\ \sigma, A \Downarrow_{pc}^{\mathsf{FIO}} \sigma', t' \end{array}}{\sigma, \texttt{prod}\ (\texttt{public}\ t) \Downarrow_{pc}^{\mathsf{FIO}} \sigma', t'} \quad [\text{F-PROD-P}]$$

$$\frac{}{\sigma, \texttt{prod bottom} \Downarrow_{pc}^{\mathsf{FIO}} \sigma, \texttt{bottom}} \quad [\text{F-PROD-B}]$$

$$\frac{k \in pc \qquad \sigma, \texttt{prod}\ F_1 \Downarrow_{pc}^{\mathsf{FIO}} \sigma', t'}{\sigma, \texttt{prod}\ (\texttt{faceted}\ k\ F_1\ F_2) \Downarrow_{pc}^{\mathsf{FIO}} \sigma', t'} \quad [\text{F-PROD-F1}]$$

$$\frac{\overline{k} \in pc \qquad \sigma, \texttt{prod}\ F_2 \Downarrow_{pc}^{\mathsf{FIO}} \sigma', t_2'}{\sigma, \texttt{prod}\ (\texttt{faceted}\ k\ F_1\ F_2) \Downarrow_{pc}^{\mathsf{FIO}} \sigma', t'} \quad [\text{F-PROD-F2}]$$

$$\frac{\begin{array}{c} k \notin pc \qquad \overline{k} \notin pc \\ \sigma_0, \texttt{prod}\ F_1 \Downarrow_{pc\cup\{k\}}^{\mathsf{FIO}} \sigma_1, t_1 \\ \sigma_1, \texttt{prod}\ F_2 \Downarrow_{pc\cup\{\overline{k}\}}^{\mathsf{FIO}} \sigma_2, t_2 \\ t' = \texttt{faceted}\ k\ t_1\ t_2 \end{array}}{\sigma_0, \texttt{prod}\ (\texttt{faceted}\ k\ F_1\ F_2) \Downarrow_{pc}^{\mathsf{FIO}} \sigma_2, t'} \quad [\text{F-PROD-F3}]$$

Figure 2.8: Semantics (part 2).

$$\frac{t \Downarrow F \quad a \notin \mathrm{dom}(\sigma) \quad F' = \langle\langle pc \;?\; F : \texttt{bottom}\rangle\rangle}{\sigma, \texttt{newFioRef}\ t \Downarrow^{\mathsf{FIO}}_{pc} \sigma[a := F'], a} \quad [\text{F-NEW}]$$

$$\frac{t \Downarrow a}{\sigma, \texttt{readFioRef}\ t \Downarrow^{\mathsf{FIO}}_{pc} \sigma, \sigma(a)} \quad [\text{F-READ}]$$

$$\frac{t_1 \Downarrow a \quad \sigma' = \sigma[a := \langle\langle pc \;?\; t_2 : \sigma(a)\rangle\rangle] \quad v = \texttt{public}\ ()}{\sigma, \texttt{writeFioRef}\ t_1\ t_2 \Downarrow^{\mathsf{FIO}}_{pc} \sigma', v} \quad [\text{F-WRITE}]$$

$$\frac{t \Downarrow h \quad pc \text{ is not visible to } view_h}{\sigma, \texttt{getCharFio}\ t \Downarrow^{\mathsf{FIO}}_{pc} \sigma, \texttt{bottom}} \quad [\text{F-GET-2}]$$

$$\frac{t \Downarrow h \quad L = view_h \quad pc \text{ is visible to } L \quad ch_1 \ldots ch_n = \sigma(h) \quad \sigma' = \sigma[h := ch_2 \ldots ch_n] \quad pc' = L \cup \{\overline{k} \mid k \notin L\} \quad F = \langle\langle pc' \;?\; \texttt{public}\ ch_1 : \texttt{bottom}\rangle\rangle}{\sigma, \texttt{getCharFio}\ t \Downarrow^{\mathsf{FIO}}_{pc} \sigma', F} \quad [\text{F-GET}]$$

$$\frac{t_1 \Downarrow h \quad L = view_h \quad pc \text{ is visible to } L \quad t_2 \Downarrow ch \quad \sigma' = \sigma[h := \sigma(h)ch]}{\sigma, \texttt{putCharFio}\ t_1\ t_2 \Downarrow^{\mathsf{FIO}}_{pc} \sigma', ()} \quad [\text{F-PUT}]$$

$$\frac{t_1 \Downarrow h \quad L = view_h \quad pc \text{ is not visible to } L}{\sigma, \texttt{putCharFio}\ t_1\ t_2 \Downarrow^{\mathsf{FIO}}_{pc} \sigma, ()} \quad [\text{F-PUT-2}]$$

Figure 2.9: Semantics (part 3).

monad: $\texttt{bind}^{\mathsf{Fac}}\ F\ v$ replaces the `public` "leaves" of the faceted value F with new faceted values obtained by calling v.

Figure 2.8 shows the impure derivation rules. The `FIO` monad operations (defined by [F-RET] and [F-BIND]) are typical of a state monad. The *pc* annotation propagates unchanged through these trivial rules.

The next five rules define `prod`, whose type is:

```
Faceted (FIO (Faceted a)) -> FIO (Faceted a)
```

The input, a faceted action, is transformed into an action that returns a faceted value. This process is straightforward for `public` and `bottom`; the `public` constructor is simply stripped away to reveal the action underneath, while `bottom` is simply transformed into a no-op. For `faceted`, the corresponding rule is [F-PROD-F3], where the process *bifurcates* into two subcomputations whose results are combined into a `faceted` result value. However, there is no need to bifurcate repeatedly for the same label k, so the bifurcation is remembered by adding k (or $\overline{k}$) to the *pc* annotation on each subcomputation. Subsequently, the optimized rules [F-PROD-F1] and [F-PROD-F2] will apply. Rather than bifurcating the computation, these rules will execute only the one path of computation that is relevant to the current *pc*.

The remainder of Figure 2.8 shows the rules for creation and manipulation of reference cells, and for input and output.

[F-NEW] describes the creation of a new faceted reference cell. To preserve the noninterference property, the cell is initialized with a faceted value that hides the true value from observers that should not know about the cell. The notation

$\langle\langle \bullet \ ? \ \bullet : \bullet \rangle\rangle$ means:

$$\langle\langle \emptyset \ ? \ t_1 : t_2 \rangle\rangle = t_1$$

$$\langle\langle \{k\} \cup pc \ ? \ t_1 : t_2 \rangle\rangle = \texttt{faceted}\ k\ \langle\langle pc \ ? \ t_1 : t_2 \rangle\rangle\ t_2$$

$$\langle\langle \{\overline{k}\} \cup pc \ ? \ t_1 : t_2 \rangle\rangle = \texttt{faceted}\ k\ t_2\ \langle\langle pc \ ? \ t_1 : t_2 \rangle\rangle$$

[F-READ] and [F-WRITE] read and write these reference cells. [F-READ] is simple because the values in the store σ will already be appropriately faceted. To prevent implicit flows, [F-WRITE] must incorporate the *pc* into the label of the value stored.

The final rules handle input and output. Each must first confirm that the file handle h is compatible with the current *pc*. The notation "*pc* is visible to L" means

$$\forall k \in pc, k \in L \qquad\qquad \text{and} \qquad\qquad \forall \overline{k} \in pc, k \notin L,$$

i.e. L is one of the views being simulated on the current branch of computation.

In [F-GET], if *pc* is visible to L, then the first character ch_1 is extracted from the file. The result is a faceted value that behaves as ch_1 for view L, but as `bottom` for all other views. If *pc* is not visible to L, then [F-GET-2] applies and the operation is ignored; the result is simply `bottom`.

In [F-PUT], if *pc* is visible to L, then a character is appended to the end of the file; otherwise, [F-PUT-2] applies and the operation is ignored.

2.4.1 Termination-Insensitive Noninterference

We first define the projection $\varepsilon_L(t)$ of a term t according to a view $L \in 2^{Label}$:

$$\varepsilon_L(\texttt{faceted}\ k\ t_1\ t_2) = \varepsilon_L(t_1) \qquad \text{if } k \in L$$

$$\varepsilon_L(\texttt{faceted}\ k\ t_1\ t_2) = \varepsilon_L(t_2) \qquad \text{if } k \notin L$$

$\varepsilon_L(\bullet)$ is homomorphic otherwise.

Similarly, we define the projection $\varepsilon_L(\sigma)$ of a store σ according to a view L:

$$\varepsilon_L(\sigma)(a) = \varepsilon_L(\sigma(a))$$

$$\varepsilon_L(\sigma)(h) = \begin{cases} \sigma(h) & \text{if } L = view_h \\ \epsilon & \text{otherwise} \end{cases}$$

where ϵ denotes the empty string. In words, the projected store maps each address to the projection of the stored value, and the projected store maps each handle either to the real file contents (if the viewer is $view_h$) or to ϵ.

A *state* is a pair of a store and a term. We identify states that are equivalent modulo alpha-renaming of addresses.

Theorem 1 (Termination-Insensitive Noninterference). Assume:

$$\varepsilon_L(\sigma_1) = \varepsilon_L(\sigma_2) \qquad \varepsilon_L(A_1) = \varepsilon_L(A_2)$$

$$\sigma_1, A_1 \Downarrow_{\emptyset}^{\mathsf{FIO}} \sigma_1', v_1 \qquad \sigma_2, A_2 \Downarrow_{\emptyset}^{\mathsf{FIO}} \sigma_2', v_2$$

Then:

$$\varepsilon_L(\sigma_1') = \varepsilon_L(\sigma_2') \qquad \varepsilon_L(v_1) = \varepsilon_L(v_2).$$

In other words, if we run two programs that are identical under the L projection, then the results will be identical under the L projection.

The proof is available in the attached Coq script.

2.5 Application: A Bi-Monadic Interpreter

To demonstrate the expressiveness of the `Faceted` library, we present a monadic interpreter for an imperative λ-calculus, whose dynamic information flow security is guaranteed by the previous noninterference theorem.

The interesting aspect about this interpreter is that it uses two distinct monads.

- The `FIO` monad captures computations (called `Action`s in the code), and is propagated along control flow paths in the traditional style of monadic interpreters.

- The `Faceted` monad serves a somewhat different purpose, which is to encapsulate the many views of the underlying `RawValue`. Unlike `FIO`, this monad is propagated along data flow paths rather than along control flow paths.

Even though the interpreter's use of the `Faceted` monad is non-traditional, faceted values need exactly this monad interface – particularly considering the necessity of the monad-specific operation

$$\texttt{join :: Faceted (Faceted } a\texttt{)} \rightarrow \texttt{Faceted } a$$

which, for the `Faceted` monad, naturally combines two layers of security labels into a single layer.

```
-- Abstract syntax tree data structure.
data Term =
    Var String                  -- Lambdas
  | Lam String Term
  | App Term Term
  | Const Value                 -- Constants

-- Runtime data structures.
data RawValue =
    CharVal Char                -- Characters
  | RefVal (FioRef Value)       -- Mutable references
  | FnVal (Value → Action)      -- Functions
type Value  = Faceted RawValue
type Action = FIO Value
type Env    = String → Value
```

Figure 2.10: Syntax for the bi-monadic interpreter.

2.5.1 The Interpreted Language

The source language is an imperative call-by-value λ-calculus whose abstract syntax is defined in Figure 2.10. The language has variables, lambda abstractions, applications, and primitive constants for manipulating reference cells, performing I/O, and creating private values.

To ensure that private characters are not printed to the output stream, our implementation opens the stream using the empty view.

2.5.2 Implementation

Figure 2.11 shows the core of the interpreter, the function `eval`. As usual, it takes an environment and a term and returns an action, which has type `Action = FIO (Faceted RawValue)`. The `RawValue` type includes characters, mutable references, and closures.

