A Library for Secure Multi-threaded Information Flow in Haskell

Ta-chung Tsai Alejandro Russo John Hughes
Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden

Abstract

Li and Zdancewic have recently proposed an approach to provide information-flow security via a library rather than producing a new language from the scratch. They have shown how to implement such a library in Haskell by using arrow combinators. However, their approach only works with computations that have no side-effects. In fact, they leave as an open question how their library, and the mechanisms in it, need to be modified to consider these kind of effects. Another absent feature in the library is support for multithreaded programs. Information-flow in multithreaded programs still remains as a challenge, and no support for that has been implemented yet. It is not surprising, then, that the two main stream compilers that provide information-flow security, Jif and FlowCaml, lack support for multithreading.

Following ideas taken from literature, this paper presents an extension to Li and Zdancewic's library that provides information-flow security in presence of reference manipulation and multithreaded programs. Moreover, an online-shopping case study has been implemented to evaluate the proposed techniques. The case study reveals that exploiting concurrency to leak secrets is feasible and dangerous in practice and how our extension helps avoiding that. To the best of our knowledge, this is the first implemented tool to guarantee information-flow security in concurrent programs and the first implementation of a case study that involves concurrency and information-flow policies.

1 Introduction

Language-based information flow security aims to guarantee that programs do not leak confidential data. It is commonly achieved by some form of static analysis which rejects programs that would leak, before they are run. Over the years, a great many such systems have been presented, supporting a wide variety of programming constructs [25]. However, the impact on programming practice has been rather limited.

One possible reason is that most systems are presented in the context of a simple, elegant, and minimal language, with a well-defined semantics to make proofs of soundness possible. Yet such systems cannot immediately be adopted by programmers—they must first be embedded in a real programming language with a real compiler, which is a major task in its own right. Only two such languages have been developed—Jif [13, 14] (based on Java) and FlowCaml [18, 29] (based on Caml).

Yet when a system implementor chooses a programming language, information flow security is only one factor among many. While Jif or FlowCaml might offer the desired security guarantees, they may be unsuitable for other reasons, and thus not adopted. This motivated Li and Zdancewic to propose an alternative approach, whereby information flow security is provided via a *library* in an existing programming language [12]. Constructing such a library is a much simpler task than designing and implementing a new programming language, and moreover leaves system implementors free to choose any language for which such a library exists.

Li and Zdancewic showed how to construct such a library for the functional programming language Haskell. The library provides an abstract type of secure programs, which are composed from underlying Haskell functions using operators that impose information-flow constraints. Secure programs are *certified*, by checking that all constraints are satisfied, before the underlying functions are invoked—thus guaranteeing that no secret information leaks. While secure programs are a little more awkward to write than ordinary Haskell functions, Li and Zdancewic argue that typically only a small part of a system needs manipulate secret data—for example, an authentication module—and only this part needs be programmed using their library.

However, Li and Zdancewic's library does impose quite severe restrictions on what a secure program fragment may do. In particular, these fragments may have no effects of any sort, since the library only tracks information flow through the inputs and outputs of each fragment. While absence of

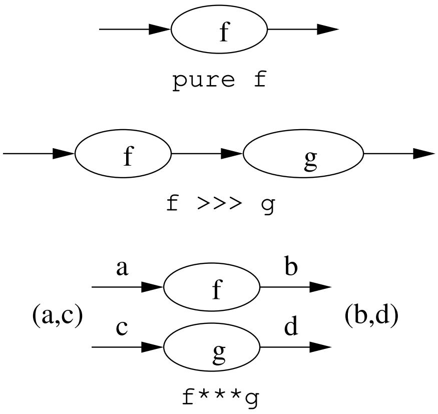

Figure 1. Basic arrow combinators.

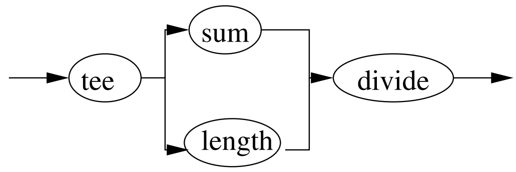

Figure 2. Average of a list.

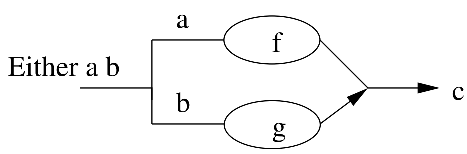

Figure 3. Choice between `f` and `g`.

side-effects can be guaranteed in Haskell (via the type system), this is still a strong restriction. Our purpose in this paper is to show that the same idea can be applied to support secure programs with a much richer set of effects—namely updateable references in the presence of (cooperative) concurrency. The underlying methods we use—an information-flow type-system for references, a restriction on the scheduler—are taken from the literature; what we show here is how to *implement* them for a real programming language following Li and Zdancewic's approach.

The rest of this paper is structured as follows. In the next section we explain Li and Zdancewic's approach in more detail. One restriction of their approach is that data-structures are assigned a *single* security level—so if any part of the output of a secure program is secret, then the entire output must be classified as secret. We need to lift this restriction in our work, allowing data-structures with mixed security levels, and in Section 3 we show how this can be achieved. This enables us to add references in Section 4. We then introduce concurrency, reviewing approaches to secure information flow in this context in Section 5, in particular ways to close the *internal timing* covert channel, and in Section 6 we describe the implementation of our chosen approach. In Section 7 we present a concurrent case study involving online shopping. With no countermeasures, an attack based on internal timing leaks can obtain a credit-card number with high probability in about two minutes. We show that our library successfully defends systems against this kind of attack. Finally, in Section 8, we draw our conclusions.

2 Encoding Information Flow in Haskell

Li and Zdancewic's approach represents secure program fragments as *arrows* in Haskell [8]. Arrows can be visualised as dataflow networks, mapping inputs on the left to outputs on the right. Arrows are constructed from Haskell functions using combinators, of which the most important are illustrated in Figure 1—`pure` converts a Haskell function to an arrow, `(>>>)` sequences two arrows, and `(***)` pairs arrows together. Any required left-to-right static dataflow can be implemented using these combinators—for example, an arrow that computes the average of a list could be constructed as

```
squareA = pure tee >>>
           (pure sum *** pure length) >>>
           pure divide
  where tee x = (x,x)
        divide (x,y) = x/y
```

Its effect is illustrated in Figure 2. To express a dynamic choice between two arrows, there is an additional combinator `f|||g`, whose input is of Haskell's sum type:

```
data Either a b = Left a | Right b
```

Its effect is illustrated in Figure 3.

