Faceted Information Flow and Bi-Monadic Interpreters

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

Thomas H. Austin  
San José State University  
thomas.austin@sjtu.edu

Kenneth Knowles  
University of California, Santa Cruz  
kknowles@cs.ucsc.edu

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

Abstract

When an application fails to ensure information flow security, it may leak sensitive data such as passwords, credit card numbers, or medical records. News stories of such failures abound. Austin and Flanagan [2012] introduce faceted values – values that present different behavior according to the privileges of the observer – as a dynamic approach to enforcing 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. To illustrate a non-trivial use of the library, we present an interpreter for a small language whose information flow security is guaranteed by our library. This interpreter uses a monad in the traditional way for encapsulating value computations may be adapted to a typed language. The contributions of this paper are:

1. We present the first formulation of faceted values and computations in a typed context. As a consequence of this formulation, we clearly demonstrate the division between the public interface and private implementation of faceted values.
2. We show how faceted values may be integrated into a language as a library, rather than requiring modifications to the language runtime environment.
3. We also present a novel clarification of the relationship between explicit flows in pure calculations and implicit flows in impure computations. Specifically, we show that implicit flows between faceted computations manifest as explicit flows in the faceted expressions that determine which computation to run. These two interact via a single function that is a distributive law.
4. Finally, we present an interpreter for an imperative lambda calculus with dynamic information flow. The security of the implementation guarantees are primarily concerned with features and correct functionality; enforcement of security properties is generally only given proper attention after an exploit has occurred.

Just as memory-safe languages relieve developers from having to reason about memory management (and the host of bugs resulting from its mismanagement), information flow analysis is a promising mechanism to enforce security properties in a systemic fashion. Information flow controls require a developer to mark sensitive information, but otherwise automatically protect any “leaks” of this data. Formally, we refer to this property as noninterference; that is, public outputs do not depend on private inputs.

Secure multi-execution [Devriese and Piessens 2010; Jaskelioff and Russo 2012; Rafnsson and Sabelfeld 2013] has risen in popularity as an information flow enforcement technique. A program execution is split into two versions: the “high” execution has access to all sensitive information, but may only write to private channels; the “low” execution may write to public channels, but does not have access to any sensitive information. With this elegant approach, noninterference is ensured.

Austin and Flanagan [2012] presented faceted values as a technique for simulating secure multi-execution with a single process. A faceted value contains both a public and private facet. During program execution, these values keep track of both the public and private views of sensitive information. With this approach, a single execution can provide many of the same guarantees that secure multi-execution provides.

This paper extends the ideas of faceted values from an untyped variant of the lambda calculus to Haskell, showing how these concepts may be adapted to a typed language. The contributions of this paper are:

1. We present the first formulation of faceted values and computations in a typed context. As a consequence of this formulation, we clearly demonstrate the division between the public interface and private implementation of faceted values.
2. We show how faceted values may be integrated into a language as a library, rather than requiring modifications to the language runtime environment.
3. We also present a novel clarification of the relationship between explicit flows in pure calculations and implicit flows in impure computations. Specifically, we show that implicit flows between faceted computations manifest as explicit flows in the faceted expressions that determine which computation to run. These two interact via a single function that is a distributive law between the pure and impure faceted monads.
4. Finally, we present an interpreter for an imperative lambda calculus with dynamic information flow. The security of the implementation guarantees are primarily concerned with features and correct functionality; enforcement of security properties is generally only given proper attention after an exploit has occurred.

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 is also able to enforce integrity properties, such as that trusted outputs are not influenced by untrusted inputs.
1.1 Information Flow and Faceted Values Overview

In traditional information flow systems, information might be tagged with a label to mark it as confidential to particular parties. For instance, if \( \text{pin} \) should be restricted to \( \text{bank} \), our language might allow us to write that as:

\[
\text{pin} = 4321^{\text{bank}}
\]

In order to protect this value, we must prevent an unauthorized viewer from seeing any effect from this value. In particular, we must defend against explicit flows where a confidential value is directly assigned to a public variable or reference cell, and implicit flows where a confidential value may be deduced by reasoning about the control flow of the program. The following code shows an explicit flow from \( \text{pin} \) to the variable \( x \).

\[
\begin{align*}
\text{pin} &= 4321^{\text{bank}} \\
\text{x} &= \text{pin} + 1
\end{align*}
\]

Protecting against explicit flows is straightforward, and is handled by the taint mode of programming languages like Perl and Ruby; in contrast, handling implicit flows is much more complex.

Then, when we run the previous program, the value of \( \text{above2K} \) will be \( \langle \text{bank} \, ? \, \text{True} : \text{False} \rangle \). The bank sees the correct value of \( \text{True} \), but an unauthorized viewer instead sees \( \text{False} \), giving a consistent picture to the unauthorized viewer.

Label-based information flow systems can reason about multiple principles by joining labels together (e.g. \( 3^{A} + 4^{B} = 7^{AB} \)). In a similar manner, faceted evaluation can nest faceted values to represent different principals

\[
(A \, ? \, 3 : 0) + (B \, ? \, 4 : 0) = (A \, \langle k \, ? \, \text{False} : \text{True} \rangle : (B \, ? \, 7 : 3) \, : \, (B \, ? \, 4 : 0))
\]

essentially constructing a tree2 matching permissions to values.

Figure 1, adapted from Austin and Flanagan [2012], demonstrates a classic code snippet first introduced by Fenton [1974]; it illustrates how the use of two conditional statements may evade some information flow controls.

The input parameter \( x \) is a confidential boolean value, represented as \( \langle k \, ? \, \text{False} : \text{True} \rangle \) for false and \( \langle k \, ? \, \text{True} : \text{True} \rangle \) 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 \( x \) is \( \langle k \, ? \, \text{False} : \text{True} \rangle \), the value for \( y \) remains unchanged. Similarly, since the update to \( z \) depends only on \( y \) it remains public as well. Since no private information is involved in the update to \( z \), all strategies work in the same manner, shown in the All Strategies column of Figure 1.