The most interesting code is the case for an application `App t1 t2` (lines 15-19

```
-- Interpreter.
eval :: Env → Term → Action
eval e (Var x)     = return $ e x
eval e (Lam x t)   = return $ return $ FnVal $ λv →
                        eval (extend e x v) t
eval e (App t1 t2) = do v1 ← eval e t1     -- working in FIO monad
                         v2 ← eval e t2
                         prod $ do
                           FnVal f ← v1     -- working in Faceted monad
                           return $ f v2
eval e (Const v)   = return v

-- Constants.
private :: RawValue
private = FnVal $ λv →
  return $ faceted "H" v bottom
ref :: RawValue
ref = FnVal $ λv → do                       -- working in FIO monad
  ref ← newFioRef v
  return $ return $ RefVal ref
deref :: RawValue
deref = FnVal $ λv → prod $ do              -- working in Faceted monad
  RefVal ref ← v
  return $ readFioRef ref
assign :: RawValue
assign = FnVal $ λv1 →
  return $ return $ FnVal $ λv2 → prod $ do-- working in Faceted monad
    RefVal ref ← v1
    rv2 ← v2
    return $ do                             -- working in FIO monad
      writeFioRef ref v2
      return v2
printChar :: RawValue
printChar = FnVal $ λv → prod $ do          -- working in Faceted monad
  CharVal c ← v
  return $ do                               -- working in FIO monad
    h ← openFileFio [] "output.txt" AppendMode
    putCharFio h c
    closeFio h
    return v
```

Figure 2.11: The bi-monadic interpreter `eval` function.

```
let x = ref (private true)  in
let y = ref true  in
let z = ref true  in
let vx = deref x  in
if (vx) {
  assign y false
}
let vy = deref y  in
if (vy) {
  assign z false
}
deref z
```

Figure 2.12: A sample program for the interpreter. For ease of reading, we assume the availability of standard encodings for `let` and boolean operations.

in Figure 2.11). As usual, we use a `do` block (in the `FIO` monad) to compose the sub-`eval`uations of `t1` and `t2` into faceted values `v1` and `v2`. To extract each underlying function (`FnVal f`) from the faceted value `v1`, we enter a second `do` block (this time in the `Faceted` monad), and then apply `f` to `v2` to yield a result of type `Action = FIO (Faceted RawValue)`, which the `return` (on line 19) then injects into type `Faceted (FIO (Faceted RawValue))`, completing the `Faceted` `do` block (lines 17-19). Finally, the `prod` function on line 17 coordinates the two monads and simplifies the type to `FIO (Faceted RawValue)`, which sequentially composes with the previous sub-evaluations of `t1` and `t2`.

The remaining language features are provided by the constants below the interpreter itself: `private`, `ref`, `deref`, `assign`, and `printChar`. As for `App`, these constants must use `prod` to perform their services securely.

Figure 2.12 expresses our running example from Figure 1 as a program p in the interpreted language (with some additional syntactic sugar); running the program `runFIO (eval env` p`) []` yields the expected result:

```
faceted "H" (public true) (public false)
```

2.6 Related Work

Most information flow mechanisms fall into one of three categories: run-time monitors that prevent a program execution from misbehaving; static analysis techniques that analyze the whole program and reject programs that might leak sensitive information; and finally secure multi-execution, which protects sensitive information by evaluating the same program multiple times.

Dynamic techniques dominated much of the early literature, such as Fenton's memoryless subsystems [11]. However, these approaches tend to deal poorly with *implicit flows*, where confidential information might leak via the control flow of the program; purely dynamic controls either ignore updates to reference cells that might result in implicit leaks of information [11] or terminate the program on these updates [35, 1]; both approaches have obvious problems, but these techniques have seen a resurgence of interest as a possible means of securing JavaScript code, where static analysis seems to be an awkward fit [10, 15, 13, 18].

Denning's work [6, 7] instead uses a static analysis; her work was also instrumental in bringing information flow analysis into the scope of programming language research. Her approach has since been codified into different type systems, such as that of Volpano et al. [32] and the SLam Calculus [14]. Jif [21] uses this strategy for a Java-like language, and has become one of the more widespread languages providing information flow guarantees. Sabelfeld and Myers [26] provide an excellent history of information flow analysis research prior to 2003. Refer to Russo [25] for a detailed comparison of static and dynamic techniques.

Secure multi-execution [9] executes the same program multiple times representing different "views" of the data. For a simple two-element lattice of high and low, a program is executed twice: one execution can access confidential (high) data but can only write to authorized channels, while the other replaces all high data with default values and can write to public channels. This approach has since been implemented in the Firefox web browser [5] and as a Haskell library [16].

Rafnsson and Sablefeld[23] show an approach to handle declassification and to guarantee transparency with secure multi-execution.

Zanarini et al. [34] notes some challenges with secure multi-execution; specifically, it alters the behavior of programs violating noninterference (potentially introducing difficult to analyze bugs), and the multiple processes might produce outputs to different channels in a different order than expected. They further address these challenges through a *multi-execution monitor*. In essence, their approach executes the original program without modification and compares its results to the results of the SME processes; if output of secure multi-execution differs from the original at any point, a warning can be raised to note that the semantics have been altered.

Faceted evaluation [2] simulates secure multi-execution by the use of special faceted values, which track different views for data based on the security principals involved[4]. While faceted evaluation cannot be parallelized as easily, it avoids many redundant calculations, thereby improving efficiency [2]. It also allows declassification, where private data is released to public channels. Austin et al. [3] exploit this benefit to incorporate policy-agnostic programming techniques, allowing for the specification of more flexible policies than traditionally permitted in information flow systems.

Li and Zdancewic [19] implement an information flow system in Haskell, em-

[4]Faceted values are closely related to the value pairs used by [22]; while intended as a proof technique rather than a dynamic enforcement mechanism, the construct is essentially identical.

bedding a language for creating secure modules. Their enforcement mechanism is dynamic but relies on static enforcement techniques, effectively guaranteeing the security of the system by type checking the embedded code at runtime. Their system supports declassification, a critical requirement for specifying many real world security policies.

Russo et al. [24] provide a monadic library guaranteeing information flow properties. Their approach includes special declassification combinators, which can be used to restrict the release of data based on the what/when/who dimensions proposed by Sabelfeld [28].

Deviese and Piessens [8] illustrate how to enforce information flow in monadic libraries. A sequence operation $e_1 \gg e_2$ is distinguished from a bind operation $e_1 \gg= e_2$ in that there are no implicit flows with the $\gg$ operator. They demonstrate the generality of their approach by applying it to classic static [32], dynamic [27], and hybrid [12] information flow systems.

Stefan et al. [30] use a *labeled IO* (LIO) monad to guarantee information flow analysis. LIO tracks the current label of the execution, which serves as an upper bound on the labels of all data in lexical scope. IO is permitted only if it would not result in an implicit flow. It combines this notion with the concept of a *current clearance* that limits the maximum privileges allowed for an execution, thereby eliminating the termination channel. Buiras and Russo[4] show how lazy evaluation may leak secrets with LIO through the use of the *internal timing covert channel*. They propose a defense against this attack by duplicating shared thunks.

Wadler [33] describes the use of monads to structure interpreters for effectful languages. There has been great effort to improve the modularity of this technique, including the application of pseudomonads [29] and of monad transformers [20]. Both of these approaches make it possible to design an interpreter's computation monad by composing building blocks that each encapsulate one kind of effect. Our bi-monadic interpreter achieves a different kind of modularity by using separate monads for effects and values. The use of a *prod* function, which links the two monads together, is originally described by Jones and Duponcheel [17].

2.7 Conclusion

We show how the *faceted values* technique can be implemented as a library rather than as a language extension. Our implementation draws on the previous work to provide a library consisting primarily of two monads, which track both explicit and implicit information flows. This implementation demonstrates how faceted values look in a typed context, as well as how they might be implemented as a library rather than a language feature. It also illustrates some of the subtle interactions between two monads. Our interpreter shows that this library can serve as a basis for other faceted value languages or as a template for further Haskell work.

Acknowledgments This research was supported by the National Science Foundation under grants CCF-1337278 and CCF-1421016.

Bibliography

[1] Thomas H. Austin and Cormac Flanagan. "Efficient Purely-dynamic Information Flow Analysis". In: PLAS '09. ACM Press, 2009.

[2] Thomas H. Austin and Cormac Flanagan. "Multiple Facets for Dynamic Information Flow". In: POPL '12. New York, NY, USA: ACM Press, 2012, 165–178.

[3] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. "Faceted Execution of Policy-agnostic Programs". In: PLAS '13. New York, NY, USA: ACM Press, 2013, 15–26.

[4] Pablo Buiras and Alejandro Russo. "Lazy Programs Leak Secrets". In: ed. by Hanne Riis Nielson and Dieter Gollmann. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2013, pp. 116–122.

[5] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. "FlowFox: A Web Browser with Flexible and Precise Information Flow Control". In: CCS '12. New York, NY, USA: ACM Press, 2012.

[6] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". In: *Communications of the ACM* 19.5 (May 1976), 236–243.

[7] Dorothy E. Denning and Peter J. Denning. "Certification of programs for secure information flow". In: *Communications of the ACM* 20.7 (1977), 504–513.

[8] Dominique Devriese and Frank Piessens. "Information Flow Enforcement in Monadic Libraries". In: TLDI '11. New York, NY, USA: ACM Press, 2011, 59–72.

[9] Dominique Devriese and Frank Piessens. "Noninterference through Secure Multi-execution". In: *Symposium on Security and Privacy*. Los Alamitos, CA, USA: IEEE, 2010.

[10] Mohan Dhawan and Vinod Ganapathy. "Analyzing Information Flow in JavaScript-Based Browser Extensions". In: *ACSAC*. IEEE, 2009.

[11] J. S. Fenton. "Memoryless Subsystems". In: *The Computer Journal* 17.2 (1974), pp. 143–147.

[12] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. “Automata-based Confidentiality Monitoring”. In: *In ASIAN’06: the 11th Asian Computing Science Conference on Secure Software*. 2006.

[13] Daniel Hedin and Andrei Sabelfeld. “Information-flow security for a core of JavaScript”. In: *CSF*. IEEE, 2012.

[14] Nevin Heintze and Jon G. Riecke. “The SLam Calculus: Programming with Secrecy and Integrity”. In: *POPL*. ACM, 1998.

[15] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. “An empirical study of privacy-violating information flows in JavaScript web applications”. In: *ACM Conference on Computer and Communications Security*. 2010.

[16] Mauro Jaskelioff and Alejandro Russo. “Secure Multi-execution in Haskell”. In: PSI’11. Berlin, Heidelberg: Springer-Verlag, 2012, 170–178.

[17] Mark P. Jones and Luc Duponcheel. *Composing Monads*. Tech. rep. Research Report YALEU/DCS/RR-1004. Yale University, 1993.

[18] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. “Towards Precise and Efficient Information Flow Control in Web Browsers”. In: *Trust and Trustworthy Computing Conference*. Springer, 2013.

[19] Peng Li and Steve Zdancewic. “Encoding Information Flow in Haskell”. In: CSFW ’06. Washington, DC, USA: IEEE Computer Society, 2006, 16–.

[20] Sheng Liang, Paul Hudak, and Mark Jones. “Monad Transformers and Modular Interpreters”. In: *Proceedings of 22nd ACM Symposium on Principles of Programming Languages*. New York: ACM Press, 1995.

[21] Andrew C. Myers. “JFlow: Practical Mostly-Static Information Flow Control”. In: *Symposium on Principles of Programming Languages (POPL)*. ACM, 1999.

[22] François Pottier and Vincent Simonet. “Information Flow Inference for ML”. In: *ACM Trans. Program. Lang. Syst.* 25.1 (Jan. 2003), 117–158.

[23] W. Rafnsson and A. Sabelfeld. “Secure Multi-execution: Fine-Grained, Declassification-Aware, and Transparent”. In: *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*. June 2013.

[24] Alejandro Russo, Koen Claessen, and John Hughes. “A Library for Lightweight Information-flow Security in Haskell”. In: Haskell ’08. New York, NY, USA: ACM, 2008, 13–24.

[25] Alejandro Russo and Andrei Sabelfeld. “Dynamic vs. Static Flow-Sensitive Security Analysis”. In: CSF ’10. Washington, DC, USA: IEEE Computer Society, 2010, 186–199.

[26] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security". In: *Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19.

[27] Andrei Sabelfeld and Alejandro Russo. "From Dynamic to Static and Back: Riding the Roller Coaster of Information-flow Control Research". In: PSI'09. Berlin, Heidelberg: Springer-Verlag, 2010.

[28] Andrei Sabelfeld and David Sands. "Declassification: Dimensions and Principles". In: *Journal of Computer Security* 17.5 (Oct. 2009), 517–548.

[29] Guy L. Steele Jr. "Building Interpreters by Composing Monads". In: POPL '94. Portland, Oregon, USA: ACM, 1994.

[30] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. "Flexible Dynamic Information Flow Control in Haskell". In: Haskell '11. New York, NY, USA: ACM, 2011, 95–106.

[31] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. *Flexible dynamic information flow control in Haskell*. Vol. 46. 12. ACM, 2011.

[32] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. "A sound type system for secure flow analysis". In: *Journal of Computer Security* 4.2-3 (1996), 167–187.

[33] Philip Wadler. "The Essence of Functional Programming". In: POPL '92. Albuquerque, New Mexico, USA: ACM, 1992.

[34] Dante Zanarini, Mauro Jaskelioff, and Alejandro Russo. *Precise Enforcement of Confidentiality for Reactive Systems*. 2013.

[35] Stephan Arthur Zdancewic. "Programming languages for information security". PhD thesis. Cornell University, 2002.

Chapter 3

Faceted Secure Multi Execution

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.

3.1 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 [57, 43], 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* [17] (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 [7]—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. [41] 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., [38, 48, 21, 29, 37, 22, 33, 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 [7]. 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 [32, 44, 49], 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].

3.2 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

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

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 [17], 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 [57, 43]).

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.

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 [41] 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.

3.3 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 3.A. Following Haskell, we distinguish between pure and side-effecting computations.

3.3.1 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.