Haskell allows any suitable type to be declared to be an arrow, by providing implementations for the basic arrow combinators. This is usually used to encapsulate some kind of effects. For example, we might define an arrow for programming with references, by declaring `ArrowRef a b` to be the type of arrows from `a` to `b`, implementing the basic combinators, and then providing arrows

```
createRefA :: ArrowRef a (Ref a)
readRefA   :: ArrowRef (Ref a) a
writeRefA  :: ArrowRef (Ref a,a) ()
```

to perform the basic operations on references. With these definitions, we can write side-effecting programs in a dataflow style. For example, an arrow to increment the contents of a reference could be programmed as

```
incrRefA :: ArrowRef (Ref Int) ()
incrRefA =
```

$$\frac{}{\vdash \mathsf{pure}\ f : \ell \to \ell} \qquad \frac{C_1 \vdash f : \ell_1 \to \ell_2 \quad C_2 \vdash g : \ell_3 \to \ell_4}{C_1, C_2, \ell_2 \sqsubseteq \ell_3 \vdash f\texttt{>>>}g : \ell_1 \to \ell_4}$$

Figure 4. Typing rules for `pure` and `>>>`.

```
(pure id &&& (readRefA >>> pure (+1)))
>>> writeRefA
where f &&& g = pure tee >>> (f***g)
```

Li and Zdancewic found another use for arrows: they realised that, since all the data- and control-flow in an arrow program is expressed using the arrow combinators, then they could define a type of *flow arrows*, whose primitive arrow combinators implement the type checking of an information flow type system. Their type system assigns a *security label* drawn from a suitable lattice, such as

```
data Label = LOW | MEDIUM | HIGH
  deriving (Eq, Ord)
```

to the input and output of each arrow (where the `deriving` clause declares that `LOW`$\leq$`MEDIUM`$\leq$`HIGH`). Their arrows themselves are represented by the type `FlowArrow l arr a b`, which is actually an *arrow transformer*: the type `l` is the security lattice, `a` and `b` are the input and output types, and `arr` is an *underlying arrow* type such as `ArrowRef`. Flow arrows *contain* arrows of type `arr a b`, together with flow information about their inputs and outputs.

In the information flow type system, an arrow is assigned a flow type $\ell_1 \to \ell_2$ under a set of constraints, where ℓ_1 and ℓ_2 are security labels. The rules for `pure` and `(>>>)` are given in Figure 4. The `FlowArrow` type represents not only the underlying computation, but also the information flow typing—it is represented as a record

```
data FlowArrow l arr a b = FA
  { computation :: arr a b,
    flow        :: Flow l,
    constraints :: [Constraint l]
  }
data Flow l = Trans l l | Flat
data Constraint l = LEQ l l
```

Here the `flow` component represents either $\ell_1 \to \ell_2$ (`Trans l1 l2`), or the "polymorphic" $\ell \to \ell$ for any ℓ (`Flat`), which is needed to give an accurate typing for `pure`. The `constraints` field just collects the constraints on the left of the turnstile. With this representation, it is easy to implement the typing rules in the arrow combinators. Security labels are introduced and checked by the arrow `tag l`, with flow `Trans l l`, which forces both its input and output to have the given security label.

Note that the information flow types are quite independent of the Haskell types! Moreover, they are not checked during Haskell type-checking. Rather, when a flow arrow is constructed during program execution, all the necessary constraints are collected dynamically—but they are checked before the underlying computation is run. Li and Zdancewic's library exports `FlowArrow` as an abstract type, and the only way to extract the underlying computation is via a certification function which solves the constraints first. If any constraint is not satisfied, then the underlying code is rejected.

Li and Zdancewic also considered declassification, which requires adding the user's security level as a context to the typing rules, and a new form of constraint—but we ignore the details here.

$$s^{\mathbb{L}} \ ::= \ \ell \mid (s^{\mathbb{L}}, s^{\mathbb{L}}) \mid (\mathbf{either}\ s^{\mathbb{L}}\ s^{\mathbb{L}}\ )^{\ell}$$

Figure 5. Extended security types

$$\frac{\ell_1 \leq \ell_2}{\ell_1 \sqsubseteq \ell_2} \qquad \frac{s_1^{\mathbb{L}} \sqsubseteq s_3^{\mathbb{L}} \quad s_2^{\mathbb{L}} \sqsubseteq s_4^{\mathbb{L}}}{(s_1^{\mathbb{L}}, s_2^{\mathbb{L}}) \sqsubseteq (s_3^{\mathbb{L}}, s_4^{\mathbb{L}})}$$

$$\frac{\ell_1 \sqsubseteq \ell_2 \quad s_1^{\mathbb{L}} \sqsubseteq s_3^{\mathbb{L}} \quad s_2^{\mathbb{L}} \sqsubseteq s_4^{\mathbb{L}}}{(\mathbf{either}\ s_1^{\mathbb{L}}\ s_2^{\mathbb{L}})^{\ell_1} \sqsubseteq (\mathbf{either}\ s_3^{\mathbb{L}}\ s_4^{\mathbb{L}})^{\ell_2}}$$

Figure 6. Subtyping relationship

3 Refining Security Types

Li and Zdancewic's library uses single security labels as security types. As a consequence, values are classified secrets when they contain, partially or totally, some confidential information. For instance, if one component of a pair is secret, the whole pair becomes confidential. This design decision might be a potential restriction to build some applications in practice. With this in mind, we extend Li and Zdancewic's work to include security types with more than one security label. The presence of several security labels in security types allows to develop a more precise, and consequently permissive, analysis of the information flow inside of a program.

3.1 Security Types

We assume a given security lattice $\mathbb{L}$ where security levels, denoted by ℓ, are ordered by a partial order $\leq$. Top and bottom elements are written $\top$ and $\bot$, respectively. Security types are given in Figure 5 and their subtyping relationship in Figure 6. Security type $(s^{\mathbb{L}}, s^{\mathbb{L}})$ provides security annotations for pair types. Security type $(\mathbf{either}\ s^{\mathbb{L}}\ s^{\mathbb{L}}\ )^{\ell}$ provides annotations for type `Either`. Security type ℓ decorates any other Haskell type (e.g. `Int`, `Float`, `[a]`, etc.). Security types are represented in our library as follows:

```
data SecType l
        = SecLabel l
        | SecPair (SecType l) (SecType l)
        | SecEither (SecType l) (SecType l) l
```

where `l` implements a lattice of security levels.