The difference between these strategies lies in how they handle the update to \( y \) in the first conditional statement when the value of \( x \) is \( \langle k \, ? \, \text{True} : \text{True} \rangle \). 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 1, the influence of \( x \) is tracked by labeling it with \( k \). Since \( y \) is false, \( z \) remains unchanged, and also remains public. Thus, as can be seen by comparing the Naive and the All strategies columns, one bit of information leaks, 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. Similarly, the Fenton strategy forbids this update, but allows execution to continue. This approach avoids abnormal termination, but it may return inaccurate results, as shown in Figure 1.

Faceted evaluation solves this dilemma by simulating different executions of this program. In the Faceted Evaluation column, we see that the update to \( y \) results in the creation of a new faceted value \( \langle k \, ? \, \text{False} : \text{True} \rangle \). Any authorized viewer able to see \( k \)-sensitive data3 is able to see the real value of \( y \); unauthorized viewers instead see \( \text{True} \), thus hiding the value of \( x \). The value of \( z \) is updated in a similar manner in the second conditional assignment. The result of this function call therefore provides noninterference, avoids terminating execution abnormally, and provides accurate results to authorized users.

1.2 Faceted I/O

Austin and Flanagan [2012] treat both facet-aware reference cells and facet-aware file I/O, though with strikingly different mechanisms. While reference cells may contain faceted values, the assumption is that external systems might not share the same capability. Therefore, we must be able to send non-faceted, or “raw” values across channels. Consider the following code that writes a faceted character \( \langle k \, ? \, \text{a} : \text{b} \rangle \) to a handle \( h \):

\[
\text{writeIORef above2K True}
\]

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

3 That is, authorized to see data marked as sensitive to principal \( k \).
<table>
<thead>
<tr>
<th>x =</th>
<th>( \langle k \text{? } \text{False : } \bot \rangle )</th>
<th>( \langle k \text{? } \text{True : } \bot \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>do</td>
<td>All strategies</td>
<td>Naive</td>
</tr>
<tr>
<td>y &lt;- newIORef True</td>
<td>y = True</td>
<td>y = True</td>
</tr>
<tr>
<td>z &lt;- newIORef True</td>
<td>z = True</td>
<td>z = True</td>
</tr>
<tr>
<td>vx &lt;- readIORef x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>when vx</td>
<td>pc = {k}</td>
<td>pc = {k}</td>
</tr>
<tr>
<td>(writeIORef y False)</td>
<td>y = ( \langle k \text{? } \text{False : } \bot \rangle )</td>
<td>stuck</td>
</tr>
<tr>
<td>vy &lt;- readIORef y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>when vy</td>
<td>pc = {}</td>
<td></td>
</tr>
<tr>
<td>(writeIORef z False)</td>
<td>z = False</td>
<td></td>
</tr>
<tr>
<td>readIORef z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result:</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

**Figure 1.** A Computation with Implicit Flows

\( \text{hPutChar } h \langle k ? \text{ } 'a' : 'b' \rangle \)

When executing the above code, we must determine whether to send \('a'\), \('b'\), or no value at all. The solution offered by Austin and Flanagan [2012] associates a fixed set of security labels to a given channel. If \(k\)-sensitive data is permitted on the channel \(h\), then the value \('a'\) is written; otherwise \('b'\) is sent instead.

Additionally, some writes are restricted altogether if they are performed in a context that should not be visible. For example, consider the following code where a write is performed in the context of a sensitive boolean value:

\[
\text{if } \langle \text{bank }\text{? }\text{True : False} \rangle \text{then }
\text{hPutChar } \text{publicChannel } \langle 'z' \rangle
\text{else }
\text{return } ()
\]

Even though the value \('z'\) is public – all constants are public – the write to publicChannel must not be allowed unless it is authorized to see private banking information.

When we add interactive I/O, more subtleties arise. Consider the following code that writes to handle \(h\) and then reads the result from the same channel:

\[
\text{hPutChar } h \langle k ? \text{ } 'a' : 'b' \rangle
\text{hGetChar } h
\]

If the channel \(h\) may view \(k\)-sensitive data, the attacker might attempt to use the above code to break noninterference, effectively using I/O to serve as a form of declassification. To prevent this attack, reading from the channel incorporates the same sensitive influences; the result is \(\langle k ? \text{ } 'a' : \bot \rangle\) in this case, hiding the \(k\)-sensitive value from unauthorized viewers.

When the channel \(h\) may not view \(k\)-sensitive data, the proper result is less clear. The original paper addresses this issue by incorporating the negative influences of the view; that is, the result would return \(\langle k ? \bot : 'b' \rangle\), where the confidential value has disappeared from the system. While this approach guarantees noninterference, it also requires the system to keep track of all possible labels, thereby leading to practical challenges for a system with an arbitrary number of labels.

In this paper, we refine the earlier approach by simply ignoring the negative influences when reading a file. In other words, if channel \(h\) cannot observe \(k\)-sensitive data, reads will return an unfaceted value of \('b'\). In our experience, this appears to be a more natural approach when writing code and avoids many challenges of implementing a system supporting faceted values.

### 1.3 Bi-Monadic Interpreter

To illustrate the expressiveness of our library, we present an interpreter for an imperative lambda calculus with dynamic information flow.

When writing an interpreter in Haskell, it is traditional to structure the code using a monad (call it \(M\)) to encapsulate the computational effects of the interpreted language [Wadler 1992, 1995]. Then the interpreter (call it \(\text{eval}\)) will have an interface like the following:

\[
\text{type Action = } M \text{ Value}
\text{eval :: Term } \rightarrow \text{ Action}
\]

given some suitable definitions for Term and Value.

The code for our interpreter follows this traditional structure (using our library's \(FIO\) monad for impure faceted computations), but there is additional structure: the Value type for our interpreter is also monadic (using our library's Faceted monad for pure faceted values). So the interface to our interpreter roughly looks as follows:

\[
\text{type Value } = \text{ Faceted } \text{ RawValue}
\text{eval :: Term } \rightarrow \text{ Action}
\]

We call this a bi-monadic design.