3.3.2 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., $\mathsf{raw}\ 42$ should be thought of as simply 42. For instance, the faceted value $\langle H\,?\,42 : 0\rangle$ from Section 3.2 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 $\mathsf{Fac\ Int}$ 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 $f :: T_1 \rightarrow \mathsf{Fac}\ T_2$, then $\mathsf{bind}\ V\ f$

- ▶ 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 $\mathsf{Fac}\ T_2$, and
- ▶ joins these various results from f into a single faceted value of type $\mathsf{Fac}\ T_2$,

which is returned from bind.

3.3.3 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 $\mathsf{FIO}\ T$ characterizes side-effectful secure computations that yield a T value. Because of being a monad, computations of type $\mathsf{FIO}\ T$ are built by two fundamental operations:

$$\begin{aligned} \mathsf{return} &:: T \rightarrow \mathsf{FIO}\ T \\ (\gg\!=) &:: \mathsf{FIO}\ T_1 \rightarrow (T_1 \rightarrow \mathsf{FIO}\ T_2) \rightarrow \mathsf{FIO}\ T_2 \end{aligned}$$

The operation $\mathsf{return}\ x$ 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, $\mathit{fio} \gg\!= f$ 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 $\mathit{fio} \gg\!= \lambda x.\mathsf{return}\ (x+1)$, which adds 1 to the value produced by computation *fio*, can be written as

$$\begin{aligned}&\mathbf{do}\ x \leftarrow \mathit{fio}\\ &\quad\ \mathsf{return}\ (x+1)\end{aligned}$$

which gives a more "imperative" feeling to programs.

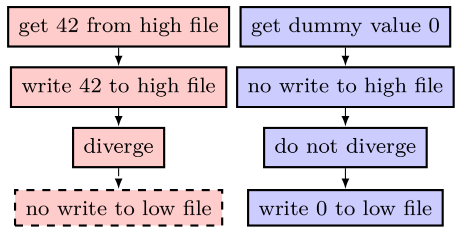
3.3.4 Building side-effectful computations based on faceted values

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:

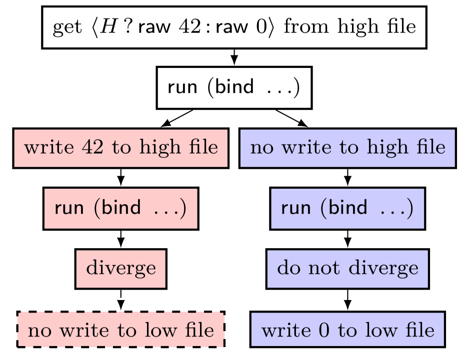
$$\mathsf{if}\ (x > 0)\ \mathsf{effect}_0\ \mathsf{effect}_1 :: \mathsf{FIO}\ ()$$

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

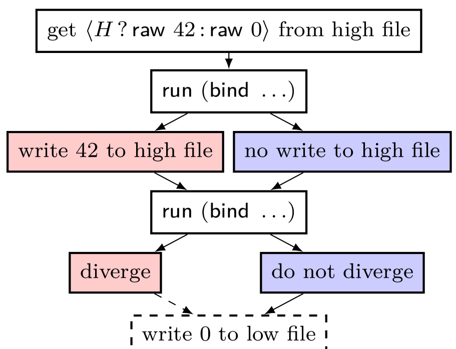
$$\mathsf{run} :: \mathsf{Fac}\ (\mathsf{FIO}\ T) \rightarrow \mathsf{FIO}\ (\mathsf{Fac}\ T)$$



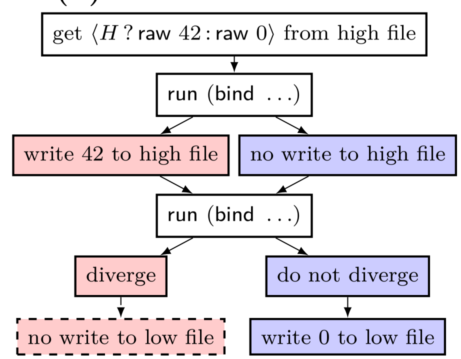
(a) SME.



(b) Demand-driven SME.



(c) MF.



(d) FSME.

Figure 3.1: 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).

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

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

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.

3.3.5 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

[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.

consider the following code fragment:

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

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 $\mathit{fx} = \langle H\,?\,\mathsf{raw}\ 42 : \mathsf{raw}\ 0\rangle$, 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 [17] would run two versions of the program, as shown in Figure 3.1a. 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 3.1b, 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 3.1c 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 3.1d 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 [41] and it proves crucial to get good performance in our implementation (see Sections 3.9 and 3.10). 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.

3.3.6 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 $l \in \mathit{Lattice}$ 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:

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

For example, $\mathit{views}(\{k_1, k_2, \overline{k}_3, \overline{k}_4\})$ 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 3.2—see Appendix 3.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

Multef syntax

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

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

$$
\begin{array}{llll}
\mathsf{run}\ \langle k\,?\,t_1 : t_2\rangle & \longrightarrow_{pc} \begin{cases} \mathsf{run}\ t_1 \\ \mathsf{run}\ t_2 \\ \langle\!\langle k\,?\,\mathsf{run}\ t_1 : \mathsf{run}\ t_2\rangle\!\rangle \end{cases} & \begin{array}{l} \text{if } \mathit{views}(pc \cup \{\overline{k}\}) = \emptyset \\ \text{if } \mathit{views}(pc \cup \{k\}) = \emptyset \\ \text{otherwise.} \end{array} & \begin{array}{l} [\textsc{F-Run-Facet-1}] \\ [\textsc{F-Run-Facet-2}] \\ [\textsc{F-Run-Facet-3}] \end{array} \\
\langle\!\langle k\,?\,t_1 : t_2\rangle\!\rangle & \longrightarrow_{pc} \langle\!\langle k\,?\,t_1' : t_2\rangle\!\rangle & \text{if } \overline{k} \notin pc \text{ and } t_1 \longrightarrow_{pc \cup \{k\}} t_1' & [\textsc{F-Thread-1}] \\
\langle\!\langle k\,?\,t_1 : t_2\rangle\!\rangle & \longrightarrow_{pc} \langle\!\langle k\,?\,t_1 : t_2'\rangle\!\rangle & \text{if } k \notin pc \text{ and } t_2 \longrightarrow_{pc \cup \{\overline{k}\}} t_2' & [\textsc{F-Thread-2}] \\
\langle\!\langle k\,?\,\mathsf{return}\ V_1 : \mathsf{return}\ V_2\rangle\!\rangle & \longrightarrow_{pc} \mathsf{return}\ \langle k\,?\,V_1 : V_2\rangle & & [\textsc{F-Merge}] \\
E[\langle\!\langle k\,?\,t_1 : t_2\rangle\!\rangle] & \longrightarrow_{pc} \langle\!\langle k\,?\,E[t_1] : E[t_2]\rangle\!\rangle & & [\textsc{F-Fork-Continuation}]
\end{array}
$$

Figure 3.2: Syntax and selected rules from the **Multef** semantics.

should only be visible at security levels in $\mathit{views}(pc \cup \{k\})$. 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 $\mathit{views}(pc)$ 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 $\mathsf{run}\ t_i$ for $i \in \{1, 2\}$ terminates to $\mathsf{return}\ V_i$, rule [F-MERGE] joins the two threads back together into a single terminated FIO computation $\mathsf{return}\ \langle k\,?\,V_1 : V_2\rangle$. 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 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 $\mathsf{put}\ o\ n$. 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 $\mathsf{get}\ i$ 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.

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

The previously described rules extend in a natural manner from terms to states. Figure 3.3 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:

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

Appendix 3.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.

3.4 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-

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

$$\begin{array}{llll}
(\mathsf{new}\ V, M, P, I, O) & \longrightarrow_{pc} (\mathsf{return}\ a, M[a := \langle\!\langle pc\,?\,V : \mathsf{raw}\ 0\rangle\!\rangle], P, I, O) & \text{if } a \notin \mathrm{dom}(M) & [\textsc{F-New}] \\
(\mathsf{read}\ a, M, P, I, O) & \longrightarrow_{pc} (\mathsf{return}\ M(a), M, P, I, O) & & [\textsc{F-Read}] \\
(\mathsf{write}\ a\ V, M, P, I, O) & \longrightarrow_{pc} (\mathsf{return}\ V, M', P, I, O) & \text{if } M' = M[a := \langle\!\langle pc\,?\,V : M(a)\rangle\!\rangle] & [\textsc{F-Write}]
\end{array}$$

Figure 3.3: Rules for references.

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 3.4 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 3.5. 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 $l \in \mathit{views}(pc)$. 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 $l \in \mathit{views}(pc)$. 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 $l \notin \mathit{views}(pc)$.

Theorem 1 (Projection)**.**

1. *If $\sigma \longrightarrow_{pc} \sigma'$ and $l \in \mathit{views}(pc)$, then either $\sigma \approx_l \sigma'$ or $\sigma{\downarrow_l} \xrightarrow{std} \sigma'{\downarrow_l}$.*
2. *If $\sigma \not\longrightarrow_{pc}$ and $l \in \mathit{views}(pc)$, then $\sigma{\downarrow_l} \not\xrightarrow{std}$.*
3. *If $(t, M, P, I, O) \longrightarrow_{pc} (t', M', P', I', O')$ and $l \notin \mathit{views}(pc)$, 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,

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

$$
\begin{array}{llll}
(\mathsf{new}\ F, M, P, I, O) & \xrightarrow{\text{std}} (\mathsf{return}\ a, M[a := F], P, I, O) & \text{if } a \notin \mathrm{dom}(M) & [\textsc{s-new}] \\
(\mathsf{write}\ a\ F, M, P, I, O) & \xrightarrow{\text{std}} (\mathsf{return}\ F, M[a := F], P, I, O) & & [\textsc{s-write}]
\end{array}
$$

Figure 3.4: Selected rules of the standard semantics.

$$
\begin{array}{lll}
\boxed{t{\downarrow_l} = t} & \langle k\,?\,F_1 : F_2\rangle{\downarrow_l} & = \begin{cases} F_1{\downarrow_l} & k \sqsubseteq l \\ F_2{\downarrow_l} & \text{otherwise} \end{cases} \\
 & \langle\!\langle k\,?\,t_1 : t_2\rangle\!\rangle{\downarrow_l} & = \begin{cases} t_1{\downarrow_l} & k \sqsubseteq l \\ t_2{\downarrow_l} & \text{otherwise} \end{cases} \\
 & (\mathsf{put}\ o\ t){\downarrow_l} & = \begin{cases} \mathsf{put}\ o\ (t{\downarrow_l}) & l_o = l \\ \mathsf{return}\ (t{\downarrow_l}) & \text{otherwise} \end{cases} \\
 & t{\downarrow_l} \text{ is homomorphic otherwise} & \\
\boxed{M{\downarrow_l} = M} & M{\downarrow_l} & = \lambda a.M(a){\downarrow_l} \\
\boxed{p{\downarrow_l} = p} & n{\downarrow_l} & = n \\
 & \langle k\,?\,p_1 : p_2\rangle{\downarrow_l} & = \begin{cases} p_1{\downarrow_l} & k \sqsubseteq l \\ p_2{\downarrow_l} & \text{otherwise} \end{cases} \\
\boxed{P{\downarrow_l} = P} & P{\downarrow_l} & = \lambda i.P(i){\downarrow_l} \\
\boxed{I{\downarrow_l} = I} & I{\downarrow_l} & = \lambda i.\begin{cases} I(i) & l_i \sqsubseteq l \\ \epsilon & \text{otherwise} \end{cases} \\
\boxed{O{\downarrow_l} = O} & O{\downarrow_l} & = \lambda o.\begin{cases} O(o) & l_o = l \\ \epsilon & \text{otherwise} \end{cases} \\
\boxed{\sigma{\downarrow_l} = \sigma} & (t, M, P, I, O){\downarrow_l} & = (t{\downarrow_l}, M{\downarrow_l}, P{\downarrow_l}, I{\downarrow_l}, O{\downarrow_l})
\end{array}
$$

Figure 3.5: Projection functions.

and run both computations with the empty $pc = \emptyset$, so the faceted framework simulates standard evaluation for all views.

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* $\sigma'_1 \approx_l \sigma'_2$.

3.5 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:

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

where $t_2 = \mathsf{put}\ \mathit{publicFile}\ 3$ 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*.

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

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.

3.6 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, Piessens, and Rezk [39] 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.

[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.

3.7 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 3.2) for a given pc and label k.

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

Recall that the definition of $\mathit{views}(pc)$ hinges on quantifying over all labels in the lattice. The definition of $\mathit{views}(pc)$ in Section 3.3 is:

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

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 $\mathit{views}(pc) = \emptyset$ 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., [41, 4]) 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 $\mathit{views}(pc)$. To check if $\mathit{views}(pc)$ 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 [50] and DLM [34].