3.2 Defining FlowArrowRef

The abstract data type `FlowArrowRef` defines our embedded language by implementing an *arrow* interface:

```
data FlowArrowRef l a b c = FARef
     { computation  :: a b c
     , flow         :: Flow (SecType l)
     , constraints  :: [Constraint (SecType l)] }
```

This definition is similar to the definition of `FlowArrow` except for using `(SecType l)` as type argument for `Flow` and `Constraint`. Constructor `Flat` needs to be removed from data type `Flow` as a consequence of dealing with security types with more than one security label. In `FlowArrow`, `Flat` is used to establish that pure computations have the same input and output security type. Unfortunately, `Flat` cannot be used in `FlowArrowRef`, otherwise secrets might be leaked. For instance, consider the program `pure ( (x,y) -> (y,x) )` that just flips components in a pair. Assume that `x`, annotated with security label `HIGH`, is a secret input and `y`, annotated with security label `LOW`, contains public information. If `(HIGH,LOW)` is the input and output security types for that program, the value of `x` will be immediately revealed!

Similarly to Li and Zdancewic's work, `FlowArrowRef` encodes a typing judgement to verify information-flow policies. Naturally, our encoding is more complex than that in `FlowArrow`. This complexity essentially arises from considering richer security types. The typing judgment has the form: $C \vdash f \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid s_2^{\mathbb{L}}$, where f is a purely-functional computation, C is a set of constrains that, when satisfied, guarantees information-flow policies, and $\tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid s_2^{\mathbb{L}}$ is a *flow type*, which denotes that f receives input values of type τ_1 with security type $s_1^{\mathbb{L}}$, and produces output values of type τ_2 with security type $s_2^{\mathbb{L}}$. Except for combinator `pure`, most of the typing rules in Li and Zdancewic's work can be easily rewritten using this typing judgment, and therefore, we omit them here.

3.3 Security Types and Combinator `pure`

Different from Li and Zdancewick's work, it is not straightforward to determine security types for computations built with arrow combinators. Basically, the difficulty comes from deciding the output security type for combinator `pure`. This combinator can take any arbitrary Haskell function as its argument. Then, the structure of its output, and consequently its output security type, can be different in every application. For instance, output security types for `pure` computations that return numbers and pair of numbers consist of security labels and pair of security labels, respectively. Moreover, although the structure of the output security type could be determined, it is also difficult to establish the security labels appearing in it. To illustrate this point, consider the computation `pure ( \(x,y) -> (x+y, y) )`, where inputs x and y have security labels `LOW` and `HIGH`, respectively. It is clear that the output security type for this example is $(\mathtt{HIGH}, \mathtt{HIGH})$. However, in order to determine that, it is necessary to know how the input is used to build the output. This input-output dependency might be difficult to track when more complex functions are considered. With this in mind, we introduce a new security type to $s^{\mathbb{L}}$:

$$s^{\mathbb{L}} \ ::= \ \ell \mid (s^{\mathbb{L}}, s^{\mathbb{L}}) \mid (\textbf{either}\ s^{\mathbb{L}}\ s^{\mathbb{L}}\ )^{\ell} \mid \textbf{only}\ \ell$$

$$\frac{f \ :: \ \tau_1 \ \rightarrow \ \tau_2}{\emptyset \vdash \textbf{pure}\ f \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid \textbf{only}\ \mathit{join}(s_l^{\mathbb{L}})}$$

Figure 7. Typing rule for combinator `pure`

Security type **only** ℓ represents any security type that contains all their security labels as ℓ. Typing rule for `pure` is given in Figure 7. Observe the use of the Haskell typing judgment (written ::) in the hypothesis of the rule. Function $\mathit{join}(s_1^{\mathbb{L}})$ computes the join of all the security labels in $s_l^{\mathbb{L}}$. Essentially, the typing rule over-approximates the output security type by using the security labels found in the input security type. By only having one piece of secret information as input, results of `pure` computations are thus confidential regardless what they do or what kind of result they return. As a consequence, computations that follow combinator `pure` cannot operate on public data any more. As an example, consider the program `f >>> pure ( \(x,y) -> y + 1) >>> g`, where computation g operates on public data and computation `f` produces a pair where the first and second components are secret and public values, respectively. This simple program just adds one to the public output of `f` and provides that as the input of `g`. However, the program is rejected by the encoded type system in our library, even though no leaks are produced by this code. The reason for this is that program `g` receives confidential information from `pure` while it expects only public inputs. Since `pure` is responsible for allowing the use of any Haskell functions in the library, this restriction seems to be quite severe to implement concrete applications.

3.4 Combinator `lowerA`

Combinator `lowerA` is introduced to mitigate the restriction of not allowing computations on public data to take some input produced by `pure` combinators. Basically, `lowerA` takes a security label ℓ and an arrow computation p, and returns a computation p'. Computation p' behaves like p and has the same input type, output type, and input security type as p. However, its output security type might be different. The output security type is constructed based on the output type of p and it contains only security labels of value ℓ. In other words, `lowerA` downgrades the output of p to the security level ℓ. In principle, this combinator might be also used to leak secrets. An attacker can just apply `(lowerA LOW)` to every computation that involves secrets! To avoid this kind of attacks, `lowerA` filters out data with security level higher than ℓ.

Input Filtering Mechanism

Filtration of data is done by replacing some pieces of information with `undefined` [1]. This idea is implemented by the function `removeData` of the type-class `FilterData`. The signature of the type-class is the following:

```
class (Lattice l) => FilterData l t where
  removeData :: l -> t -> (SecType l) -> t
```

Method `removeData` receives a security level `l`, a value of type `t`, and a security type `(SecType l)`, and produces another value of type `t` where the information with security label higher than `l` is replaced by `undefined`. As an example, instantiations for integers and pairs are given in Figure 8. Observe how the use of type-classes allows to define different filtering policies for different kind of data. This is particularly useful when references are introduced in the language (see Section 4.6).

The introduction of undefined values might also introduce leaks due to termination. For instance, if filtered values are used inside of computations that branches on secrets, then the program might terminate (or not) depending on which branch is executed. However, these kind of leaks only reveal one bit of information about confidential data. In some scenarios, leaking one bit due to termination is acceptable and *termination-insensitive* security conditions are adopted for those cases. In fact, our library is particularly suitable to guarantee *termination-insensitive* security specifications.

Building Output Security Types

Besides introducing a filtering mechanism, `lowerA` constructs output security types where security labels are all the same. We define the following type-class:

```
class (Lattice l) => BuildSecType l t where
  buildSecType :: l -> t -> (SecType l)
```

[1] This is an undefined value in Haskell and it is member of every type.

```
instance (Lattice l)
         => FilterData l Int where
            removeData l x (SecLabel l') =
             if label_leq l' l then x
             else undefined
instance (Lattice l, FilterData l a,
          FilterData l b)
          => FilterData l (a,b) where
          removeData l (x, y) (SecPair lx ly) =
            (removeData l x lx, removeData l y ly)
```

Figure 8. Instantiations for `FilterData`

```
instance (Lattice l) =>
         BuildSecType l Int where
         buildSecType l _ = (SecLabel l)

instance
 (Lattice l, BuildSecType l a, BuildSecType l b)
 => BuildSecType l (a,b) where
    buildSecType l _ =
        (SecPair (buildSecType l (undefined::a))
                 (buildSecType l (undefined::b)))
```

Figure 9. Instantiations for `BuildSecType`

Method `buildSecType` receives a security label `l` and a value of type `t`, and produces a security type for `t` where security labels are `l`. For instance, it produces security type `(l,l)` for pair of integers. Instantiations for pairs and integers are given in Figure 9. Observe that the value of the second argument of `buildSecType` is not needed, but its type. Type-classes provide a mechanism to access information about types in Haskell and take different actions, like building different security types, depending on them.