One consequence of this is that the two monads can become "entangled" when an action depends on the raw data inside a value. In such a case, the action will be stuck inside the value monad, resulting in a type like this:

\[
\text{Faceted } (\text{FIO FIO Value})
\]

To turn these into Actions, we must use the function \(\text{swap}\) provided by our library, which has type

\[
\text{Faceted FIO a } \rightarrow \text{ FIO Faceted a}
\]

Applying this function will yield an object of type

\[
\text{FIO FIO Faceted RawValue},
\]

after which \(\text{liftM join}\) suffices to obtain the Action type

\[
\text{FIO FIO Faceted RawValue}
\]

and thus successfully “disentangle” the two monads.

### 2. Library Overview

We implement faceted computation in Haskell as a library that enforces information flow security dynamically, using abstract data types to prevent circumvention of dynamic protections. In contrast, the original formulation of Austin and Flanagan [2012] added faceted values pervasively to the semantics of a dynamically-typed,
type Label = String

data Faceted a

makePublic :: a → Faceted a
makeFaceted :: Label → Faceted a → Faceted a → Faceted a

bottom :: Faceted a

instance Functor Faceted
instance Applicative Faceted
instance Monad Faceted

The value makeFaceted accepts a value of type a, represented by the type Faceted a. (Section 2.1)

2. Imperative faceted computations influenced by the control flow path, represented by the type FIO a. (Section 2.2)

3. Faceted reference cells, represented by type FIORef a. (Section 2.3)

4. Faceted-enabled file handles / network sockets, represented by type FIOHandle. (Section 2.4)

2.1 Pure Faceted Values: Faceted a

Interface The public interface for the pure fragment of our library for faceted values is shown in Figure 2.

Information flow security is defined with respect to security labels. For simplicity and concision of presentation, we presume that security labels are strings, and set type Label = String, though leaving the type of labels abstract is straightforward. Our approach is compatible with other label systems, such as the decentralized label model of Myers [1999]. A security label may represent the privilege necessary to view a value or, dually, the level of integrity of a value. A view is a set of security labels. Intuitively, if a view L contains a label k, then that view has the privilege of viewing data protected by k or, dually, trusts data influenced by k.

A value of type Faceted a simultaneously represents the values, or facets, of type a observable to any view. The facets are not directly observable; the data type is abstract.

The function makePublic 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 view.

The function makeFaceted 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 view including k, the result behaves as priv. To all other views, the result behaves as pub (and so on, recursively).

The value bottom (which we occasionally abbreviate ⊥) is a member of Faceted a for any a, and represents a lack of a value. This is useful inside the public facet of a faceted value when a default value is necessary. Any computation based on bottom results in bottom.

From makeFaceted we can define a variety of derived constructors for creating faceted values with a minimum of effort. Consider the following two examples, which we provide with the library, but which do not increase expressivity.

makePrivate :: Label → a → Faceted a
makePrivate k v = makeFaceted k (makePublic v) bottom

makeFacets :: Label → a → a → Faceted a
makeFacets k priv pub = makeFaceted k (makePublic priv) (makePublic pub)

The function makePrivate creates a faceted value from a label k and a value v of type a. This value behaves as v for any view containing k, and is otherwise bottom. The function makeFacets creates a faceted value from a label k and two non-faceted values, simply allowing the programmer to elide tedious calls to makePublic.

The remainder of the public interface for the pure fragment of our library consists only of type class instances for Functor, Applicative, and Monad. The first two are necessarily determined by the instance for Monad, which is formulated to propagate security labels appropriately. Consider the following example, using Haskell's specialized do syntax to multiply two values of type Faceted Int.

\[
\begin{align*}
\text{do } x & \leftarrow (\text{makePrivate "k" } 7) \\
y & \leftarrow (\text{makePrivate "l" } 6) \\
\text{return } (x \times y)
\end{align*}
\]

The result will behave as 42 to any view containing both "k" and "l". To all other views, the result will behave as bottom.

Implementation The private implementation of faceted values is a simple algebraic data type:

data Faceted a =
| Raw a 
| Faceted Label (Faceted a) (Faceted a) 
| Bottom

The term Raw x denotes a faceted value with no visibility constraints. The term Faceted k priv pub (written \{ k ? priv : pub \}) by Austin and Flanagan [2012] and in our introduction; we now abandon that notation in favor of concrete Haskell syntax.) denotes a value that behaves as priv to any view containing k and behaves as pub to any other view (and so on, recursively). The term Bottom denotes a lack of a value.

In fact, this data type may be recognized as the free monad\(^4\) over the following non-recursive data type Facets:

data Facets a = Facets Label a a | Bot

data Free f a = Pure a | Free (f (Free f a))
type Faceted = Free Facets

This fact provides ready-made instances of Monad, Applicative, and Functor which are guaranteed to obey the appropriate equational laws. However, computing with the Haskell implementation of the free monad introduces an extra layer of data constructors pervasively. So for concision and clarity, we continue our presentation using the recursive data type Faceted.

\(^4\) This computation may also be written as (\ast\) \leftrightarrow (\text{makePrivate "k" } 7) \&\& (\text{makePrivate "l" } 6).

In general, computations on faceted values may be written tersely using Monad and Applicative combinators such as liftM2, liftA2, \&\&, \&\&\&, according to taste. Here we choose to use Haskell’s do-notation as it is readable and familiar to even novice Haskell programmers.