3.7.1 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 $\texttt{Alice} \wedge \texttt{Bob}$ indicates that information is sensitive to both principals and requires their joint consensus to observe it. In contrast, $\texttt{Alice} \vee \texttt{Bob}$ 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 $\texttt{Alice} \wedge \texttt{Bob}$ 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 $\texttt{Alice} \lor \texttt{Bob}$ 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 $L_1 = \langle \mathsf{Bob}, \mathsf{Bob} \lor \mathsf{Alice} \rangle$, where data is confidential to Bob but he does not assume full responsibility for it, and label $L_2 = \langle \mathsf{Bob} \land \mathsf{Alice}, \mathsf{Bob} \rangle$ 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 $\mathsf{Alice} \land \mathsf{Bob} \vdash \mathsf{Bob}$. However, $\mathsf{Alice} \lor \mathsf{Bob} \nvdash \mathsf{Bob}$; 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 [20, 1].

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.

$$\langle C_0, I_0 \rangle \sqcup \langle C_1, I_1 \rangle = \langle C_0 \land C_1, I_0 \lor I_1 \rangle$$

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.

3.8 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.

3.8.1 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 3.7.1 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 $\langle \bullet \mathbin{?} \bullet : \bullet \rangle$ in our calculus, respectively. With faceted values in place, we proceed to provide the FIO operations in our calculus.

```haskell
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 3.B for further details.

3.8.2 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 3.2, 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 3.A, Figure 3.13. For instance, `execute (Return t :>>=: rest)` is equivalent to `execute (rest t)`—thus matching the rule [F-BIND-FIO] in Figure 3.10. 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.

3.8.3 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 3.3.

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 3.B for details.

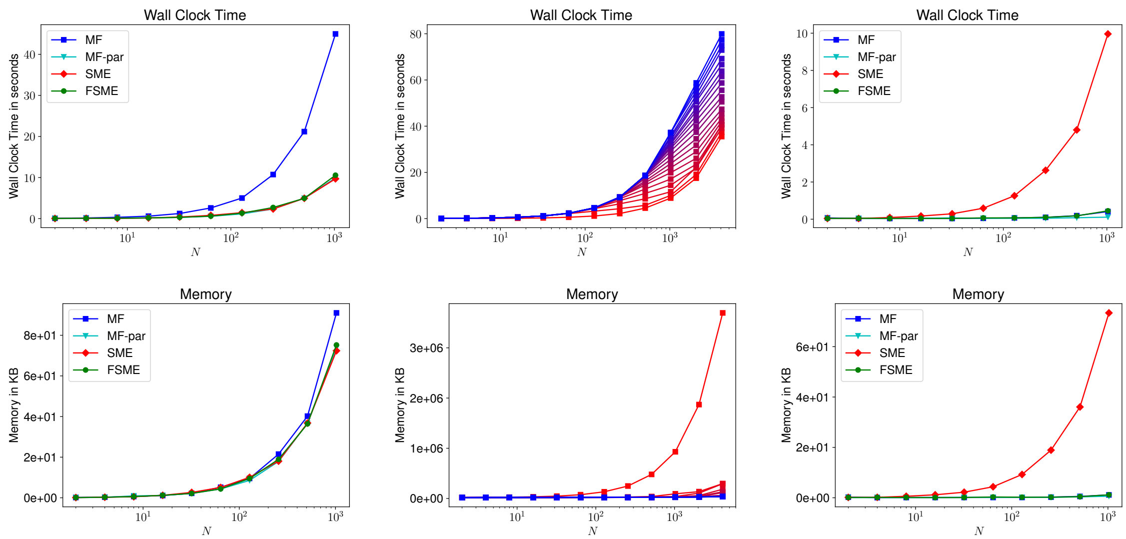
With this new representation, we can write the behavior of `RunBind` for SME.

```
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 3.B for the details.

3.8.5 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 3.C for the details.



(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 3.6: Time and memory consumption for different micro-benchmarks

3.9 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 $l_1, \ldots, l_n \in \mathit{Lattice}$. Let $s_i = \langle l_i \mathrel{?} \ldots : \ldots \rangle$ 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.

```haskell
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 3.6a 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 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 3.6a 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 3.6b 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 3.6c 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.

3.10 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].

3.10.1 Labeling policy

File owners specifies how information can be shared with different plugins. Initially, every file in `User`'s folders are labeled as $\{\langle \texttt{User}, \texttt{User} \rangle\}$, 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 $\{\langle \bot, \texttt{Plugin} \rangle\}$—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 $\langle \mathsf{User}, \mathsf{User} \vee \mathsf{Comments} \rangle$. 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 $\langle \bot, \mathsf{Tarball} \rangle$ to indicate that the plugin may have influenced the contents of the files, i.e., the tarball gets the label $(\bigsqcup l_f) \sqcup \langle \bot, \mathsf{Tarball} \rangle$.

▶ 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 $l_f \sqcup \langle \bot, \mathsf{Checksum} \rangle$. 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.

3.10.2 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 3.7 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.

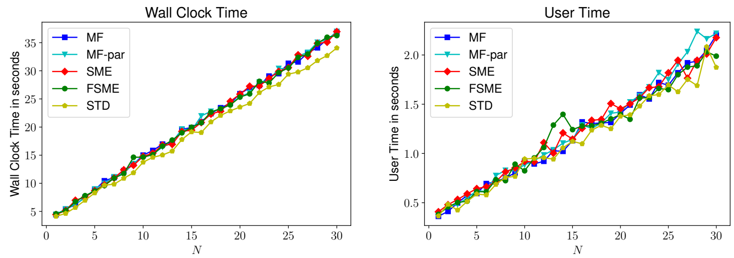


Figure 3.7: Time for different executors in the Tarball benchmark, where N is the number of files.

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 3.9.

The performance is dominated by network overheads. For this reason it is important that the safe executors do not introduce large numbers of sequential requests. The code under test in Figure 3.7 does not display such weakness. It is possible to construct programs similar to the first benchmark in Section 3.9 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 3.9. 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.

3.11 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 [17] 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 [8] and their implementations, e.g., FlowFox [14]. The implementation accompanying [8] 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 [57, 43], 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 [7] 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. [41] 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 [7]) and spawns multi-executions on-demand while [41] 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 [31, 44, 54, 2], dynamically [49], or as a combination of both [16, 9]. 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.

3.12 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.

Bibliography

[1] Sheldon B. Akers. “Binary decision diagrams”. In: *IEEE Transactions on computers* 6 (1978), pp. 509–516.

[2] Maximilian Algehed and Alejandro Russo. “Encoding DCC in Haskell”. In: *Proc. of the 2017 Workshop on Programming Languages and Analysis for Security*. PLAS ’17. ACM, 2017.

[3] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. “Predictive black-box mitigation of timing channels”. In: *Proc. of the 17th ACM conference on Computer and Communications Security*. ACM, 2010.

[4] Thomas H. Austin and Cormac Flanagan. “Multiple facets for dynamic information flow”. In: *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*. 2012, pp. 165–178.

[5] Thomas H. Austin and Cormac Flanagan. “Multiple facets for dynamic information flow”. In: *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’12. ACM, 2012.

[6] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. “Secure multi-execution through static program transformation”. In: *Formal Techniques for Distributed Systems (FMOODS/FORTE 2012)*. 2012.

[7] Nataliia Bielova and Tamara Rezk. “Spot the Difference: Secure Multi-execution and Multiple Facets”. In: *European Symposium on Research in Computer Security*. 2016, pp. 501–519.

[8] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. “Reactive non-interference for a browser model”. In: *Proceedings of the 5th International Conference on Network and System Security (NSS 2011)*, Sept. 2011.

[9] P. Buiras, D. Vytiniotis, and A. Russo. “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, 2015.

[10] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. “Preventing Information Leaks through Shadow Executions”. In: *Proc. of the Annual Computer Security Applications Conference*. ACSAC ’08. IEEE Computer Society, 2008.

[11] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. “Nonmalleable Information Flow Control”. In: *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*. 2017, pp. 1875–1891.

[12] Koen Claessen. “Parallel Parsing Processes”. In: *Journal of Funcitonal Programming* 14.6 (2004), pp. 741–757.

[13] Maximiliano Cristiá and Pablo Mata. “Runtime Enforcement of Noninterference by Duplicating Processes and their Memories”. In: *Workshop de Seguridad Informática WSEGI 2009, Argentina*. 38 JAIIO. 2009.

[14] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. “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. New York, NY, USA: ACM, 2012.

[15] Dorothy E. Denning and Peter J. Denning. “Certification of Programs for Secure Information Flow”. In: *Commun. ACM* 20.7 (July 1977), pp. 504–513.

[16] D. Devriese and F. Piessens. “Information flow enforcement in monadic libraries”. In: *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI ’11)*. ACM, 2011.

[17] D. Devriese and F. Piessens. “Noninterference through Secure Multi-execution”. In: *Proc. of the 2010 IEEE Symposium on Security and Privacy*. SP ’10. IEEE Computer Society, 2010.

[18] Dropbox. *Dropbox*. `https://www.dropbox.com`.

[19] Dropbox. *Dropbox HTTP API*. `https://www.dropbox.com/developers/documentation/http/overview`.

[20] Niklas Eén and Niklas Sörensson. “An extensible SAT-solver”. In: *International conference on theory and applications of satisfiability testing*. Springer. 2003, pp. 502–518.

[21] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. “Labels and event processes in the Asbestos operating system”. In: *Proc. of the twentieth ACM symp. on Operating systems principles*. SOSP ’05. ACM, 2005.

[22] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. “Hails: Protecting Data Privacy in Untrusted Web Applications”. In: *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 2012.

[23] J.A. Goguen and J. Meseguer. “Security policies and security models”. In: *Proc of IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1982.

[24] M. Jaskelioff and A. Russo. “Secure multi-execution in Haskell”. In: *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*. LNCS. Springer-Verlag, June 2011.

[25] Mark P Jones and Luc Duponcheel. *Composing monads*. Tech. rep. Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale University, 1993.

[26] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. “Simple unification-based type inference for GADTs”. In: *Proc. of the ACM SIGPLAN International Conf. on Functional Programming, ICFP*. 2006.

[27] Simon Peyton Jones, ed. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge, England: Cambridge University Press, 2003.

[28] V. Kashyap, B. Wiedermann, and B. Hardekopf. “Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach”. In: *Proc. of IEEE Symposium on Sec. and Privacy*. IEEE, 2011.

[29] Maxwell N. Krohn, Alexander Yip, Micah Z. Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Tappan Morris. “Information flow control for standard OS abstractions”. In: *Proc. of the 21st ACM Symposium on Operating Systems Principles*. 2007, pp. 321–334.

[30] B. W. Lampson. “A Note on the Confinement Problem”. In: *Communications of the ACM* 16.10 (Oct. 1973).

[31] P. Li and S. Zdancewic. “Arrows for secure information flow”. In: *Theoretical Computer Science* 411.19 (2010), pp. 1974–1994.

[32] P. Li and S. Zdancewic. “Encoding Information Flow in Haskell”. In: *Proc. of the IEEE Workshop on Computer Security Foundations (CSFW ’06)*. IEEE Computer Society, 2006.

[33] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. “Fabric: A Platform for Secure Distributed Computation and Storage”. In: *Proc. of the ACM SIGOPS Symposium on Operating Systems Principles*. ACM, 2009.

[34] B. Montagu, B.C. Pierce, and R. Pollack. “A Theory of Information-Flow Labels”. In: *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*. 2013.

[35] Scott Moore, Aslan Askarov, and Stephen Chong. “Precise enforcement of progress-sensitive security”. In: *the ACM Conference on Computer and Communications Security, CCS’12*. 2012.

[36] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. "seL4: From General Purpose to a Proof of Information Flow Enforcement". In: *2012 IEEE Symposium on Security and Privacy* 0 (2013).

[37] Andrew C Myers. "JFlow: Practical mostly-static information flow control". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1999, pp. 228–241.

[38] Andrew C Myers and Barbara Liskov. "Protecting privacy using the decentralized label model". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9.4 (2000), pp. 410–442.

[39] M. Ngo, F. Piessens, and T. Rezk. "Impossibility of Precise and Sound Termination-Sensitive Security Enforcements". In: *IEEE Symposium on Security and Privacy (SP)*. 2018.

[40] Minh Ngo, Fabio Massacci, and Olga Gadyatskaya. "MAP-REDUCE Runtime Enforcement of Information Flow Policies". In: *CoRR* (2013).

[41] Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. "A better facet of dynamic information flow control". In: *The Web Conference. Research track: Security and privacy of the Web. (WWW'18)*. 2018.

[42] S. Peyton Jones, A. Gordon, and S. Finne. "Concurrent Haskell". In: *Proc. of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96)*. ACM, 1996.

[43] Willard Rafnsson and Andrei Sabelfeld. "Secure multi-execution: fine-grained, declassification-aware, and transparent". Submitted. Feb. 2013.

[44] A. Russo, K. Claessen, and J. Hughes. "A library for light-weight information-flow security in Haskell". In: *Proc. ACM SIGPLAN symposium on Haskell (HASKELL '08)*. ACM, Sept. 2008.

[45] Alejandro Russo and Andrei Sabelfeld. "Dynamic vs. Static Flow-Sensitive Security Analysis". In: *Proc. of the 2010 23rd IEEE Computer Security Foundations Symp.* CSF '10. IEEE Computer Society, 2010, pp. 186–199.

[46] Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. "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*. 2016.

[47] Daniel Schoepe, Musard Balliu, Frank Piessens, and Andrei Sabelfeld. "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*. 2016.

[48] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. "Disjunction Category Labels". In: *Proc. of the Nordic Conference on Information Security Technology for Applications (NORDSEC '11)*. Springer-Verlag, 2011.

[49] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. "Flexible Dynamic Information Flow Control in Haskell". In: *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*. 2011.

[50] Deian Stefan, Alejandro Russo, David Mazières, and John C Mitchell. "Disjunction category labels". In: *Nordic conference on secure IT systems*. Springer. 2011.

[51] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. "Protecting Users by Confining JavaScript with COWL". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Oct. 2014.

[52] Wouter Swierstra and Thorsten Altenkirch. "Beauty in the Beast: A Functional Semantics of the Awkward Squad". In: *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell*. Freiburg, Germany, 2007, pp. 25–36.

[53] Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. "Stateful declassification policies for event-driven programs". In: *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE, July 2014.

[54] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. "MAC A Verified Static Information-Flow Control Library". In: *Journal of Logical and Algebraic Methods in Programming* (2017).

[55] Philip Wadler. "Monads for functional programming". In: *Program design calculi*. Springer, 1993, pp. 233–264.

[56] Philip Wadler. "Monads for functional programming". In: *International School on Advanced Functional Programming*. Springer. 1995, pp. 24–52.

[57] Dante Zanarini, Mauro Jaskelioff, and Alejandro Russo. "Precise Enforcement of Confidentiality for Reactive Systems." In: *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE, 2013, pp. 18–32.

[58] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. "Making information flow explicit in HiStar". In: *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation*. USENIX, 2006.

[59] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. “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*. 2015, pp. 503–516.

Appendix 3.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 3.8 and 3.9 along with the semantics in Figures 3.10 and 3.11 and the type system in Figure 3.12. Figure 3.13 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 $\mathsf{measure}(l, \sigma)$, 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. $\square$

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. $\square$

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 $\sigma_2 \approx_l \sigma_1$.* $\square$

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. $\square$

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. $\square$

Appendix 3.B Implementation

DC labels, see section 3.7.1, are represented as a Haskell data type:

```
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:

```haskell
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.

```haskell
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:

```haskell
  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 3.8.4. Based on these continuation-based primitives, we implement non-continuation-based wrappers that have the expected type matching Figure 3.12.

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

```
(>>=) :: 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 3.7 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:

```haskell
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 3.3.

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 3.3 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 ()`.

Appendix 3.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.

```haskell
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)
```

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

Figure 3.8: Full syntax (part I).

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

Figure 3.9: Full syntax (part II).

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

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

$$\begin{array}{llll}
((\lambda x.t_1)\ t_2, M, P, I, O) & \longrightarrow_{pc} (t_1[x := t_2], M, P, I, O) & & [\textsc{F-App}] \\
(n_1 + n_2, M, P, I, O) & \longrightarrow_{pc} (n, M, P, I, O) & \text{if } n = n_1 + n_2 & [\textsc{F-Plus}] \\
(\mathsf{if}\ n\ t_1\ t_2, M, P, I, O) & \longrightarrow_{pc} (t_1, M, P, I, O) & \text{if } n \neq 0 & [\textsc{F-If-1}] \\
(\mathsf{if}\ n\ t_1\ t_2, M, P, I, O) & \longrightarrow_{pc} (t_2, M, P, I, O) & \text{if } n = 0 & [\textsc{F-If-2}] \\
((\mathsf{return}\ t_1) \gg= t_2, M, P, I, O) & \longrightarrow_{pc} (t_2\ t_1, M, P, I, O) & & [\textsc{F-Bind-FIO}] \\
(\mathsf{run}\ (\mathsf{raw}\ t), M, P, I, O) & \longrightarrow_{pc} (t \gg= \lambda x.\mathsf{return}\ (\mathsf{raw}\ x), M, P, I, O) & & [\textsc{F-Run-Raw}] \\
(\mathsf{run}\ \langle k\,?\,t_1 : t_2\rangle, M, P, I, O) & \longrightarrow_{pc} \begin{cases} (\mathsf{run}\ t_1, M, P, I, O) & \text{if } \mathit{views}(pc \cup \{\overline{k}\}) = \emptyset \\ (\mathsf{run}\ t_2, M, P, I, O) & \text{if } \mathit{views}(pc \cup \{k\}) = \emptyset \\ (\langle\!\langle k\,?\,\mathsf{run}\ t_1 : \mathsf{run}\ t_2\rangle\!\rangle, M, P, I, O) & \text{otherwise.} \end{cases} & & \begin{array}{l}[\textsc{F-Run-Facet-1}] \\ [\textsc{F-Run-Facet-2}] \\ [\textsc{F-Run-Facet-3}]\end{array} \\
(\mathsf{run}\ (\mathsf{bind}\ (\mathsf{raw}\ t_1)\ t_2), M, P, I, O) & \longrightarrow_{pc} (\mathsf{run}\ (t_2\ t_1), M, P, I, O) & & [\textsc{F-Bind-Fac-1}] \\
(\mathsf{run}\ (\mathsf{bind}\ \langle k\,?\,V_1 : V_2\rangle\ t), M, P, I, O) & \longrightarrow_{pc} (\mathsf{run}\ \langle k\,?\,\mathsf{bind}\ V_1\ t : \mathsf{bind}\ V_2\ t\rangle, M, P, I, O) & & [\textsc{F-Bind-Fac-2}] \\
(\mathsf{run}\ (\mathsf{bind}\ (\mathsf{bind}\ t_1\ t_2)\ t_3), M, P, I, O) & \longrightarrow_{pc} (\mathsf{run}\ (\mathsf{bind}\ t_1\ (\lambda x.\mathsf{bind}\ (t_2\ x), M, P, I, O)\ t_3)) & & [\textsc{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) & & [\textsc{F-Fork-Continuation}] \\
(\langle\!\langle k\,?\,\mathsf{return}\ V_1 : \mathsf{return}\ V_2\rangle\!\rangle, M, P, I, O) & \longrightarrow_{pc} (\mathsf{return}\ \langle k\,?\,V_1 : V_2\rangle, M, P, I, O) & & [\textsc{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) & \text{if } \overline{k} \notin pc \text{ and } t_1 \longrightarrow_{pc \cup \{k\}} t_1' & [\textsc{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) & \text{if } k \notin pc \text{ and } t_2 \longrightarrow_{pc \cup \{\overline{k}\}} t_2' & [\textsc{F-Thread-2}]
\end{array}$$

Figure 3.10: Full semantics (part 1).

$$
\begin{array}{llll}
(\textsf{new}\ V, M, P, I, O) & \longrightarrow_{pc} (\textsf{return}\ a, M[a := \langle\!\langle pc\,?\,V : \textsf{raw}\ 0\rangle\!\rangle], P, I, O) & \text{if } a \notin \mathrm{dom}(M) & [\textsc{f-new}] \\
(\textsf{read}\ a, M, P, I, O) & \longrightarrow_{pc} (\textsf{return}\ M(a), M, P, I, O) & & [\textsc{f-read}] \\
(\textsf{write}\ a\ V, M, P, I, O) & \longrightarrow_{pc} (\textsf{return}\ V, M', P, I, O) & \text{if } M' = M[a := \langle\!\langle pc\,?\,V : M(a)\rangle\!\rangle] & [\textsc{f-write}] \\
(\textsf{get}\ i, M, P, I, O) & \longrightarrow_{pc} (\textsf{return}\ \langle l_i\,?\,V : \textsf{raw}\ 0\rangle, M, P[i := p'], I, O) & \text{if } (V, p') = \mathrm{fac\_get}(pc, P(i), I(i)) & [\textsc{f-get}] \\
(\textsf{put}\ o\ n, M, P, I, O) & \longrightarrow_{pc} \begin{cases} (\textsf{return}\ n, M, P, I, O[o := O(o) +\!\!+\, n]) \\ (\textsf{return}\ n, M, P, I, O) \end{cases} & \begin{array}{l} \text{if } l_o \in \mathit{views}(pc) \\ \text{if } l_o \notin \mathit{views}(pc) \end{array} & \begin{array}{l} [\textsc{f-put-1}] \\ [\textsc{f-put-2}] \end{array}
\end{array}
$$

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

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

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

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

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

$$\frac{n \geq \mathrm{length}(I(i))}{(\textsf{raw}\ (-1), \langle\!\langle pc\,?\,n + 1 : n\rangle\!\rangle) = \mathrm{fac\_get}(pc, n, ns)} \quad [\textsc{r-raw-eof}]$$

Figure 3.11: Full semantics (part 2).

$$\textsc{[t-var]}\quad \Gamma, \Delta \vdash x :: \Gamma(x)$$

$$\textsc{[t-lam]}\quad \frac{\Gamma[x := T_1], \Delta \vdash t_2 :: T_2}{\Gamma, \Delta \vdash \lambda x.t_2 :: T_1 \rightarrow T_2}$$

$$\textsc{[t-app]}\quad \frac{\Gamma, \Delta \vdash t_0 :: T_1 \rightarrow T_2 \qquad \Gamma, \Delta \vdash t_1 :: T_1}{\Gamma, \Delta \vdash t_0\ t_1 :: T_2}$$

$$\textsc{[t-addr]}\quad \frac{}{\Gamma, \Delta \vdash a :: \Delta(a)}$$

$$\textsc{[t-int]}\quad \frac{}{\Gamma, \Delta \vdash n :: \mathsf{Int}}$$

$$\textsc{[t-plus]}\quad \frac{\Gamma, \Delta \vdash t_1 :: \mathsf{Int} \qquad \Gamma, \Delta \vdash t_2 :: \mathsf{Int}}{\Gamma, \Delta \vdash t_1 + t_2 :: \mathsf{Int}}$$

$$\textsc{[t-if]}\quad \frac{\Gamma, \Delta \vdash t_0 :: \mathsf{Int} \qquad \Gamma, \Delta \vdash t_1 :: T \qquad \Gamma, \Delta \vdash t_2 :: T}{\Gamma, \Delta \vdash \mathsf{if}\ t_0\ t_1\ t_2 :: T}$$

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

$$\textsc{[t-facet]}\quad \frac{\Gamma, \Delta \vdash V_1 :: \mathsf{Fac}\ T \qquad \Gamma, \Delta \vdash V_2 :: \mathsf{Fac}\ T}{\Gamma, \Delta \vdash \langle k \,?\, V_1 : V_2 \rangle :: \mathsf{Fac}\ T}$$

$$\textsc{[t-bind-fac]}\quad \frac{\Gamma, \Delta \vdash t_1 :: \mathsf{Fac}\ T_1 \qquad \Gamma, \Delta \vdash t_2 :: T_1 \rightarrow \mathsf{Fac}\ T_2}{\Gamma, \Delta \vdash \mathsf{bind}\ t_1\ t_2 :: \mathsf{Fac}\ T_2}$$

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

$$\textsc{[t-bind-fio]}\quad \frac{\Gamma, \Delta \vdash t_1 :: \mathsf{FIO}\ T_1 \qquad \Gamma, \Delta \vdash t_2 :: T_1 \rightarrow \mathsf{FIO}\ T_2}{\Gamma, \Delta \vdash t_1 \gg\!= t_2 :: \mathsf{FIO}\ T_2}$$

$$\textsc{[t-new]}\quad \frac{\Gamma, \Delta \vdash t :: \mathsf{Fac}\ T}{\Gamma, \Delta \vdash \mathsf{new}\ t :: \mathsf{FIO}\ (\mathsf{FIORef}\ T)}$$

$$\textsc{[t-read]}\quad \frac{\Gamma, \Delta \vdash t :: \mathsf{FIORef}\ T}{\Gamma, \Delta \vdash \mathsf{read}\ t :: \mathsf{FIO}\ (\mathsf{Fac}\ T)}$$

$$\textsc{[t-write]}\quad \frac{\Gamma, \Delta \vdash t_1 :: \mathsf{FIORef}\ T \qquad \Gamma, \Delta \vdash t_2 :: \mathsf{Fac}\ T}{\Gamma, \Delta \vdash \mathsf{write}\ t_1\ t_2 :: \mathsf{FIO}\ (\mathsf{Fac}\ T)}$$

$$\textsc{[t-get]}\quad \frac{}{\Gamma, \Delta \vdash \mathsf{get}\ i :: \mathsf{FIO}\ (\mathsf{Fac}\ \mathsf{Int})}$$

$$\textsc{[t-put]}\quad \frac{\Gamma, \Delta \vdash t :: \mathsf{Int}}{\Gamma, \Delta \vdash \mathsf{put}\ o\ t :: \mathsf{FIO}\ \mathsf{Int}}$$

$$\textsc{[t-run]}\quad \frac{\Gamma, \Delta \vdash t :: \mathsf{Fac}\ (\mathsf{FIO}\ T)}{\Gamma, \Delta \vdash \mathsf{run}\ t :: \mathsf{FIO}\ (\mathsf{Fac}\ T)}$$

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

Figure 3.12: 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]

$$
\begin{array}{llll}
(\mathsf{new}\ F, M, P, I, O) & \xrightarrow{\text{std}} (\mathsf{return}\ a, M[a := F], P, I, O) & \text{if } a \notin \mathrm{dom}(M) & [\text{S-NEW}] \\
(\mathsf{write}\ a\ F, M, P, I, O) & \xrightarrow{\text{std}} (\mathsf{return}\ F, M[a := F], P, I, O) & & [\text{S-WRITE}] \\
(\mathsf{get}\ i, M, P, I, O) & \xrightarrow{\text{std}} (\mathsf{return}\ (\mathsf{raw}\ n), M, P[i := P(i) + 1], I, O) & \text{if } n = I(i)_{P(i)} & [\text{S-GET}] \\
(\mathsf{get}\ i, M, P, I, O) & \xrightarrow{\text{std}} (\mathsf{return}\ (\mathsf{raw}\ (-1)), M, P, I, O) & \text{if } P(i) \geq \mathrm{length}(I(i)) & [\text{S-GET-EOF}] \\
(\mathsf{put}\ o\ n, M, P, I, O) & \xrightarrow{\text{std}} (\mathsf{return}\ n, M, P, I, O[o := O(o) +\!\!+\, n]) & & [\text{S-PUT}]
\end{array}
$$

Figure 3.13: Full standard semantics.

Chapter 4

FacetBook

4.1 Research questions