When `lowerA` receives a computation as an argument, it needs to know its output type in order to properly apply `buildSecType`. For that purpose, we introduce another type-class:

```
class (Lattice l, Arrow a)
  => TakeOutputType l a b c where
     deriveSecType :: l -> (a b c) -> (SecType l)
```

Method `deriveSecType` receives a security label `l`, an arrow computation `(a b c)`, and returns the corresponding security type `(SecType l)` for the output type `c`. The instantiation of this type-class is shown in Figure 10.

To put it briefly, combinator `lowerA` creates a new computation that behaves as the computation received as argument, but calling the methods `removeData` and `buildSecType` in due course. The type signature for `lowerA` is given in Figure 11. Typing rule for `lowerA` is shown in Figure 12. Observe how the output security type is changed. Function ρ is defined in Figure 13 and implemented by the method `buildSecType`. As a simple example of the use of `lowerA`, we rewrite the example

```
instance
 (Lattice l, BuildSecType l c, Arrow a)
 => TakeOutputType l a b c where
 deriveSecType l ar =
    buildSecType l (undefined::c)
```

Figure 10. Instantiation for `TakeOutputType`

```
lowerA :: ( Lattice l, Arrow a,
            FilterData l b, BuildSecType l c,
            TakeOutputType l
              (FlowArrowRef l a) b c )
          => l -> FlowArrowRef l a b c ->
             FlowArrowRef l a b c
```

Figure 11. Type signature for `lowerA`

in Section 3.3 as follows: `f >>> lowerA LOW (pure (\(x,y) -> y + 1)) >>> g`. Observe that the value received by program `g` is not confidential anymore, and consequently, the program passes the type-checking tests in our library. In this example, the filtering mechanism of `lowerA` does not introduce leaks due to termination. In general, the possibilities to exploit undefined values introduced by computations like `lowerA LOW p` are related to the security of `p`. If `p` only produces `LOW` values, no leaks due to termination are introduced. Otherwise, if `p` presents, for instance, some flows from secret data to its output, a one-bit leak due to termination might happen as a price to pay for not being able to predict the input-output dependency of `p` and avoiding leaking the whole secret.

One alternative implementation to the input filter mechanism in `lowerA` ℓ p could have been to reject computation p if it takes some input with security label higher than ℓ. Unfortunately, this idea might not work properly when programs take input from external modules or components, which frequently provide data with different security levels to arrow computations. Consequently, the pattern `lowerA` ℓ (`pure` f) is particularly useful to get any values at security levels below ℓ regardless the security input type of `pure` f.

4 Adding References

Dealing with information-flow security in languages with reference manipulation is not a novelty. Unsurprisingly, Jif and FlowCaml include them as a language feature. Nevertheless, it is stated as an open question how Li and Zdancewic's library needs to be modified to consider side-effects. In particular, what arrows could be used to handle them and how their encoded type system needs to be modified. We have already started answering these question with the modification of `pure` and the introduction of `lowerA` in Section 3. We will complete answering Li and Zdancewic's questions by showing how to extend their library to introduce references. The developed techniques in this section can be considered for other kind of side-effects as well.

$$\frac{C \vdash f \;:\; \tau_1 \mid s_1^{\mathbb{L}} \;\rightarrow\; \tau_2 \mid s_2^{\mathbb{L}}}{C \vdash \texttt{lowerA}\;\ell\; f \;:\; \tau_1 \mid s_1^{\mathbb{L}} \;\rightarrow\; \tau_2 \mid \rho(\ell, \tau_2)}$$

Figure 12. Typing rule for `lowerA`

$$\frac{}{\rho(int, \ell) \;\rightarrow\; \ell}$$

$$\frac{\rho(\tau, \ell) \;\rightarrow\; s_1^{\mathbb{L}}}{\rho(\tau\; ref, \ell) \;\rightarrow\; s_1^{\mathbb{L}}\; \mathbf{ref}^{\ell}}$$

$$\frac{\rho(\tau_1, \ell) \;\rightarrow\; s_1^{\mathbb{L}} \quad \rho(\tau_2, \ell) \;\rightarrow\; s_2^{\mathbb{L}}}{\rho((\tau_1, \tau_2), \ell) \;\rightarrow\; (s_1^{\mathbb{L}}, s_2^{\mathbb{L}})}$$

$$\frac{\rho(\tau_1, \ell) \;\rightarrow\; s_1^{\mathbb{L}} \quad \rho(\tau_2, \ell) \;\rightarrow\; s_2^{\mathbb{L}}}{\rho(either\; \tau_1\; \tau_2, \ell) \;\rightarrow\; (\mathbf{either}\; s_1^{\mathbb{L}}\; s_2^{\mathbb{L}})^{\ell}}$$

Figure 13. Definition for Function ρ

4.1 Security Types for References

The treatment of references is based on Pottier and Simonet's work [18]. They introduce security types for references containing two parts: a security type and a security label. The security type provides information about the data that is referred to, while the security label gives a security level to the reference itself as a value. Following the same approach, we extend our security types as follows:

$$s^{\mathbb{L}} \;::=\; \ell \mid (s^{\mathbb{L}}, s^{\mathbb{L}}) \mid (\mathbf{either}\; s^{\mathbb{L}}\; s^{\mathbb{L}}\;)^{\ell} \mid \mathbf{only}\;\ell \mid s^{\mathbb{L}}\;\mathbf{ref}^{\ell}$$

Observe that security types for references ($s^{\mathbb{L}}\;\mathbf{ref}^{\ell}$) are composed of two parts as mentioned before. The subtyping relationship is also extended as follows:

$$\frac{s_1^{\mathbb{L}} = s_2^{\mathbb{L}} \quad \ell_1 \sqsubseteq \ell_2}{s_1^{\mathbb{L}}\;\mathbf{ref}^{\ell_1} \sqsubseteq s_2^{\mathbb{L}}\;\mathbf{ref}^{\ell_2}} \tag{1}$$

In order to avoid aliasing problems[15], this rule imposes an invariant in the subtyping relationship by requiring $s_1^{\mathbb{L}}$ to be the same as $s_2^{\mathbb{L}}$. Clearly, this invariant needs to be preserved by the arrow combinators in the library. However, `lowerA` could break that invariant! Remember that it changes every security label in the output security type of a given computation. As a consequence, we need to modify its implementation (see Section 4.2).

