Faceted Dynamic Information Flow via Control and Data Monads

Thomas Schmitz[1], Dustin Rhodes[1], Thomas H. Austin[2], Kenneth Knowles[1], and Cormac Flanagan[1]

[1] U. C. Santa Cruz
[2] San José State

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.

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[3].

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.

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.

[3] 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.

– 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 Review of Information Flow and Faceted Values

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

Fig. 1. A computation with implicit flows.

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:

```
pin = 4321^bank
```

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 straight-

forward 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 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 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 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 $\langle bank \ ?$ `True` : `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. $3^A + 4^B = 7^{AB}$). In a similar manner, faceted evaluation nests faceted values to represent multiple principals, essentially constructing a tree[4] mapping permissions to values:

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

[4] 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.

Figure 1, adapted from Austin and Flanagan [2], demonstrates a classic 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 $\langle k \ ?$ `False` $: \bot\rangle$ for false and $\langle k \ ?$ `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 $\langle k \ ?$ `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 $\langle k \ ?$ `True` $: \bot\rangle$ is more interesting, as illustrated in Figure 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 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 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 $\langle k \ ?$ `False` $:$ `True`$\rangle$. Any viewer authorized to see k-sensitive data[5] 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 $\langle\, k \,?$ `True` $:$ `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.

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

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:

- 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`).

3.1 Pure Faceted Values: `Faceted a`

Figure 2 shows the public interface for the pure fragment of our library. This fragment tracks explicit data flow information in pure computations.

```
type Label = String

data Faceted a

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

instance Monad Faceted
```

Fig. 2. Interface for the pure fragment of the `Faceted` library.

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 $\bot$) 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.

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

For the pure language of Section 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 ())
```

Fig. 3. Interface for `FIO` and `FioRef`.

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 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 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 (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 4, we will explain the machinery that implements this secure behavior.

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 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

```
data FHandle

type View = [Label]

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

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

Fig. 4. Interface for `FHandle`.

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
```

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 5 and 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.

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.
- $\sigma, A \Downarrow_{pc}^{\mathsf{FIO}} \sigma', v$ means that the Haskell program "`main = runFIO` A pc" changes the store from σ to σ' and yields the result v.

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

Fig. 5. Source syntax.

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

Fig. 6. Runtime syntax.

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

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

$$\frac{t \text{ not a value} \quad t \Downarrow v_1 \quad E[v_1] \Downarrow v_2}{E[t] \Downarrow v_2}\ [\text{E-CTXT}] \qquad \frac{}{\texttt{return}^{\mathsf{Fac}}\ t \Downarrow \texttt{public}\ t}\ [\text{E-RET}]$$

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

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

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

Fig. 7. Semantics (part 1).

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c} t \Downarrow F \\ a \notin \mathrm{dom}(\sigma) \\ F' = \langle\langle pc\ ?\ F : \mathtt{bottom}\rangle\rangle \end{array}}{\sigma, \mathtt{newFioRef}\ t \Downarrow_{pc}^{\mathsf{FIO}} \sigma[a := F'], a}\ [\text{F-NEW}]$$

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

$$\frac{\begin{array}{c} t_1 \Downarrow a \\ \sigma' = \sigma[a := \langle\langle pc\ ?\ t_2 : \sigma(a)\rangle\rangle] \\ v = \mathtt{public}\ () \end{array}}{\sigma, \mathtt{writeFioRef}\ t_1\ t_2 \Downarrow_{pc}^{\mathsf{FIO}} \sigma', v}\ [\text{F-WRITE}]$$

$$\frac{\begin{array}{c} t \Downarrow h \\ pc \text{ is not visible to } view_h \end{array}}{\sigma, \mathtt{getCharFio}\ t \Downarrow_{pc}^{\mathsf{FIO}} \sigma, \mathtt{bottom}}\ [\text{F-GET-2}]$$

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

$$\frac{\begin{array}{c} t_1 \Downarrow h \\ L = view_h \\ pc \text{ is visible to } L \\ t_2 \Downarrow ch \\ \sigma' = \sigma[h := \sigma(h)ch] \end{array}}{\sigma, \mathtt{putCharFio}\ t_1\ t_2 \Downarrow_{pc}^{\mathsf{FIO}} \sigma', ()}\ [\text{F-PUT}]$$

$$\frac{\begin{array}{c} t_1 \Downarrow h \\ L = view_h \\ pc \text{ is not visible to } L \end{array}}{\sigma, \mathtt{putCharFio}\ t_1\ t_2 \Downarrow_{pc}^{\mathsf{FIO}} \sigma, ()}\ [\text{F-PUT-2}]$$

Fig. 8. Semantics (part 2).

Figure 7 depicts the pure derivation rules. These rules describe a call-by-name λ-calculus with opaque constants and two library functions: $\texttt{return}^{\mathsf{Fac}}$ and $\texttt{bind}^{\mathsf{Fac}}$. These monad operators for `Faceted` are particularly simple because it is a free 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 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 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:

$$\begin{aligned}
\langle\langle \emptyset\ ?\ t_1 : t_2 \rangle\rangle &= t_1 \\
\langle\langle \{k\} \cup pc\ ?\ t_1 : t_2 \rangle\rangle &= \texttt{faceted}\ k\ \langle\langle pc\ ?\ t_1 : t_2 \rangle\rangle\ t_2 \\
\langle\langle \{\overline{k}\} \cup pc\ ?\ t_1 : t_2 \rangle\rangle &= \texttt{faceted}\ k\ t_2\ \langle\langle pc\ ?\ t_1 : t_2 \rangle\rangle
\end{aligned}$$

[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

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

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.

4.1 Termination-Insensitive Noninterference

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

$$\begin{aligned}
\varepsilon_L(\texttt{faceted}\ k\ t_1\ t_2) &= \varepsilon_L(t_1) && \text{if } k \in L \\
\varepsilon_L(\texttt{faceted}\ k\ t_1\ t_2) &= \varepsilon_L(t_2) && \text{if } k \notin L \\
\varepsilon_L(\bullet) &\text{ is homomorphic otherwise.}
\end{aligned}$$

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

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

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:

$$\begin{aligned}
\varepsilon_L(\sigma_1) &= \varepsilon_L(\sigma_2) & \varepsilon_L(A_1) &= \varepsilon_L(A_2) \\
\sigma_1, A_1 &\Downarrow^{\mathsf{FIO}}_{\emptyset} \sigma_1', v_1 & \sigma_2, A_2 &\Downarrow^{\mathsf{FIO}}_{\emptyset} \sigma_2', v_2
\end{aligned}$$

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.

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.

5.1 The Interpreted Language

The source language is an imperative call-by-value λ-calculus whose abstract syntax is defined in Figure 9. The language has variables, lambda abstractions, applications, and primitive constants for manipulating reference cells, performing I/O, and creating private values.