An important goal for achieving security is to minimize the size of the *trusted computing base* (TCB), which is the portion of code that must be carefully audited for security [6]. (We refer to the remaining code as the *untrusted computing base* (UCB).)

Our hypothesis is that faceted execution (as implemented by the FIO library) makes it *easier* to minimize the size of the TCB in realistic applications. In particular, we have two research questions:

1. Does FIO help minimize TCB size when coding a secure application?
2. Does FIO help minimize TCB size when changing an existing application to meet new requirements?

Our experimental design to investigate these questions is as follows:

- Create Design V1 for a prototype application called FacetBook.
- Create Implementation V1-FIO using FIO, minimizing the TCB.

Version	Lines of code FIO	TCB	UCB	Total application-specific code
V1-FIO	108	99	352	451
V2-FIO	108	99	360	459
V1-NoFIO	0	118	295	413
V2-NoFIO	0	419	0	419
V2-NoFIO-minTCB	0	128	298	426

Table 4.1: The number of lines of code in each version of FacetBook. The emphasized entries are useful for quantifying security.

- Create Implementation V1-NoFIO without FIO, minimizing the TCB.
- Measure TCB size of the two implementations.
- Create Design V2 by making a small change to Design V1.
- Create Implementation V2-FIO by modifying V1-FIO.
- Create Implementation V2-NoFIO by modifying V1-NoFIO.
- Create Implementation V2-NoFIO-minTCB from V2-NoFIO by minimizing the TCB.
- Quantify the effect on security by comparing the increase in TCB size when going from V1 to V2.
- Quantify the *ease* of achieving security by comparing the number of lines of code changed when minimizing the TCB size in V2.

Table 4.1 shows the number of lines of code for each version. Table 4.2 shows the number of edit actions required to change each version to the next. The full source code is available at `https://github.com/tommy-schmitz/facetbook`. In the sections below, we discuss these results.

Version	Changes (measured in lines of code) Modified	Moved	Inserted	Deleted
V1-FIO				
V2-FIO	1	0	8	0
V1-NoFIO				
V2-NoFIO	2	3	6	0
V2-NoFIO-minTCB	4	6	7	0

Table 4.2: The differences between each version of FacetBook. Each row in the table lists the differences from the version in the row above it. The emphasized entries are useful for quantifying *ease* of achieving security.

4.2 Design V1: FacetBook

4.2.1 Overview

FacetBook is a prototype social networking website. Users can submit *posts* (pieces of text that are visible to a subset of other users of the website) and can play *Tic Tac Toe* with other users, which is a simple and well-known game that children commonly play using pencil and paper. (In this case, the game is played using two computers equipped with web browsers and mouse pointer devices.)

For the purposes of our experiment, the "posts" feature exists so that FacetBook has a rich TCB (because the information flow requirements are complex), while the "Tic Tac Toe" feature exists so that it has a rich UCB (because the information flow requirements are simple, but the other computations are relatively complex).

4.2.2 User interface

Figure 4.1 illustrates the structure of FacetBook's webpages.

The `login` page allows typing a username and clicking the "Submit" button to go to the `dashboard` page. For simplicity, authentication always succeeds with no password required—sophisticated authentication machinery would remain constant

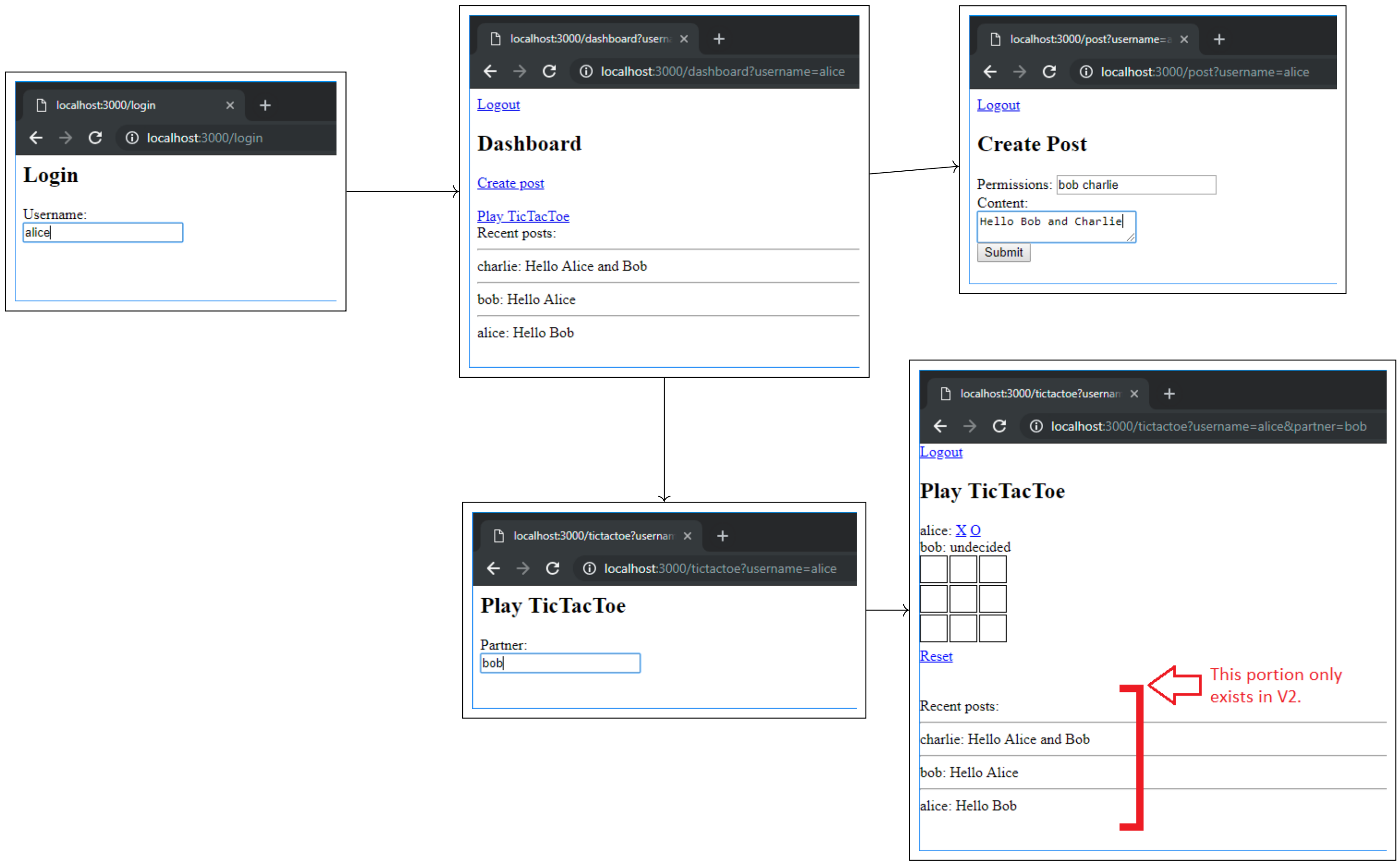


Figure 4.1: Screenshots of FacetBook.

throughout all six versions of FacetBook, and so would simply add a constant number of lines of code to the TCB. Unlike other work [3], we make no attempt here to remove authentication code (i.e. password-checking code) from the TCB.

The `dashboard` page shows a list of 20 recent posts created by users of FacetBook. The list comes from the server's database of all posts, but contains only those that the currently authenticated user is permitted to view. The page also has two links: one going to the `post` page and one to the `tictactoe` page.

The `post` page allows users to compose posts, and so has a form with two fields: the `permissions` field expects a space-delimited list of usernames indicating who is allowed to see the post, and the `content` field expects any string. Upon clicking "Submit," the form is submitted via HTTP POST protocol to the `/post` endpoint, and the server saves the submitted data in a database.

The `tictactoe` page initially shows a form with a single field `partner` expecting the username of the person with whom to play Tic Tac Toe. Upon clicking "Submit," the Tic Tac Toe board and its controls appear on the page. If this pair of users (the currently authenticated user and the specified `partner`) has never played Tic Tac Toe together before, then the server begins by adding a fresh game to the list of ongoing games in the database. Then the server retrieves the game (whether freshly-created or pre-existing) from the database and renders it into HTML when serving the page. Thereafter, if the user clicks on the controls of the game, then the web browser sends a request (using Javascript) specifying what action to take, and the server updates the game in the database as appropriate. Then the server replies with updated HTML, which replaces (using Javascript) the display in the browser.

4.2.3 Information security

In FacetBook, restricted information arrives via HTTP POST protocol at the `/post` endpoint. This endpoint is how users express their information flow desires, namely that only the users specified in the `permissions` field can know about this post and its content (in the `content` field).

The restricted output channel is the server's response to any incoming HTTP request—unless that request contains credentials of an appropriate user. In FacetBook, requests specify credentials in the HTTP GET parameter `username` (rather than in a cookie).

These information security specifications implicitly define a specific attacker model that considers some potential attacks and ignores others. Notably, our model ignores the correctness of the user interface, which is important because we intend to place the client code in the UCB. If an attacker controls the UCB, then the attacker could interfere with the creation of the POST request by, for instance, adding an extra entry to the `permissions` field before submitting the POST request. In the design of FacetBook, we explicitly ignore such an attack and choose instead to assume that the POST parameters received at the server correctly reflect the user's intentions.

4.3 FIO library

Figure 4.2 shows the interface of the FIO library. The main difference from Chapter 2 is that this code now supports using an arbitrary security lattice [5], rather than specifically a power set security lattice over the set of `String`s. As a result, the type constructors `Fac`, `FIORef`, `FIO`, and `PC` now take an additional type parameter for specifying the security lattice. The corresponding `Lattice`

```haskell
class Lattice a where
  leq :: a -> a -> Bool
  lub :: a -> a -> a
  bot :: a

data Fac l a where
  Undefined :: Fac l a
  Raw       :: a -> Fac l a
  Fac       :: l -> Fac l a -> Fac l a -> Fac l a
  BindFac   :: Fac l a -> (a -> Fac l b) -> Fac l b

data FIORef l a = FIORef (IORef (Fac l a))

data FIO l a where
  Return  :: a -> FIO l a
  BindFIO :: FIO l a -> (a -> FIO l b) -> FIO l b
  Swap    :: Fac l (FIO l a) -> FIO l (Fac l a)
  IO      :: l -> IO a -> FIO l a
  New     :: a -> FIO l (FIORef l a)
  Read    :: FIORef l a -> FIO l (Fac l a)
  Write   :: FIORef l a -> Fac l a -> FIO l ()

data PC l = PC [l] [l]

runFIO :: Lattice l => PC l -> FIO l a -> IO a
```

Figure 4.2: The interface of the FIO library in all versions of FacetBook.

type class (lines 1 through 4) specifies the methods (`leq`, `lub`, and `bot`) that the security lattice must implement.

In addition to the extra type parameter, we change slightly the representation of the `PC` datatype, so now `PC ks1 ks2` denotes the set of lattice elements k such that

- $k' \sqsubseteq k$ for all $k' \in$ `ks1`, and
- $k' \not\sqsubseteq k$ for all $k' \in$ `ks2`.

The main library function is `runFIO`, which runs an `FIO` computation safely, namely by respecting the information flow requirements specified by any faceted values used in the computation. The computation bifurcates if necessary.

The FIO library contains 108 lines of code. (Only the interface is shown in Figure 4.2.)

4.4 V1-FIO

FacetBook V1-FIO is the initial version of the code, which implements Design V1, uses the FIO library, and is organized so as to minimize the size of the TCB.

4.4.1 Tour of TCB

Security lattice

The lattice of security labels is defined in Figure 4.3. The label `Bot` is for public data; the label `Whitelist` *users* is for data visible only to the users listed in the list *users*. The datatype `Label` forms a lattice, as evidenced by the type class instance `Lattice Label` and its three methods `leq`, `lub`, and `bot`.