Data type `SecType` is extended as follows:

```
data SecType l
        = SecLabel l
        | SecPair (SecType l) (SecType l)
        | SecEither (SecType l) (SecType l) l
        | SecRef (SecType l) l
```

where `SecRef (SecType l) l` represents security types for references.

4.2 References and Combinator `lowerA`

Combinator `lowerA` could break the subtyping invariant for references described in (1). As a result, aliasing problems, and therefore leakage of secrets, might be introduced. The root of this problem comes from the fact that `lowerA` only uses output types to determine output security types. To illustrate this problem, consider a program that has two public references, `r1` and `r2`, with security type `(SecRef (SecLabel LOW) LOW)`. Assume that both references refer to the same value. If `r1`, for instance, is fed into the computation `lowerA HIGH (pure id)`, the output produced, which is obviously `r1`, will have security type `(SecRef (SecLabel HIGH) HIGH)`. Observe that the security type for the content of the reference has changed. After doing that, leaks can occur by writing secrets using `r1` and reading them out by using `r2`. Naturally, `lowerA` could also examine input security types, but unfortunately this is not enough. Once again, the difficulty to track input-output dependencies of `pure` computations (see Section 3.3) makes it difficult to determine, for instance, which reference from the input correspond to which reference in the output. Consequently, it is also difficult to determine security types for references in the output based on the input security types. To overcome this problem, we use a mechanism that can transport security information about contents of references from the input to the output of an arrow computation. In this way, `lowerA` can read this information and place the corresponding security types references when needed, and thus keep the subtyping invariant. This mechanism relies on the use of singleton types, which are the topic of the next section.

4.3 Preserving Subtyping Invariants

On one side, combinator `lowerA` builds output security types based on the output type of computations. On the other hand, security types for the content of references must never be changed. So, why not encoding in the Haskell type system the security type for the content of references? Hence, `lowerA` can take the encoded information and precisely determines the corresponding security type for the content of each reference.

Singleton types [16] are adequate to represent specific values at the level of types. Essentially, they allow to have a match between values and types and vice versa. Our goal is, therefore, to encode values of type `(SecType l)` in more fine-grained Haskell types. For instance, the encoding for values of type `(SecType Label)` can be done as follows:

```
data SLow    = VLow
data SMedium = VMedium
data SHigh   = VHigh

data SSecLabel  lb        = VSecLabel  lb
data SSecPair   st1 st2   = VSecPair   st1 st2
data SSecEither st1 st2 lb= VSecEither st1 st2 lb
data SSecRef    st  lb    = VSecRef    st  lb
```

Observe how one type has been introduced for each constructor appearing in `Label` and `SecType`. With this encoding, we can now represent security types in the Haskell type system. As an example, security type `(SecRef (SecLabel HIGH) LOW)` can be encoded using the value `(VSecRef (VSecLabel VHigh) VLow)` of type `(SSecRef (SSecLabel SHigh) SLow)`.

As mentioned before, `lowerA` should use the encoded information to place the corresponding security types for content of references. In order to achieve that, we need a mapping from singleton types to values of type `(SecType l)`. The following code implements that:

```
class STLabel lb l where
  toLabel :: lb -> l

instance STLabel SLow Label where
  toLabel _ = LOW
instance STLabel SMedium Label where
  toLabel _ = MEDIUM
instance STLabel SHigh Label where
  toLabel _ = HIGH

class STSecType st l where
  toSecType :: st -> SecType l

instance STLabel lb l
  => STSecType (SSecLabel lb) l where
  toSecType _
    = SecLabel (toLabel (undefined::lb))
instance (STSecType st l, STLabel lb l)
  => STSecType (SSecRef st lb) l where
  toSecType _
    = SecRef (toSecType (undefined::st))
             (toLabel (undefined::lb))
instance (STSecType st1 l, STSecType st2 l)
  => STSecType (SSecPair st1 st2) l where
  toSecType _
    = SecPair (toSecType (undefined::st1))
              (toSecType (undefined::st2))
instance (STSecType st1 l,
          STSecType st2 l, STLabel lb l)
  => STSecType (SSecEither st1 st2 lb) l where
  toSecType _
    = SecEither (toSecType (undefined::st1))
                (toSecType (undefined::st2))
                (toLabel (undefined::lb))
```

Functions `toLabel` and `toSecType` return security labels and security types based on singleton types, respectively.

Having our encoding ready, we introduce references as values of the data type: `data Ref st a = Ref st`

(IORef a), where (IORef a) is the type for references in Haskell and st is a singleton type encoding the security type for its content. At this point, we are in conditions to extend the function buildSecType, used by lowerA, to build output security types:

```
instance (Lattice l, STSecType st l)
   => BuildSecType SecType l (SRef st a) where
  buildSecType l _
    = (SecRef (toSecType (undefined::st)) l)
```

Observe how buildSecType calls toSecType to build the security type for the content of the reference by passing an undefined value of singleton type st. The subtyping invariant is now preserved by lowerA. In fact, this technique can be used to preserve any subtyping invariant required in the library.

4.4 Reference Manipulation

Li and Zdancewic's library uses the *underlying arrow* (->) to perform computations. However, we need to modify that in order to include side-effects produced by references. The following data type defines the *underlying arrow* used in our library: `data ArrowRef a b = a -> IO b`. *Underlying computations* can therefore take an argument of type a and return a value of type (IO b), which probably produces some side-effects related to references.