\(^5\) A Haskell implementation of free monads over functors is available on Hackage in the library free [Kmett 2014]
The implementations of `makePublic` and `makeFaceted` are simple wrappers on the appropriate constructors:

```haskell
makePublic = Raw
makeFaceted = Faceted
```

The instances of `Functor`, `Monad`, and `Applicative` are straightforward – they are all consequences of `Faceted` being a free monad. Here are their implementations in full:

```haskell
instance Functor Faceted where
  fmap f (Raw x) = Raw (f x)
  fmap f (Faceted k x y) = Faceted k (f x) (f y)
  fmap _ Bottom = Bottom

instance Applicative Faceted where
  pure x = Raw x
  (Faceted k f g) <*> x = Faceted k (f x) (g x)
  Bottom <*> _ = Bottom

instance Monad Faceted where
  return x = Raw x
  (Faceted k x y) >>= f = Faceted k (x >>= f) (y >>= f)
  Bottom >>= _ = Bottom
```

### 2.2 Side-Effecting Faceted Computations: `FIO` α

Information flow security is nearly trivial for the pure language of Section 2.1 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 may differ depending on the value of private data.

In the following classic example, assume that `secret` has type `Int` and the variable `x` is a reference cell of type `IORef Int` that initially contains the value 0.

```haskell
do if secret == 42
  then writeIORef x 1
  else return ()
readIORef x
```

The return value will be 1 if and only if `secret` is 42. Note, however, that the value stored at `x` does not explicitly depend on `secret`.

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 as

```haskell
do v <- secret
  do if v == 42
    then writeIORef x 1
    else return ()
  readIORef x
```

The outer do begins a computation in the `Faceted` monad, while the inner do begins a computation in the `IO` monad. This expression has type `Faceted (IO Int)`, so it cannot be “run” as part of a Haskell program. Using only the pure fragment of our library, implicit flows are prevented. Unfortunately, all implicit flows are prevented, even those that do not violate information flow security.

Guided by the types, we need a way to convert a value of type `Faceted (IO a)` to a value of type `IO (Faceted a)`. This latter should then be run to yield a value of type `Faceted a`, where the implicit flow is taken into account in the facets. In terms of our example, for any view for which `secret` is visible, the value returned should behave as 1, while for all other views it should behave as 0.

The remainder of this section focuses on the general design of faceted `IO` computations, while Section 2.3 and Section 2.4 illustrate implementations of specific side-effects.

---

**Interface** Faceted IO computations take place in the `FIO` monad (the name is short for “Faceted I/O”). Figure 3 shows the public interface for this fragment of the library. When control flow is influenced by faceted data, the result of a computation implicitly depends on the facets observed in determining the control flow path; the implementation of `FIO` manages this transparently.

The `Monad` instance for `FIO` allows for sequencing computations in the usual way, so `FIO` should act 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`.

```haskell
do v1 <- fio1
  v2 <- fio2
return (v1 + v2)
```

The function `secureRunFIO` converts a value of type `FIO α` to a value of type `IO α` visible to the empty view. Thus any computation depending on private data is not observable.

The novel, and vital, component of this interface is the function `swap`, which mediates between `Faceted` values and side-effecting `FIO` computations:

```haskell
swap :: Faceted (FIO a) → FIO (Faceted a)
```

Consider the following fragment, where `someFacetedBool` has type `Faceted Bool` and each of `fio1` and `fio2` have type `FIO a`.

```haskell
fio :: Faceted (FIO a)
fio = do v <- someFacetedBool
        if v then fio1 else fio2
runnable :: FIO (Faceted a)
runnable = swap fio
```

The type of `fio` is `Faceted (FIO α)`. As in the prior example, it cannot be run. To run `fio` in a sound way, we use the function `swap` to convert from `Faceted (FIO a)` to `FIO (Faceted a)`. The side effects of the resulting computation account for the implicit flow from `someFacetedBool`.

**Implementation** Somewhat analogous to how a value of type `Faceted a` simultaneously represents a value of type `a` for all views, a computation of type `FIO a` simultaneously represents many computations of type `IO a`. For each set of influences on the control flow path, the computation performed may differ. For a particular label `k`, the control flow path may have been influenced by data from a private facet protected by `k`, data from the public facet of a value faceted on `k`, or neither. We call the influence of a label on the current control flow a `branch` and model it directly via the data type `Branch` as either `Private k` or `Public k` (The notations of Austin and Flanagan [2012] for these are `k` and `K`, respectively).

```haskell
data Branch = Private Label | Public Label
```

The branch `Private k` denotes a control flow path that has been influenced by the private facet of a value faceted on `k`. The branch `Public k` denotes a control flow path that has been influenced by the public facet of a value faceted on `k`.

---

### Figure 3. Interface for `FIO`
A program counter label is a finite set of such branches describing all the influences on the current control flow path. We model a program counter label with a list here for readability.

```
type PC = [Branch] -- finite
```

Now we may precisely state that a value of type FIO a simultaneously represents some underlying IO a for any program counter label. In other words, a faceted computation may be represented as a function of type PC → IO a.

```
data FIO a = FIO (runFIO :: PC → IO a)
```

This data type is private to the library, and we expose only the limited ability to run an FIO with an empty program counter label.

```
secureRunFIO fio = runFIO fio []
```

The implementation of swap simulates an optimized multi-execution, adding the appropriate branches to the program counter label when determining which computations to perform. Inconsistent paths – where the program counter label contains both Public k and Private k for any k – are pruned.

```
swap (Raw fio) = FIO (Apc → runFIO fio pc >>= (return ○ Raw))
swap (Bottom) = FIO (Apc → return Bottom)
```

```
swap (Faceted k priv pub) = FIO branchOnPC
where branchOnPC pc
  | Private k 'elem' pc = runFIO (swap priv) pc
  | Public k 'elem' pc = runFIO (swap pub) pc
  | otherwise = do privV ← runFIO (swap priv) (Private k : pc)
                   pubV ← runFIO (swap pub) (Public k : pc)
                   return (Faceted k privV pubV)
```

If the value is not faceted, then the computation is run directly, and the result of the computation is not faceted. If the value is Bottom, then we run a trivial computation that returns Bottom.

If the value is faceted, then we must consider whether its label is already visible for the current program counter label pc. If so, then the private computation is run; if the label is not visible for pc, then the public computation is run. On the other hand, if the program counter label has not been influenced by Private k or Public k, then each computation is run under the appropriate added assumption about k. In the last case, the computations are sequenced, creating an apparent asymmetry. However, for the actual FIO operations we provide – discussed below in Section 2.3 and Section 2.4 – the two computations operate on disjoint facets. So the result is symmetrical, and could be executed in the opposite order. This behavior is a design constraint for any FIO operation.