```haskell
data Label = Whitelist [User]
           | Bot
instance Lattice Label where
  leq Bot _ = True
  leq _ Bot = False
  leq (Whitelist us1) (Whitelist us2) =
    let subset xs ys = all (\x -> x `elem` ys) xs  in
    us2 `subset` us1
  lub Bot k = k
  lub k Bot = k
  lub (Whitelist us1) (Whitelist us2) =
    Whitelist (List.intersect us1 us2)
  bot = Bot
```

Figure 4.3: The code for the `Label` datatype in all versions of FacetBook.

```haskell
type Post     = String
data FList a  = Nil
              | Cons a (Fac Label (FList a))
type PostList = FList Post
type Database = (FIORef Label PostList, FIORef Label [TicTacToe])
```

Figure 4.4: The code for the `FList` datatype and associated type definitions in V1-FIO.

Database format

The database format is defined in Figure 4.4. For simplicity, we keep the database in memory rather than on disk (unlike other work on using faceted values with databases [7, 2]). The `Database` type is a pair of two mutable references (`FIORef`s), one for holding the current list of posts and a second for holding the current list of ongoing Tic Tac Toe games. The `PostList` type makes use of a custom datatype `FList`, which is a singly-linked list datatype whose "next" pointer is always faceted. The `Post` type is simply an alias for Haskell's built-in `String`

```
44 main :: IO ()
45 main = do  --IO
46   database <- runFIO (Constraints [] []) $ do  --FIO
47     r1 <- New Nil
48     r2 <- New []
49     return (r1, r2)
50   let port = 3000
51   Warp.run port $ \request respond -> do  --IO
52     let (k1, k2) = policy request
53     let fio_respond = \x -> IO k2 $ do  --IO
54          respond x
55          return ()
56     let faceted_request = Fac k1 (Raw request) Undefined
57     runFIO (Constraints [] []) $
58         UCB.handle_request faceted_request database fio_respond
59     return ResponseReceived
```

Figure 4.5: The code for the `main` function in V1-FIO.

type.

The faceted values in an `FList` potentially allow the "list" to be structured actually as a tree with branching factor 2. However, in practice, when appending to the list, each facet shares a suffix with the opposing facet, so in fact the structure in memory forms a directed acyclic graph whose size is linear in the total number of posts.

Main function

Figure 4.5 shows the `main` function. Its purpose is to start the web server and set up appropriate security sandboxes before handling each request.

Line 47 initializes the database with an empty list of posts, and line 48 initializes it with an empty list of Tic Tac Toe games. Line 51 creates a socket (using the Haskell library function `Warp.run`) for listening for incoming HTTP requests, which

```
60 policy :: WAI.Request -> (Label, Label)
61 policy request =
62   if WAI.pathInfo request == ["login"] then
63     (Bot, Bot)
64   else case check_credentials request of
65     Nothing ->
66       (Bot, Bot)
67     Just username -> case WAI.pathInfo request of
68       ["post"] ->
69         let permissions = get_parameter request "permissions"  in
70         let users = words permissions  in
71         if all valid_username users then
72           (Whitelist (username : users), Whitelist [username])
73         else
74           (Whitelist [username], Whitelist [username])
75       _ ->
76         (Bot, Whitelist [username])
```

Figure 4.6: The code for the `policy` function in V1-FIO.

are handled by the code on lines 52 through 59. Line 58 calls `UCB.handle_request`, which is outside the TCB; however, its inputs (`database`, `faceted_request`, and `fio_respond`) are all faceted appropriately, and its side effects are sandboxed appropriately by `runFIO (Constraints [] [])` on line 57.

Policy function

The function `policy` (called on line 52) computes the appropriate labels to use in FacetBook. Its code is shown in Figure 4.6. We parse the request to determine its meaning, and then we return two labels: one for the confidentiality of the request, and one for the label of the output channel for returning an HTTP response to the user.

Specifically, this policy assigns `Bot` for both labels (lines 63 and 66) when the

```haskell
{-# LANGUAGE OverloadedStrings #-}
module UCB where
import qualified Data.List as List
import Data.Monoid((<>))
import Data.String(fromString)
import qualified Data.ByteString.Lazy.Char8 as ByteString(intercalate)
import Network.HTTP.Types.Status(status200, status404)
import qualified Network.Wai as WAI(Request, pathInfo, ResponseLBS)
import Shared
import FIO(FIO(Read, Write, Swap), Fac(), FIORef)
```

Figure 4.7: The import statements for the `UCB` module in V1-FIO.

user is not logged in, which is the case when requesting the login page (line 62) or when lacking credentials on any other page (line 65). When the user has valid credentials, the HTTP response label is `Whitelist [username]` (lines 72, 74, and 76), indicating that the response can contain private information belonging to the authenticated user. For most pages, the confidentiality label on the request is `Bot` (line 76), which means that the request itself carries no sensitive information; however, on the `"post"` page, the label `Whitelist (username : users)` (line 72) indicates that the request is visible only to the users named in the `permissions` parameter of the request (and the currently authenticated user too). This label ensures that when the submitted post is written to the database, it will be faceted appropriately. The label `Whitelist [username]` on line 74 is used in case a client sends a malformed request where the `permissions` parameter contains invalid entries.

Import statements

The TCB includes the import statements at the top of each file. Primarily, we must verify that the `UCB` module imports (Figure 4.7) do not include `FIO(runFIO,`

```haskell
{-# LANGUAGE OverloadedStrings #-}
module Shared where
import Data.String(fromString)
import Data.ByteString.Char8(unpack)
import qualified Network.Wai as WAI(Request, queryString)
import qualified Data.List as List(intersect)
import FIO
```

Figure 4.8: The import statements for the `Shared` module in V1-FIO.

```haskell
{-# LANGUAGE OverloadedStrings #-}
module TCB where
import qualified Network.Wai.Handler.Warp as Warp(run)
import qualified Network.Wai as WAI(Request, pathInfo)
import Network.Wai.Internal(ResponseReceived(ResponseReceived))
import Shared
import FIO
import qualified UCB as UCB(handle_request)
```

Figure 4.9: The import statements for the `TCB` module in V1-FIO.

```haskell
check_credentials :: WAI.Request -> Maybe User
check_credentials request =
  let username = get_parameter request "username"  in
  if valid_username username  then  Just username
                              else  Nothing

get_parameter :: WAI.Request -> String -> String
get_parameter request key =
  case lookup (fromString key) (WAI.queryString request) of
    Just (Just value) -> unpack value
    _                 -> ""

valid_username :: String -> Bool
valid_username s =
  s /= ""  &&
  all (\c -> (c>='0' && c<='9') ||
             (c>='a' && c<='z') ||
             (c>='A' && c<='Z') ||
             c=='_'                ) s
```

Figure 4.10: The code for the helper functions in V1-FIO.

`FIO(IO), Fac(Raw, Fac, Undefined, BindFac))`. As a result, these import statements are actually part of the TCB.

The import statements in the `TCB` and `Shared` modules are also in the TCB, naturally, and help auditors determine which standard libraries must be trusted.

Helper functions

For completeness, we include the TCB's helper functions, which are shown in Figure 4.10. `check_credentials` is the password-checking function. It gets the username from the HTTP GET parameters. For simplicity, it always succeeds without any password. When no username is supplied, it returns `Nothing`, indicating invalid credentials. `get_parameter` extracts an HTTP GET parameter from

```
121 type Handler = Database -> (WAI.Response -> FIO ()) -> FIO ()
122 handle_request :: Fac Label WAI.Request -> Handler
123 handle_request faceted_request database respond = do  --FIO
124   Swap $ do  --Fac
125     request <- faceted_request
126     return $ do  --FIO
127       let handler = parse_request request
128       handler database respond
129   return ()
```

Figure 4.11: The code for the `handle_request` function in V1-FIO.

a request. `valid_username` checks that a string is non-empty and contains only letters, numbers, and underscores.

Summary

In summary, the TCB of FacetBook V1-FIO contains 99 lines: 41 in `TCB.hs`, 48 in `Shared.hs`, and 10 import statements in `UCB.hs`.

4.4.2 Tour of UCB

Handle-request function

The entry point to the UCB is `handle_request`, called on line 58 in `main`. Figure 4.11 shows its code. Its purpose is to "unfacet" the request (i.e. bifurcate if necessary, using `Swap` to do so), and then defer to the helper function `parse_request` and its return value `handler` to do the actual processing. At the call site (line 58 in `main`), the faceted request always has a specific shape, namely with `Undefined` in the low-security facet. As a result, the bifurcation at line 124 executes the high-security path like normal (with a changed PC), and then the low-security path is a no-op.

This code illustrates a typical interaction between the two monads `Fac` and `FIO`. Line 124 uses `Swap` to change the current monad from `FIO` to `Fac` to allow extracting `request` from `faceted_request` on line 125. Then line 126 uses `return` to change the current monad back from `Fac` to `FIO` to allow executing the action on line 128. By using two monads, we can delimit the scope of the bifurcation to be lines 125 to 128. The computations join back together at line 129.

Parse-request function

The `parse_request` function translates an incoming web request (of type `WAI.Request`, imported from Haskell's `WAI` library for web servers) into an appropriate action (of type `Handler`) to take in response to that request. Figure 4.12 shows its code. It duplicates some functionality (checking whether the request is for the "login" page, checking credentials, etc.) from the `policy` function in the TCB, so it would be reasonable to refactor the code to reduce redundancy. We decided against doing so because the function names `policy` and `parse_request` document their purposes well, whereas it is nontrivial to choose a good name for the newly created functions and intermediate datatypes in the refactored version; in any case, the amount of duplicated code is small.

Handler functions

The `parse_request` function delegates functionality to eight other functions called `Handler`s, namely:

- `login`: sends to the client a login page.
- `authentication_failed`: sends a page to redirect back to the login page.
- `do_create_post` *username content users*: inserts a new post into the database and redirects to the dashboard page.

```
parse_request :: WAI.Request -> Handler
parse_request request =
  if WAI.pathInfo request == ["login"] then
    login
  else case check_credentials request of
    Nothing ->
      authentication_failed
    Just username -> case WAI.pathInfo request of
      ["post"] ->
        let content = get_parameter request "content"  in
        let permissions = get_parameter request "permissions"  in
        let users = words permissions  in
        if content /= "" && all valid_username users then
          do_create_post username content users
        else
          compose_post username
      ["dashboard"] ->
        dashboard username
      ["tictactoe"] ->
        let partner = get_parameter request "partner"  in
        if valid_username partner then
          let action = get_parameter request "action"  in
          tictactoe_play username partner action
        else
          tictactoe_select_partner username
      _ ->
        not_found
```

Figure 4.12: The code for the `parse_request` function in V1-FIO.

- `compose_post` *username*: sends to the client a page displaying a form in which the user can compose a new post.
- `dashboard` *username*: sends a page displaying a few links to other pages, as well as a list of recent posts.
- `tictactoe_play` *username partner action*: updates a Tic Tac Toe game in the database (if necessary) and sends to the client a page displaying the current state of the game.
- `tictactoe_select_partner` *username*: sends to the client a page prompting the user to type the name of another user.
- `not_found`: sends a page with "404 bad request" on it.

The `Handler` type is defined on line 121

```
type Handler = Database -> (WAI.Response -> FIO ()) -> FIO ()
```

and its definition means that it takes as input the database reference cells (type `Database` defined on line 43) and a callback function (of type `WAI.Response -> FIO ()`) whose behavior when called is to send an HTTP response to the user's web browser. Thanks to the code in `main`, the database contents are secure (inside `FIORefs`) and the response callback function will not work if the current control flow has been influenced by information that the user should not know (in that case, the callback would behave as a no-op).

Summary

The UCB of FacetBook V1-FIO contains 352 lines: 362 in `UCB.hs` minus the 10 import statements at the top of the file, which are actually part of the TCB.

4.5 V1-NoFIO

FacetBook V1-NoFIO is the next version of the code, which implements Design V1, does not use the FIO library, and is organized so as to minimize the size of the TCB. In this section, we highlight the differences between V1-FIO and V1-NoFIO.

4.5.1 Removing undesirable dependence on FIO

The FIO library is unnecessary in this version of FacetBook, so we can simplify the code by removing dependence on FIO.

First, and most obviously, we remove the file `FIO.hs` from the codebase. As a result, we remove all calls to `Swap`, which is now unnecessary due to the lack of faceted values. Similarly, we replace uses of `New`, `Read`, and `Write` with uses of `newIORef`, `readIORef`, and `writeIORef`, respectively. Continuing likewise, we remove the `FList` datatype (which uses faceted values) and update the `PostList` type definition:

```haskell
type PostList = [(Label, Post)]
```

These simple changes affect the line count very little (aside from removing the 108-line FIO library).

4.5.2 Removing desirable dependence on FIO

Next, we completely remove the `policy` function and the lines in `main` that depend on it. Figure 4.13 shows the new `main` function. At this point, the functionality of FacetBook is intact, but its security guarantees have disappeared—in particular, all posts are now visible to all users, regardless of any permission settings on any posts. To reimplement this security feature, we define a new