```
data Term =
    Var String                  -- Lambdas
  | Lam String Term
  | App Term Term
  | Const Value                 -- Constants
```

Fig. 9. Syntax for the bi-monadic interpreter.

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

5.2 Implementation

Figure 10 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 in Figure 10). 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

```
1  -- Runtime data structures.
2  data RawValue =
3      CharVal Char                -- Characters
4    | RefVal (FioRef Value)       -- Mutable references
5    | FnVal (Value → Action)      -- Functions
6  type Value  = Faceted RawValue
7  type Action = FIO Value
8  type Env    = String → Value
9
10 -- Interpreter.
11 eval :: Env → Term → Action
12 eval e (Var x)     = return $ e x
13 eval e (Lam x t)   = return $ return $ FnVal $ λv →
14                        eval (extend e x v) t
15 eval e (App t1 t2) = do v1 ← eval e t1       -- working in FIO monad
16                         v2 ← eval e t2
17                         prod $ do
18                           FnVal f ← v1       -- working in Faceted monad
19                           return $ f v2
20 eval e (Const v)   = return v
21
22 -- Constants.
23 private :: RawValue
24 private = FnVal $ λv →
25   return $ faceted "H" v bottom
26 ref :: RawValue
27 ref = FnVal $ λv → do                        -- working in FIO monad
28   ref ← newFioRef v
29   return $ return $ RefVal ref
30 deref :: RawValue
31 deref = FnVal $ λv → prod $ do               -- working in Faceted monad
32   RefVal ref ← v
33   return $ readFioRef ref
34 assign :: RawValue
35 assign = FnVal $ λv1 →
36   return $ return $ FnVal $ λv2 → prod $ do  -- working in Faceted monad
37     RefVal ref ← v1
38     rv2 ← v2
39     return $ do                             -- working in FIO monad
40       writeFioRef ref v2
41       return v2
42 printChar :: RawValue
43 printChar = FnVal $ λv → prod $ do           -- working in Faceted monad
44   CharVal c ← v
45   return $ do                               -- working in FIO monad
46     h ← openFileFio [] "output.txt" AppendMode
47     putCharFio h (return c)
48     closeFio h
49     return v
```

Fig. 10. The bi-monadic interpreter `eval` function.

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.

```
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
```

Fig. 11. A sample program for the interpreter. For ease of reading, we assume the availability of standard encodings for `let` and boolean operations.

Figure 11 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)
```

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[6]. 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, embedding a language for creating secure modules. Their enforcement mechanism

[6] 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.

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 `>>` e_2 is distinguished from a bind operation e_1 `>>=` e_2 in that there are no implicit flows with the `>>` 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].

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.

Acknowledgements

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

References

[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 et al. “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 et al. “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 et al. “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 et al. “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 et al. "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 et al. "Flexible Dynamic Information Flow Control in Haskell". In: Haskell '11. New York, NY, USA: ACM, 2011, 95–106.

[31] Deian Stefan et al. *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.