Another way to describe swap is as a distributive law – meaning it respects the monad structures of FIO and Faceted in particular ways, see Barr and Wells [1985] – between the Faceted and FIO monads. Via swap, the composition FIO ∘ Faceted forms a compatible monad, where any number of “layers” of FIO and Faceted are collapsed. Delving further into category theory is not the aim of this paper, but intuitively this monad appears similar to the computational domain of prior work, where faceted computation and faceted values are conflated. We do conjecture that, relative to the appropriate semantic interpretation of Faceted and FIO, the function swap forms a strong distributive law [Lüth and Ghani 2002] making this compatible monad equivalent to the coproduct FIO ⊕ Faceted.

### 2.3 Faceted Reference Cells: FIORef

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 adjust facets according to influences on the control flow path.

```
data FIORef a
newFIORef :: Faceted a → FIO (FIORef (Faceted a))
readFIORef :: FIORef (Faceted a) → FIO (Faceted a)
writeFIORef :: FIORef (Faceted a) → Faceted a → FIO ()
```

#### Figure 4. Interface for FIORef

**Interface** Figure 4 presents the core of the public interface to FIORef a. The interface directly parallels that of conventional reference cells of type IORef a.

When a reference cell x of type FIORef (Faceted a) is read using readFIORef, it yields a value v of type Faceted a that is faceted according to the explicit and implicit flows that led to v being written to x.

When a reference cell x is initialized via newFIORef or written via writeFIORef, the value v (of type Faceted a) is stored at x in a way that takes into account all the influences that led to the initialization or write being performed.

Using this interface, we can express a minimal example of an implicit flow using FIO and FIORef. In the following, secret has type Faceted Int and x has type FIORef (Faceted Int) and x is initially set to makePublic 0.

```
do swap (do v ← secret
  if v == 42
    then writeFIORef x 1
    else return ()
readFIORef x
```

Note the necessity of the function swap. The argument to swap has type Faceted (FIO ()), and is converted to a value of type FIO (Faceted ()) so it can be run. The resulting faceted value read from x will account for the influence from the facets of secret. What is an implicit flow in classical examples is actually quite explicit here!

```
This pattern is extremely common, and can be encapsulated in the helpful combinator iff:

```
iff :: Faceted Bool → FIO a → FIO a → FIO (Faceted a)
iff facettedBool thenBranch elseBranch =
  swap (do v ← facettedBool
  return $ if v then thenBranch else elseBranch)
```
```
The above example may then be written

```
do iff (do (v ← secret; return $ v == 42 ))
  (writeFIORef x 1)
  (return ())
readFIORef x
```
```
#### Implementation

The private representation for a value of type FIORef (Faceted a) is simply a reference cell that stores a faceted value, of type IORef (Faceted a).