```haskell
main :: IO ()
main = do  --IO
  r1 <- newIORef []
  r2 <- newIORef []
  let database = (r1, r2)
  let port = 3000
  Warp.run port $ \request respond -> do  --IO
    let unit_respond = \x -> do  --IO
          respond x
          return ()
    handle_request request database unit_respond
    return ResponseReceived
```

Figure 4.13: The code for the `main` function in V1-NoFIO.

function `filter_posts`:

```haskell
filter_posts :: Label -> PostList -> PostList
filter_posts k = filter (\(k',p) -> leq k' k)
```

and we call it inside the `dashboard` function just after reading the posts from the database:

```haskell
labeled_posts <- readIORef (fst database)
let posts = filter_posts (Whitelist [username]) labeled_posts
```

We must also add a line to the `do_create_post` function to label posts just before they are written into the database (line 175):

```haskell
174 d <- readIORef (fst database)
175 let labeled_data = ( Whitelist (username : users) ,
176                      username ++ ": " ++ content  )
177 writeIORef (fst database) (labeled_data : d)
```

4.5.3 Minimizing the TCB

With only the changes mentioned so far, the file `UCB.hs` is poorly named because it now contains code that belongs in the TCB. To rectify this situation, we begin by moving four functions from `UCB.hs` to `TCB.hs`, namely `handle_request`, `parse_request`, `do_create_post`, and `dashboard`. Finally, to keep the TCB as small as possible, we must rewrite `parse_request` so that it uses sandboxing for the other six types of request (besides `do_create_post` and `dashboard`). The new code is in Figure 4.14. Line 179 defines the `sandbox` function, which simply arranges for the posts to be censored from the database before calling a given handler `h`. By calling it on lines 182, 185, 194, 201, 203, and 205, we avoid the need to move any more functions from `UCB.hs` to `TCB.hs`.

Summary

In V1-NoFIO, the TCB contains 118 lines of code: 63 in `TCB.hs`, 45 in `Shared.hs`, and 10 import statements in `UCB.hs`. The UCB contains 295 lines of code: 305 in `UCB.hs` minus the 10 import statements at the top of the file.

Qualitatively comparing V1-FIO to V1-NoFIO is largely subjective. The application-specific TCB is smaller in V1-FIO; on the other hand, since FIO is part of the TCB, the total TCB size is less in V1-NoFIO.

Furthermore, the TCB code is qualitatively different in the two implementations. In V1-FIO, the structure of the TCB (especially the `policy` function) relieves auditors from digging through the codebase to find and verify security-critical operations, such as filtering the list of posts before displaying it, and correctly labeling new posts before inserting them into the database. On the other hand, one can argue that the `policy` function complicates the control flow. The control flow in V1-NoFIO is more straightforward, since there is no need to parse the

```
parse_request request =
  let sandbox h = \database respond ->
       let censored = (undefined, snd database)  in
       h censored respond  in
  if WAI.pathInfo request == ["login"] then
    sandbox $ UCB.login
  else case check_credentials request of
    Nothing ->
      sandbox $ UCB.authentication_failed
    Just username -> case WAI.pathInfo request of
      ["post"] ->
        let content = get_parameter request "content"  in
        let permissions = get_parameter request "permissions"  in
        let users = words permissions  in
        if content /= "" && all valid_username users then
          do_create_post username content users
        else
          sandbox $ UCB.compose_post username
      ["dashboard"] ->
        dashboard username
      ["tictactoe"] ->
        let partner = get_parameter request "partner"  in
        if valid_username partner then
          let action = get_parameter request "action"  in
          sandbox $ UCB.tictactoe_play username partner action
        else
          sandbox $ UCB.tictactoe_select_partner username
      _ ->
        sandbox $ UCB.not_found
```

Figure 4.14: The code for the `parse_request` function in V1-NoFIO.

```
respond $ WAI.responseLBS status200 headers $
    render_tictactoe new_game username partner
```

Figure 4.15: Excerpt of the code to display a Tic Tac Toe game in V1-FIO.

request twice.

4.6 Design V2: Adding a widget

Design V2 is the same as Design V1 except that the `tictactoe` page should now also display recent posts below the Tic Tac Toe game board. Figure 4.1 highlights the design change in the screenshot of the `tictactoe` page.

This design change affects the information flow of FacetBook because the `tictactoe` page now includes information from both portions of the database: the posts and the games.

4.7 V2-FIO

FacetBook V2-FIO implements Design V2, uses the FIO library, and is organized so that the change from V1 to V2 is as convenient as possible.

Figures 4.15 and 4.16 show the differences between V1-FIO and V2-FIO. Only these lines must change to implement the new widget.

In V2-FIO, the TCB is the same as in V1-FIO. The UCB contains 8 more lines of code.

Since the TCB is the same in V1-FIO and V2-FIO, no further changes are needed to minimize the TCB, which suggests that the information security is no worse than it was before. Furthermore, no special effort is required to maintain

```
209 d <- Read (fst database)
210 Swap $ do  --Fac
211   all_posts <- flatten d
212   return $ do  --FIO
213     respond $ WAI.responseLBS status200 headers $
214         render_tictactoe new_game username partner <>
215         "<br /><br />Recent posts:<hr />" <>
216         ByteString.intercalate "<hr />" (map escape (take 20 all_posts))
217 return ()
```

Figure 4.16: The new code to display a Tic Tac Toe game in V2-FIO.

```
218 respond $ WAI.responseLBS status200 headers $
219     render_tictactoe new_game username partner
```

Figure 4.17: Excerpt of the code to display a Tic Tac Toe game in V1-NoFIO.

confidence in security when making the change from Design V1 to Design V2.

4.8 V2-NoFIO

FacetBook V2-NoFIO implements Design V2 without using the FIO library, and is organized so that the change from V1 to V2 is as convenient as possible.

Figures 4.17 and 4.18 show the differences between V1-NoFIO and V2-NoFIO. Aside from these changes, we must also remove the call to `sandbox` on line 202, which ruins the carefully audited boundary between the TCB and UCB. As a result, in V2-NoFIO, the file `UCB.hs` is poorly named because its contents must now be audited for information leaks. The TCB includes the whole codebase: 429 lines of code.

Note that V2-NoFIO is still secure (thanks to the call to `filter_posts` on line

```
220 labeled_posts <- readIORef (fst database)
221 let d = filter_posts (Whitelist [username]) labeled_posts
222 let posts = flatten d
223 respond $ WAI.responseLBS status200 headers $
224     render_tictactoe new_game username partner <>
225     "<br /><br />Recent posts:<hr />" <>
226     ByteString.intercalate "<hr />" (map escape (take 20 posts))
```

Figure 4.18: The new code to display a Tic Tac Toe game in V2-NoFIO.

221), just like all the other versions of FacetBook; however, the auditing effort to confirm its information security increased significantly when we removed the call to `sandbox` on line 202.

4.9 V2-NoFIO-minTCB

FacetBook V2-NoFIO-minTCB implements Design V2 without using the FIO library, and is organized so as to minimize the size of the TCB. In this section, we highlight the differences from V2-NoFIO.

To minimize the TCB, we must move the `tictactoe_play` function from `UCB.hs` to `TCB.hs`. To keep the TCB as small as possible, we also refactor it to call three new functions: `UCB.tictactoe_error_response`, `UCB.update_game`, and `UCB.tictactoe_play_response`.

Figure 4.19 shows the new code for `tictactoe_play`. Lines 231 and 234 set up appropriate sandboxes for calling the UCB functions on lines 232 and 236, which relieves auditors from reading the code in `UCB.hs` (aside from its import statements).

Compared to V1-NoFIO, the TCB is 10 lines larger, which suggests that the change has reduced confidence in the security of the system. Compared to V2-

```
tictactoe_play username partner action database respond =
  if partner == username then
    respond $ UCB.tictactoe_error_response
  else do  --IO
    let censored_database = (undefined, snd database)
    new_game <- UCB.update_game username partner action censored_database
    labeled_posts <- readIORef (fst database)
    let d = filter_posts (Whitelist [username]) labeled_posts
    let posts = flatten d
    respond $ UCB.tictactoe_play_response new_game username partner posts
```

Figure 4.19: The code for the `tictactoe_play` function in V2-NoFIO-minTCB.

NoFIO, we modified 4 lines, moved 6 lines, and inserted 7 new lines; these changes were necessary to minimize the size of the TCB, suggesting that some nontrivial effort is required to maintain confidence in security. When FIO is unavailable, the next best sandboxing techniques lead to an inflexible architecture that becomes outdated when requirements change.

4.10 Conclusions

To quantitatively answer the question of whether FIO makes it easier to achieve information security, we constructed the prototype social network application FacetBook, and measured the code changes required to add a widget for displaying recent posts alongside the Tic Tac Toe game.

4.10.1 Research question 1

Does FIO help minimize TCB size when coding a secure application?

The FIO library has 108 lines of code, and the application-specific TCB in V1-FIO has 99 lines of code. The application-specific TCB in V1-NoFIO has 118 lines of code.

In terms of total size, the TCB is smaller in V1-NoFIO. On the other hand, the code in FIO is not application-specific, and so the burden of auditing it for correctness can be amortized over many applications. So our results our inconclusive on this question, as FIO could be considered helpful or not, depending on one's point of view.

4.10.2 Research question 2

Does FIO help minimize TCB size when changing an existing application to meet new requirements?

In the FIO version of FacetBook, the feature extension requires no significant refactoring:

- We merely add code for getting the posts and displaying them in a widget. The extension adds 0 lines of code to the TCB, and no special refactoring is required.

On the other hand, in the non-FIO codebase, we have two unappealing options:

- We could simply remove the sandboxing and implement the extension without refactoring any module boundaries. By taking this approach, we greatly increase the size of the TCB, which now includes all of the code pertaining to Tic Tac Toe, including all helper functions: 419 lines of code altogether.

- We could carefully refactor the modules so that we only add to the TCB the code related to displaying the new widget; the other helper functions can remain outside of the TCB. The net result is still a larger TCB (10 more lines) and extra developer effort (17 changes) spent on refactoring.

From this experiment, we conclude that the FIO library makes it possible in some situations to extend the functionality of applications at no extra cost (in terms of TCB lines and refactoring effort). In comparison, without FIO, this feature extension either significantly decreases security (via a larger TCB) or requires additional refactoring effort to mitigate such a decrease.

4.11 Discussion

One design decision is the richness of the security policy. For instance, we could include all of the rules of the Tic Tac Toe game in the policy, thus enforcing fair and correct playing of the game. However, since the security policy lies within the TCB, a larger policy means greater difficulty auditing the policy itself for correctness. Therefore, since correct functionality of the Tic Tac Toe game is less important than enforcing post visibility settings, we choose to include in the policy only the code pertaining to the latter criterion.

Another design choice is whether to make the policy a "transparent" wrapper around the functioning system (analogous to higher-order contracts being projections [4] that do not modify the behavior of correct programs) or to integrate the policy into the functioning system itself. For instance, in FacetBook, the policy code must inspect the request parameters to determine the request's meaning; should this part of the code be duplicated in the functioning system, which also needs to determine each request's meaning? We have chosen to duplicate this code, so there are some similarities in the control flow of functions `policy` and `parse_request` (Figures 4.6 and 4.12).

For the database, we use the FIORef type from our FIO library to keep persistent state in memory. For the list of ongoing Tic Tac Toe games, the FIORef will never become faceted because that data is public for everyone to see; however, for the faceted list of posts, the situation is more complicated. Specifically, since faceted execution works by refusing to update the facets that are forbidden from seeing the effects of the currently executing code, the data structure must operate in an append-only manner, lest we degrade performance by creating an exponentially large faceted structure. Some work by Algehed, Russo, and Flanagan [1] will address this performance-related limitation of faceted execution. For now, in FacetBook, we simply use two separate FIORefs: one for the list of Tic Tac Toe games (a non-faceted, non-append-only data structure), and one for the list of posts (a faceted, append-only data structure).

Bibliography

[1] Maximilian Algehed, Alejandro Russo, and Cormac Flanagan. "Optimizing Faceted Secure Multi-Execution". In: *Computer Security Foundations Symposium (CSF'19)*. IEEE. 2019.

[2] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. "Secure serverless computing using dynamic information flow control". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), p. 118.

[3] Ethan Cecchetti, Andrew C Myers, and Owen Arden. "Nonmalleable information flow control". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 1875–1891.

[4] Robert Bruce Findler and Matthias Blume. "Contracts as pairs of projections". In: *International Symposium on Functional and Logic Programming*. Springer. 2006, pp. 226–241.

[5] Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. "A Better Facet of Dynamic Information Flow Control". In: *WWW'18 Companion: The 2018 Web Conference Companion*. 2018, pp. 1–9.

[6] Michael D Schroeder. "Engineering a security kernel for multics". In: *ACM SIGOPS Operating Systems Review*. Vol. 9. 5. ACM. 1975, pp. 25–32.

[7] Jean Yang, Travis Hance, Thomas H Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. "Precise, dynamic information flow for database-backed applications". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2016.