Three primitives are respectively provided to create, read, and write references: createRefA, readRefA, and writeRefA. Basically, these functions lift the traditional Haskell operations to manipulate references into FlowArrowRef, but performing some checking related to information-flow security (see Section 4.5). However, from a programmer's point of view, they look similar to any primitives that deal with references. For instance, createRefA has the following signature:

```
createRefA :: (Lattice l, STSecType st l,
               BuildSecType l a) =>
  st -> l -> FlowArrowRef l ArrowRef a (Ref st a)
```

where singleton type st encodes the security type for the content of the reference, and l is the security level of the reference as a value. Observe that ArrowRef is used for the underlying computation. As an example, (createRefA (VSecLabel VHigh) LOW) returns a computation that creates a public reference to a secret value received as argument. This is the only primitive where programmers must use singleton types and where the library exploits the correspondence between values and types. Because of that, it could be possible to remove the argument st from createRefA to make its type signature simpler. However, by doing that, programmers would need to explicitly specify the type for every occurrences of createRefA with their corresponding (STSecType st l) and (Ref st a).

$$\frac{}{e(\ell) \rightarrow \ell} \qquad \frac{e(s_1^{\mathbb{L}}) \rightarrow \ell_1 \quad e(s_2^{\mathbb{L}}) \rightarrow \ell_2}{e((s_1^{\mathbb{L}}, s_2^{\mathbb{L}})) \rightarrow \ell_1 \sqcup \ell_2}$$

$$\frac{}{e((\mathbf{either}\ s_1^{\mathbb{L}}\ s_2^{\mathbb{L}})^{\ell}) \rightarrow \ell} \qquad \frac{}{e(\mathbf{only}\ \ell) \rightarrow \ell}$$

$$\frac{}{e(s^{\mathbb{L}}\ \mathbf{ref}^{\ell}) \rightarrow \ell}$$

Figure 14. Definition for Function e

4.5 Typing Rules for Reference Primitives

Pottier and Simonet present a type-based information flow analysis for CoreML equipped with references, exceptions and let-polymorphism [18]. Particularly, their type system is constraint-based and uses effects to deal with references. We restate some of their ideas in the framework of our library. More precisely, we adapt our encoded type-checker to include effects and consequently involve references.

We enhance the typing judgement introduced in Section 3.2 as follows: $pc\,, C \vdash f\ :\ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow\ \tau_2 \mid s_2^{\mathbb{L}}$, where the new parameter, *pc*, is a lower bound on the security level of the memory cell that is written. In Figure 15, we show how typing rules for pure, sequential, and branching computations are rewritten using this new parameter. Typing rules for other combinators are adapted similarly. Rule (PURE) produces no side-effects and therefore it imposes no lower bounds in *pc*. Rule (SEQ) takes the meet of the lower bounds for side-effects as the new *pc*. Rule (CHOICE) essentially requires that the branching computation does not produce side-effects or results that are below the guard of the branch, which has type *either* $\tau_1\ \tau_3$. These requirements are enforced by $(\mathbf{either} s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell} \blacktriangleleft (pc_1 \sqcap pc_2)$ and $(\mathbf{either}\ s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell} \blacktriangleleft e(\uparrow (s_2^{\mathbb{L}} \sqcup s_4^{\mathbb{L}}, \ell))$, respectively. As defined in Simonet and Pottier's work, constraint $s^{\mathbb{L}} \blacktriangleleft \ell$ imposes ℓ as an upper bound for every security label in $s^{\mathbb{L}}$. Function e determines the security level of a given value (see Figure 14). Operator $\uparrow$ lifts security labels that are below certain security level, but not violating subtyping invariants (see Figure 16).

Typing rules for references are introduced in Figure 17. Singleton type $\mathbf{s}^{\mathbb{L}}$ encodes the security type $s^{\mathbb{L}}$ and is generated by the value $(\mathbf{s}^{\mathbb{L}})_v$. Rule (CREATE) requires that the singleton type passed as argument matches the input security type. Otherwise, programmers could introduce inconsistencies in the type-checking process. The side-effect produced by creation of references is allocation of memory. Therefore, the pc is related with the security level of the content of the created reference ($e(s_1^{\mathbb{L}})$). Rule (READ) lifts security labels in the output security type considering the security level of the reference ($\uparrow (s_1^{\mathbb{L}}, \ell)$). Rule (WRITE)

$$(PURE)\frac{f \ : \ \tau_1 \ \rightarrow \ \tau_2}{\top, \emptyset \vdash \textbf{pure}\ f \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid \textbf{only}\ \ell}$$

$$(SEQ)\frac{pc_1, C_1, \vdash f_1 \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid s_2^{\mathbb{L}} \quad pc_2, C_2 \vdash f_2 \ : \ \tau_2 \mid s_3^{\mathbb{L}} \ \rightarrow \ \tau_4 \mid s_4^{\mathbb{L}}}{pc_1 \sqcap pc_2, C_1 \cup C_2 \cup \{s_2^{\mathbb{L}} \sqsubseteq s_3^{\mathbb{L}}\} \vdash f_1 \ \ggg \ f_2 \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_4 \mid s_4^{\mathbb{L}}}$$

$$(CHOICE)\frac{pc_1, C_1 \vdash f_1 \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid s_2^{\mathbb{L}} \quad pc_2, C_2, \vdash f_2 \ : \ \tau_3 \mid s_3^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid s_4^{\mathbb{L}}}{pc_1 \sqcap pc_2, C_1 \cup C_2 \cup C_3 \vdash f_1 \ ||| \ f_2 \ : \ flow}$$

$$flow \ = \ either\ \tau_1\ \tau_3 \mid (\textbf{either}\ s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell} \ \rightarrow \ \tau_2 \mid \uparrow (s_2^{\mathbb{L}} \sqcup s_4^{\mathbb{L}}, \ell)$$

$$C_3 = \{(\textbf{either} s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell} \blacktriangleleft (pc_1 \sqcap pc_2), (\textbf{either}\ s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell} \blacktriangleleft e(\uparrow (s_2^{\mathbb{L}} \sqcup s_4^{\mathbb{L}}, \ell))\}$$

Figure 15. Typing rules for pure, sequential composition, and choice combinators

$$\frac{\ell_1 \sqsubseteq \ell_2}{\uparrow \ \ell_1\ \ell_2 \ \rightarrow \ \ell_2} \qquad \frac{\ell_2 \sqsubset \ell_1}{\uparrow \ \ell_1\ \ell_2 \ \rightarrow \ \ell_1} \qquad \frac{\ell_1 \sqsubseteq \ell_2}{\uparrow \ (s^{\mathbb{L}}\ \textbf{ref}^{\ell_1})\ \ell_2 \ \rightarrow \ s^{\mathbb{L}}\ \textbf{ref}^{\ell_2}} \qquad \frac{\ell_2 \sqsubset \ell_1}{\uparrow \ (s^{\mathbb{L}}\ \textbf{ref}^{\ell_1})\ \ell_2 \ \rightarrow \ s^{\mathbb{L}}\ \textbf{ref}^{\ell_1}}$$