```
data FIORef a = FIORef (IORef a)
```

This implementation suggests that it might be possible to create an FIORef that contains an ordinary, non-Faceted value. However, the public interface functions only work when the contained type is Faceted.

Reading an FIORef requires no special logic because the appropriate facets are always established upon writing a value.

---

6Haskell has many structures providing functionality of mutable references; we are gradually adding support for other types of mutable references as well as the plethora of convenience functions provided in Haskell’s standard library.
readFIORef (FIORef x) = FIO (λpc → readFIORef x)

Initialization and subsequent writes to an FIORef incorporate the current program counter label, building facets to indicate that the value being stored may depend on any labels in the program counter. This process is implemented in the auxiliary function pcF, corresponding to \( (pc? x : y) \) from Austin and Flanagan [2012]. The value \( pcF pc x y \) behaves as \( x \) to any view consistent with \( pc \) and behaves as \( y \) to any other view.

\[
\begin{align*}
\text{pcF} & : \text{PC} \rightarrow \text{Faceted a} \rightarrow \text{Faceted a} \rightarrow \text{Faceted a} \\
\text{pcF} [\text{private b}] & : \text{bs} x y = \text{Faceted k} (\text{pcF bs x y}) y \\
\text{pcF} [\text{public k}] & : \text{bs} x y = \text{Faceted k} y (\text{pcF bs x y})
\end{align*}
\]

To initialize a new FIORef (Faceted a) requires an initial value \( v \). The value stored is faceted according to the current program counter, with a default value of \text{bottom}.

\[
\begin{align*}
\text{newFIORef} v & = \text{FIO} \& \lambda pc \rightarrow \\
\text{do} & \text{var} ← \text{newIORef} (\text{pcF pc v} \text{bottom}) \\
\text{return} & (\text{FIORef var})
\end{align*}
\]

Writing a faceted reference cell updates the stored (faceted) value in such a way that it appears unchanged to any view that is inconsistent with the current program counter label, but updated for any view that is consistent.

\[
\begin{align*}
\text{writeFIORef} (\text{FIORef x}) v & = \text{FIO} \& \lambda pc \rightarrow \\
\text{do} & \text{old} ← \text{readFIORef x} \\
\text{writeFIORef} x & = (\text{pcF pc v old})
\end{align*}
\]

Together, these definitions ensure that the value stored for any FIORef always contains facets that incorporate influences due to implicit flows.

2.4 File I/O and Network Communication: FHandle

Faceted file I/O differs from reference cells in that the network and file system, which we collectively refer to as the environment, lies outside the purview of our programming language. The environment has no knowledge of facets and cannot be retrofitted. In addition, it must be assumed that there are other agents/programs able to read from and write to the file. Thus we assume that the environment’s access controls are used appropriately to restrict other users of the handle. We merely provide facilities within Haskell to express the same policy that the environment is presumed to uphold.

Because the contents of a file can be written or read without warning, writing to a file and sending a message on an asynchronous channel are considered equivalent, while reading from a file and receiving a message over an asynchronous channel are likewise considered equivalent. For the purposes of demonstrating functions that closely match standard Haskell functions, we will use files but with this network send/receive semantics.

\textbf{Interface} Figure 5 shows the core of the public interface for facet-aware file handles, type \text{FHandle} (short for “facet-aware Handle”). Since there are a great many operations for handles, we present just those necessary to discuss how I/O interacts with faceted computation. In particular, we focus on sends and receives of a single value of type \text{Char} via \text{hGetChar} and \text{hPutChar}.

The policies that we support are those expressed by associating to each file handle \( h \) a finite view\(^7\) (a finite set of labels) \( \text{view}_h \), of type \text{View} that indicates the minimum confidentiality for data written to \( h \) and, dually, the maximum integrity of data read from \( h \).

\footnote{Providing performant faceted wrappers for a wide variety of operations on handles remains a work in progress.}

\footnote{Finiteness of views for file I/O is a stronger property than absolutely necessary. Our design requires at least that inclusion and non-inclusion of a label in a view be effectively computable.}

\textbf{Figure 5. Interface for FHandle}

\[
\begin{align*}
\text{data FHandle} & \\
\text{type View} & = [\text{Label}] -- \text{finite} \\
\text{openFileF} : : \text{View} \rightarrow \text{FilePath} \rightarrow \text{IODevice} \rightarrow \text{FHandle} & \\
\text{hCloseF} : : \text{FHandle} \rightarrow \text{FIO} () & \\
\text{hGetCharF} : : \text{FHandle} \rightarrow \text{FIO} (\text{Faceted Char}) & \\
\text{hPutCharF} : : \text{FHandle} \rightarrow \text{Faceted Char} \rightarrow \text{FIO} () &
\end{align*}
\]

\( h \). The detailed discussion below of reading and writing to facet-aware handles clarifies the meaning of these policies.

The function \text{openFileF} accepts a view \text{view}_h along with a file path and mode and returns a (computation that returns) a facet-aware handle \( h \) protected by the policy expressed by \text{view}_h. The computation \text{hCloseF} \( h \) closes a facet-aware file handle \( h \).

When writing to a handle \( h \) via \text{hPutCharF}, the view \text{view}_h serves as an upper bound on the confidentiality assured by the external world (a lower bound on the integrity expected) for data written to \( h \). In other words, we trust that the external world will protect the data with those labels in \text{view}_h. Dually, we certify that no labels beyond those in \text{view}_h have influenced the values sent. Thus if the program counter label has been influenced by higher confidentiality (lower integrity) values, the write must not occur.

Furthermore, the exact facet that is written is just that visible to implicit flows.

\[
\begin{align*}
\text{fio} & : : \text{FIO} () \\
\text{fio} & = \text{do} h ← \text{openFileF} [\text{"z", "w", "x"}] \text{"/tmp/foo" WriteMode} \\
& \text{hPutCharF h (makeFacets "z" 'a' 'b')} \\
& \text{hPutCharF h (makeFacets "z" 'c' 'd')}
\end{align*}
\]

First, consider running runFIO fio \([\text{"z", "w", "x"}]\). Because the label \text{"x"} is in the \text{PC} passed to runFIO, it is considered to have influenced control flow, and no writes will occur because \text{"x"} is not contained in \text{view}_h. Information confidential to \text{"x"} would be leaked by the mere fact of a write occurring.

Next, consider running runFIO fio \([\text{"z"]}. Because \text{"z"]} is a subset of \text{view}_h, the writes are permitted. The first call to \text{hPutChar} will output \text{"a"} because \text{"z"} is a contained in \text{view}_h. The second call to \text{hPutChar} will output \text{"d"} because \text{"x"} is not contained in \text{view}_h.

When reading from a handle \( h \) via \text{hGetCharF}, we treat \text{view}_h as a lower bound on the confidentiality expected by the external world (an upper bound on the integrity it promises) for data read from \( h \). In other words, we trust that the external world will prevent tainting from data with labels not in \text{view}_h. Dually, we certify that we will protect the data with sufficient security to avoid leaking to lesser views\(^8\). For example, consider the following computation.

\[
\begin{align*}
\text{do} h & ← \text{openFileF} [\text{"k", "l"] \text{"/tmp/socket.0" ReadMode} \\
& \text{hGetCharF h}
\end{align*}
\]

The character thus read from \( h \) is confidential, observable only to views that include labels \text{"k"} and \text{"l"}. Dually, the value is considered to be tainted by labels \text{"k"} and \text{"l"}, which may represent untrusted parties that should not be able to influence control flow.

These four primitives are sufficient to securely support communication via file handles. For convenience, our library also provides the following function to print a whole string, rather than just a single character at a time.

\footnote{We assume an asynchronous protocol, so an external writer can never observe whether a read has taken place.}
hPutStrF :: FHandle → Faceted String → FIO ()
hPutStrF h fs = do -- working in FIO monad
  swap $ do -- working in Faceted monad
    s ← fs
    return $ sequence $ map (hPutCharF h o return) s
  return ()

Implementation The representation of a value of type FHandle is as a pair of a Handle and a View. The functions openFileF and hCloseF are simple wrappers on the standard library functions openFile and hClose.

data FHandle = FHandle View Handle

openFileF :: View → FilePath → IOMode → FIO FHandle
openFileF view path mode = FIO $ λpc →
  do handle ← openFile path mode
     return (FHandle view handle)

hCloseF :: FHandle → FIO ()
hCloseF (FHandle _ handle) = FIO $ λ_ → hClose handle

The implementation of hGetCharF creates appropriate facets to provide information flow security for values received from a handle h. To any view that is not a superset of viewh, the value will appear as bottom.

hGetCharF :: FHandle → FIO (Faceted Char)
hGetCharF (FHandle view handle) = FIO hGetCharForPC where hGetCharForPC pc =
  do ch ← hGetChar handle
     return (pcF (map Private view) (Raw ch) bottom)

The implementation of hPutCharF actually performs a write only when viewh is visible to the current program counter label, as computed by the auxiliary function visibleTo. When writing a faceted character ch, the non-facted character written is the projection of ch to viewh, as computed by the auxiliary function project.

visibleTo :: PC → View → Bool
visibleTo pc view = all consistent pc
  where consistent [Private k] = k 'elem' view
        consistent [Public k] = k 'notElem' view

project :: View → Faceted a → Maybe a
project view Bottom = Nothing
project view (Raw v) = Just v
project view (Faceted k priv pub) |
  | k 'elem' view = project view priv |
  | k 'notElem' view = project view pub

hPutCharF :: FHandle → Faceted Char → FIO ()
hPutCharF (FHandle view handle) = FIO $ hPutCharForPC where hPutCharForPC pc =
  | pc 'visibleTo' view = case project view ch of
    | Just c → hPutChar handle c
    | Nothing → return () |
  | otherwise = return ()

For example, if viewh is ["k"] then the computation

hPutCharF h $ Faceted "k" (makeFacets "l" 'a' 'b') (makePublic 'c')

will write the character 'b' to h, because the private component of the "k" facet will be selected, while the public component of the "l" facet will be selected.

If the projection of the faceted value onto the view is Bottom, then no write is actually performed.

3. Application: A Bi-Monadic Interpreter

To demonstrate the expressiveness of the Faceted library, we present a monadic interpreter for the imperative lambda calculus with dynamic information flow. Its information flow security is guaranteed by the correctness of the library. This interpreter also shows the necessity of swap, which allows I/O computations to depend on faceted values.

3.1 The Interpreted Language

The source language is an imperative lambda calculus with dynamic information flow, whose abstract syntax is defined in Figure 6. The language has:

- variables, lambda abstractions, and applications,
- integer literals, addition, and test for positivity,
- a primitive for printing integers to a global output stream,
- reference cells with imperative assignment and dereference, and
- a primitive for upgrading the security level of a value.

The imperative language constructs have side effects during evaluation, and the evaluation order is call-by-value.

The desired security property is that high security integers are not printed to the output stream. To guarantee this property, our implementation opens the stream using openFileF with the empty view, and ensures that high-security values are faceted on the label "H".

3.2 Implementation

Figure 7 shows the interpreter eval. The interesting aspect of this interpreter is that it requires two distinct monads.

- The FIO monad captures computations (called Actions 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.

\[\text{Figure 6. Data types for the bi-monadic interpreter.}\]
-- Postfix function application.
x |> f = f x

eval :: Env → Term → Action

eval (Var x) = return $ e x

eval (Lam x t) = return $ makePublic $ FnVal $ λv → eval (extend e x v) t

eval (Num n) = return $ makePublic $ IntVal n

eval (MakePrivate t) = do v ← eval e t
-- working in FIO monad

eval e (Plus t1 t2) = do v1 ← eval e t1
v2 ← eval e t2
-- working in FIO monad
( do IntVal n1 ← v1
IntVal n2 ← v2
return $ IntVal (n1 + n2) ) |> return
-- working in Faceted monad

eval e (Ref t) = do v ← eval e t
ref ← newFIORef v
return $ makePublic $ LocVal ref
-- working in FIO monad

eval (App t1 t2) = do v1 ← eval e t1
v2 ← eval e t2
-- working in Faceted monad
( do FnVal f ← v1
return $ f v2 ) |> swap |> liftM join
-- working in FIO monad

eval (IfPositive tb t1 t2) = do vb ← eval e tb
-- working in Faceted monad
( do IntVal n ← vb
if n > 0 then
return $ eval e t1
else
return $ eval e t2
) |> swap |> liftM join
-- working in FIO monad

eval (Deref t) = do v ← eval e t
-- working in Faceted monad
( do LocVal ref ← v
return $ readFIORef ref ) |> swap |> liftM join
-- working in Faceted monad

eval (Print t) = do v ← eval e t
-- working in Faceted monad
( do IntVal n ← v
return $ facetAwarePrint n ) |> return ) |> swap
-- working in FIO monad

eval (Assign t1 t2) = do v1 ← eval e t1
v2 ← eval e t2
-- working in FIO monad
( do LocVal ref ← v1
rv2 ← v2
( do writeFIORef ref v2
return rv2 ) ) |> return ) |> swap
-- working in FIO monad

facetAwarePrint :: Int → FIO ()
facetAwarePrint n = do
h ← openFileF [] "output.txt" AppendMode
hPutStrF h (return $ facetAwarePrint n)
hCloseF h
return ()

extend :: Env → String → Value → Env
extend e x v y | x == y = v
| otherwise = e y

Figure 7. The bi-monadic interpreter eval function.
Even though the Faceted monad is used in this non-traditional manner by our monadic interpreter, it seems that the monad interface is exactly what is needed for faceted values—particularly considering the necessity of the monad-specific operation

\[ \text{join} :: \text{Faceted} \ (\text{Faceted} \ a) \rightarrow \text{Faceted} \ a \]

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

In the case of Var, the value is obtained from the environment and is thus already faceted; we use return to inject it into the FIO monad. For Lam and If, we use makePublic to inject the constant into the Faceted monad, and then return to inject into FIO.

The case for MakePrivate looks quite standard for a monadic interpreter. The subterm is evaluated to a value, and then a new value is computed (using makeFaceted in this case) and returned.

For the Plus operation, the interpreter should first evaluate the two subterms (accumulating their side effects) to obtain two faceted values. Next, it should extract integers from these values, and then add the integers. Then return turns the sum into a faceted value, and a final return yields an Action. The swap operation is not necessary in this case because there are no computational effects that depend on the contents of a faceted value.