$$\frac{\uparrow \ s_1^{\mathbb{L}}\ \ell \ \rightarrow \ s_3^{\mathbb{L}} \quad \uparrow \ s_2^{\mathbb{L}}\ \ell \ \rightarrow \ s_4^{\mathbb{L}}}{\uparrow \ (s_1^{\mathbb{L}}, s_2^{\mathbb{L}})\ \ell \ \rightarrow \ (s_3^{\mathbb{L}}, s_4^{\mathbb{L}})} \qquad \frac{\ell_1 \sqsubseteq \ell_2 \quad \uparrow \ s_1^{\mathbb{L}}\ \ell_2 \ \rightarrow \ s_3^{\mathbb{L}} \quad \uparrow \ s_2^{\mathbb{L}}\ \ell_2 \ \rightarrow \ s_4^{\mathbb{L}}}{\uparrow \ (\textbf{either}\ s_1^{\mathbb{L}}\ s_2^{\mathbb{L}})^{\ell_1}\ \ell_2 \ \rightarrow \ (\textbf{either}\ s_3^{\mathbb{L}}\ s_4^{\mathbb{L}})^{\ell_2}}$$

$$\frac{\ell_2 \sqsubset \ell_1 \quad \uparrow \ s_1^{\mathbb{L}}\ \ell_2 \ \rightarrow \ s_3^{\mathbb{L}} \quad \uparrow \ s_2^{\mathbb{L}}\ \ell_2 \ \rightarrow \ s_4^{\mathbb{L}}}{\uparrow \ (\textbf{either}\ s_1^{\mathbb{L}}\ s_2^{\mathbb{L}})^{\ell_1}\ \ell_2 \ \rightarrow \ (\textbf{either}\ s_3^{\mathbb{L}}\ s_4^{\mathbb{L}})^{\ell_1}} \qquad \frac{\ell_1 \sqsubseteq \ell_2}{\uparrow \ (\textbf{only}\ \ell_1)\ \ell_2 \ \rightarrow \ \textbf{only}\ \ell_2} \qquad \frac{\ell_2 \sqsubset \ell_1}{\uparrow \ (\textbf{only}\ \ell_1)\ \ell_2 \ \rightarrow \ \textbf{only}\ \ell_1}$$

Figure 16. Definition for Function $\uparrow$

imposes the constraint $\ell \lhd s^{\mathbb{L}}$. Similarly to Simonet and Pottier's work, constraint $\ell \lhd s^{\mathbb{L}}$ requires $s^{\mathbb{L}}$ to have security level ℓ or greater, and is used to record a potential information flow.

We modify the implementation of the type-system in our library to include effects. Consequently, data type `FlowArrowRef` is extended with a new field called `pc` to represent lower bounds for side-effects as explained above. Data type `Constraint` is also extended to involve operators $\lhd$ and $\blacktriangleleft$. Moreover, we add unification mechanisms inside of arrow combinators to pass information about security types when needed. As a consequence, a few security annotations need to be provided by programmers. Li and Zdancewic's library does not need this feature since their security types are very simple. One of the interesting aspect of implementing unification inside of arrows is the generation of fresh names. Our library generates fresh names by applying renaming functions when arrow combinators are applied, but we omit the details here due to lack of space.

4.6 Filtering References

References introduce the possibility of having shared resources in programs. In Section 3.4, the filtering mechanism replaces some pieces of information with `undefined`. However, it is not a good idea to replace the content of some reference with `undefined` since it might be used by other parts (or threads) in the program. We still need to restrict the access to that content somehow. In order to do that, we introduce projection functions for each reference handled by the library. Projection functions are basically functions that return values less informative than their arguments. The concept of projection functions has been indirectly used in semantic models for information-flow security [10, 26]. The instance for references of the method `removeData` creates projection functions that, when applied to the contents of their associated references, return values where some information higher than some security level is replaced by `undefined`. However, the content of the reference itself is not modified. Observe that the filtering principle applied by projection functions and `removeData` is the same. Combinator `readRefA` is also modified to return the content of the reference by firstly passing it through its corresponding projection function. Due to lack of space, we omit the implementation of these ideas here.

5 Information Flow in a Concurrent Setting

Concurrency introduces new covert channels, or unintended ways, to leak secret information to an attacker. As a consequence, the traditional techniques to enforce information flow policies in sequential programs are not sufficient for multithreaded languages [32]. One particularly dangerous covert channel is called *internal timing*. It allows to leak information when secrets affect the timing behavior of

$$(CREATE)\frac{}{e(s^{\mathbb{L}}_{1}), \emptyset \vdash \texttt{createRefA}\ (\mathbf{s^{\mathbb{L}}_{1}})_v\ \ell\ :\ \tau \mid s^{\mathbb{L}}_{1}\ \rightarrow\ \tau\ ref \mid s^{\mathbb{L}}_{1}\ \mathbf{ref}^{\ell}}$$

$$(READ)\frac{}{\top, \emptyset \vdash \texttt{readRefA} : \tau\ ref \mid s^{\mathbb{L}}_{1}\ \mathbf{ref}^{\ell}\ \rightarrow\ \tau \mid \uparrow (s^{\mathbb{L}}_{1}, \ell)}$$

$$(WRITE)\frac{}{e(s^{\mathbb{L}}), \{\ell \lhd s^{\mathbb{L}}\} \vdash \texttt{writeRefA} : (\tau\ ref, \tau) \mid (s^{\mathbb{L}}\ \mathbf{ref}^{\ell}, s^{\mathbb{L}})\ \rightarrow\ () \mid \bot}$$

Figure 17. Typing rules for reference primitives

a thread, which via the scheduler, affects the order in which public computations occur. Consider the following two imperative programs running in two different threads:

$$t_1 : (\texttt{if h} > 0\ \texttt{then skip}(120)\ \texttt{else skip}(1));\ \texttt{l} := 1$$
$$t_2 : \texttt{skip}(60);\ \texttt{l} := 0 \qquad (2)$$

Variables h and l store secret and public information, respectively. Assume $\texttt{skip}(n)$ executes n consecutive *skip* commands. Notice that both t_1 and t_2 are secure in isolation under the notion of *noninterference* [25]. However, by running them in parallel, it is possible to leak information about h. To illustrate that, we assume an scheduler with time slice of 80 steps that always starts by running t_1. On one hand, if $\texttt{h} > 0$, t_1 will run for 80 steps, and while being running $\texttt{skip}(120)$, t_2 is scheduled and run until completion. Then, the control is given again to t_1, which completes its execution. The final value of `l` is 1. On the other hand, if $\texttt{h} \leq 0$, t_1 finishes first its execution. After that, t_2 is scheduled and run until completion. In this case, the final value of `l` is 0. An attacker can, therefore, deduce if $\texttt{h} > 0$ (or not) by observing the final value of `l`. Different from the *external timing* covert channel, the attacker does not need to observe the actual execution time of a program in order to deduce some secret information. Moreover, internal timing leaks can also be magnified via loops, where each iteration of the loop can leak one bit of the secret. Hence, entire secret values can be leaked.