The Ref operation first evaluates its subterm and then allocates and returns a location. This does not require swap either because it is not necessary to access the contents of any faceted value during this operation.

On the other hand, the cases for App, IfPositive, and Deref require swap. They all share similar structure. First, one or two subterms are evaluated (accumulating side effects). Next, a Faceted value is inspected to determine what action should be run next. The appropriate action is returned, and the resulting type is Faceted (FIO (Faceted RawValue)). Next, the function swap is used to get FIO (Faceted (Faceted RawValue)), and then (liftM join) flattens the two levels of facets. In the case of App, one level is from the function itself, and the other is from evaluating its body; in the case of IfPositive, one level is from the discriminator, and the other is from the arm of the conditional; in the case of Deref, one level is from the location itself, and the other is from the contents of the location.

In the remaining cases (Print and Assign), it is not necessary to use (liftM join) because these cases do not need to join the security levels of two different values. However, they must still use swap to enable the side effects to depend on a faceted value.

Figure 8 shows a program in the interpreted language. It has an illegal implicit flow, just like the one in Figure 1. The result of the program corresponds to the column for faceted evaluation:

Faceted "H" (Raw (IntVal 1)) (Raw (IntVal 0))

The dependence on the private input x is isolated to the private facet (IntVal 1); the public facet (IntVal 0) is independent of x.

3.3 Discussion

Traditional monadic interpreters use just one monad that encapsulates all computational effects. Monad transformers are popular for combining multiple effects [Steele 1994; Liang et al. 1995], but the resulting composite is still a single monad.

Jones and Duponcheel [1993] observe that two monads can form a composed monad (in our case FIO ◦ Faceted) when there exists an operation

\[ \text{prod} :: \text{Faceted} \ (\text{FIO} \ (\text{Faceted} \ a)) \rightarrow \text{FIO} \ (\text{Faceted} \ a) \]

Our swap operator is clearly sufficient, since we may define

\[ \text{prod} = \text{liftM} \ \text{join} \circ \text{swap} \]

Hence, we could have used the traditional design, with (FIO ◦ Faceted) as our monad. However, this would obscure the two monads’ distinct roles: FIO encapsulates side effects in the control flow, while Faceted encapsulates side effects in the data flow.

For instance, consider using the Maybe monad in place of both FIO and Faceted. This gives us a language supporting two different kinds of failed computations: using Maybe on the values will allow us to propagate failure along data flow paths (much like NaN in floating point arithmetic); on the other hand, using Maybe on the actions will allow us to abort computations wholesale (much like exceptions).

To find the appropriate definition for swap, it helps to give each monad its own name and constructors:

\[
\begin{align*}
\text{data} & \ \text{Maybe1} \ a = \ \text{Success} \ a \mid \ \text{Error} \\
\text{data} & \ \text{Maybe2} \ a = \ \text{Defined} \ a \mid \ \text{Undefined} \\
\text{swap} :: & \ \text{Maybe1} \ (\text{Maybe2} \ a) \rightarrow \ \text{Maybe1} \ (\text{Maybe2} \ a) \\
\text{swap} & = \ \text{case} \ \text{swap} \ (\text{Defined} \ \text{Error}) \rightarrow \ \text{Success} \ \text{Undefined} \\
\text{swap} & = \ \text{case} \ \text{swap} \ (\text{Defined} \ \text{Success} \ x) \rightarrow \ \text{Success} \ (\text{Defined} \ x)
\end{align*}
\]

4. 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 [Fenton 1974]. 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 [Fenton 1974] or terminate the program on these updates [Zdancewic 2002; Austin and Flanagan 2009]; 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 [Dhawan and Ganapathy 2009; Jang et al. 2010; Hedin and Sabellfeld 2012; Kerschbaumer et al. 2013].

Denning’s work [Denning 1976; Denning and Denning 1977] 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. [1996] and the SLam Calculus [Heintze and Riecke 1998]. Jif [Myers 1999] uses this strategy for a Java-like language, and has become one of the more widespread languages providing information flow guarantees. Sabellfeld and Myers [2003] provide an excellent history of information flow analysis research prior to 2003. For a more detailed comparison of the benefits of dynamic controls and static analysis
for information flow guarantees, see Russo and Sabelfeld [2010]'s discussion between static and dynamic analyses.

Secure multi-execution [Devriese and Piessens 2010] executes the same program multiple times representing different “views” of the data. For a simple two-element lattice of high and low, a program could be executed twice, once where confidential (high) data is included but may only write to authorized channels, and again where high data has been replaced by default values and which writes to public channels. This approach has since been implemented in the Firefox web browser [De Groof et al. 2012] and as a Haskell library [Jaskelioff and Russo 2012]. Rafnsson and Sabelfeld [2013] show an approach to handle declassification and to guarantee transparency with secure multi-execution.

Zanarini et al. [2013] note 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 [Austin and Flanagan 2012] simulates secure multi-execution by the use of special faceted values, which track different views for data based on the security principals involved. While faceted evaluation cannot be parallelized as easily, it avoids many redundant calculations, thereby improving efficiency [Austin and Flanagan 2012]. It also allows declassification, where private data is released to public channels. Austin et al. [2013] exploit this benefit to incorporate policy-agnostic programming techniques, allowing for the specification of more flexible policies than traditionally permitted in information flow systems.

Research on information flow analysis in Haskell has been limited, perhaps in part because idiomatic pure Haskell code avoids the complexity of implicit flows. Nonetheless, side effects are important for Haskell programming, and information flow security for Haskell in the presence of implicit flows has been studied before. Li and Zdancewic [2006] implement an information flow system in Haskell, embedding 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. [2008] 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 who/when/what dimensions proposed by Sabelfeld and Sands [2009].

Devriese and Piessens [2011] illustrate how to enforce information flow in monadic libraries. A sequence operation e₁ >>= e₂ is distinguished from a bind operation e₁ >>= e₂ in that there are no implicit flows with the >>= operator. They demonstrate the generality of their approach by applying it to classic static [Volpano et al. 1996], dynamic [Sabelfeld and Russo 2010], and hybrid [Guernic et al. 2006] information flow systems.

Stefan et al. [2011] 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 [2013] 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 [1992] 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 [Steele 1994] and of monad transformers [Liang et al. 1995]. 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.

References


11


P. Wadler. Monads for functional programming. In Advanced Functional Programming, First International Spring School on Advanced Func-