There are several existing approaches to tackling internal timing flows. Several works by Volpano and Smith [32, 35, 30, 31] propose a special primitive called `protect`. By definition, $\texttt{protect}(c)$ takes one atomic step in the semantics with the effect of executing c to the end. Internal timing leaks are removed if every computation that branches on secrets is wrapped by `protect()` commands. However, implementing `protect` imposes a major challenge [27, 23, 20] (except for cooperative schedulers [21]). These proposals rely on the modification of the run-time environment or the assumption of randomized schedulers, which are rarely found in practice. Russo et al. [19] propose a transformation to close internal timing channels that does not require the modification of the run-time environment. The transformation works for programs that run under a wide class of round-robin schedulers and only rejects those ones that have symptoms of illegal flows inherent from sequential settings. Boudol and Castellani [2, 3] propose type systems for languages that do not rely on the `protect` primitive. However, they reject programs with assignments to low variables after some computation that branches on secrets. Internal timing problem can also be solved by considering external timing. Definitions related to external timing involve stronger attackers. As expected, an stronger attacker model imposes more restriction on programs. For instance, loops branching on secrets are disallowed. There are several works on that direction [1, 27, 23, 24, 11]. Zdancewic and Myers [37] prevent internal timing leaks by disallowing races on public data. However, their approach rejects innocent secure programs like $l := 0 \parallel l := 1$ where l is a public variable. Recently, Huisman et al. [9] improved Zdancewic and Myers' work by using logic-based characterizations and well known model checking techniques. Several proposals have been explored in process-calculus settings [6, 4, 22, 7, 17], but without considering the impact of scheduling.

The referred works above have neglected to consider implementing case studies where the proposed enforcement mechanisms are applied. This work presents, to the best of our knowledge, the first concrete implementation of a case study that consider information-flow policies in presence of concurrency.

6 Closing Internal Timing Channels

We incorporate a run-time mechanism to close internal timing covert channels in our library. We base our approach in a combination of ideas taken from the literature. On one hand, Russo and Sabelfeld [21] show how to implement `protect()` for cooperative schedulers. Essentially, their work states that threads must not yield control inside of computations that branch on secrets. Russo et al. [19], on the other hand, express that a class of round-robin schedulers does not suffer from leaks due to dynamic thread creation. As a consequence, creation of threads can be allowed at any point in programs. By mixing these two ideas, we modify the *underlying* arrow combinators in order to implement a cooperative round-robin scheduler and to guarantee that computations branching on secrets do not yield control when running. In this way, internal timing leaks are removed from programs and a flexible treatment for dynamic thread creation is also obtained. In fact, the introduced modifications are completely independent to the encoded type system described in Section 3 and 4.

Cooperative schedulers are based on yielding control when programs indicate that. On the other hand, programs are written using arrow combinators, which can be seen as a kind of *building blocks*. In our library, *simple* arrow combinators yield control after finishing their execution if they are not part of computations that branch on secrets. Example of such combinators are `pure`, `createRef`, `readRef`, and `writeRef`. Computations branching on secrets do not yield control regardless how many building blocks compose them. As result, *simple* arrow combinators and computations that branch on secrets are atomic computational units involved in interleavings. The round-robin scheduler is obtained by yielding control in a particular way.

Concurrency is introduced in our implementation by importing the Haskell module `Control.Concurrent` [28, 33] . This module provides dynamic thread creation and pre-emptive concurrency. Since threads can be scheduled anytime, some synchronization is needed to restrict their execution as round-robin. Software transactional memory(STM) [5] provides easy-to-reason and simple primitives to do that. We could have chosen more standard primitives like semaphores or `MVar` [28]. However, the obtained code would have been more complicated.

We start introducing information about scheduling on the *underlying* arrow `ArrowRef`:

```
data RRobin a = RRobin
   { data  :: a, iD :: ThreadId,
     queue :: TVar [ThreadId], blocks :: Int }

data ArrowRef a b
     = AR ((RRobin a) -> IO (RRobin b))
```

Data type `(RRobin a)` stores information related to scheduling in the input and output values of arrows. Field `data` stores the input data for the arrow. Field `iD` stores the thread identification number where the arrow computation is executed. Field `queue` stores a round-robin list of threads identifiers and its access is protected by a mutex (`TVar [ThreadId]`). The list is updated when creation or termination of threads occur. Field `blocks` indicates if the thread executing the arrow computation must wait for its turn to run and then, when finishing, yields the control to another thread. This field plays an essential rôle to guarantee atomic execution of computations that branch on secrets.

We introduce two new combinators in the underlying arrow: `waitForYield` and `yieldControl`. Essentially, these combinators are responsible for implementing a round-robin scheduler. Combinator `waitForYield` blocks until the content of the head of the round-robin queue (`TVar [ThreadId]`) is the same as the thread identification (`iD`) running this combinator. Combinator `yieldControl` removes the head of the round-robin

```
waitTurn :: RRobin a -> IO ()
waitTurn sch = if (blocks sch) > 0 then return ()
               else atomically (
                     do q <- readTVar (queue sch)
                        if head q /= (iD sch)
                        then retry
                        else return ())

waitForYield :: ArrowRef a a
waitForYield = AR (\sch -> do waitTurn sch
                              return sch)

nextTurn :: RRobin a -> IO ()
nextTurn sch
  = if (blocks sch) > 0 then return ()
    else atomically (
           do q <- readTVar (queue sch)
              writeTVar (queue sch)
                        ((tail q)++[head q])
              return () )

yieldControl :: ArrowRef a a
yieldControl = AR (\sch -> do nextTurn sch
                              return sch)
```

Figure 18. Primitives for yielding control

```
beginAtomic :: ArrowRef a a
beginAtomic
  = waitForYield >>>
    AR (\sch -> return sch {blocks =
                             ((blocks sch)+1)} )

endAtomic :: ArrowRef a a
endAtomic
  = AR (\sch -> return sch {blocks =
                             ((blocks sch)-1)})
    >>> yieldControl
```

Figure 19. Primitives for atomicity

queue and put it as the last element. Both combinators have no computational effects if the field `blocks` is different from zero. The implementation of these combinators is shown in Figure 18. Function `atomically` guarantees mutual exclusion access to the round-robin queue. Function `retry` blocks the thread until `queue` changes its value. When this happens, it resumes its execution from the first command wrapped by `atomically`. It is important to remark that combinators in the underlying arrow are not accessible for users of the library.

Simple arrow combinators include now `waitForYield` and `yieldControl` before and after finishing their computations, respectively. Nevertheless, combinators related with branches are threaded differently. Computations that branch on secrets must not yield control until finishing their execution. Branching combinators, like (|||), can be applied to arrow computations that involve `yieldControl` in their bodies. As a consequence,

$$\frac{pc, C \vdash f \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid s_2^{\mathbb{L}}}{pc, C \vdash \texttt{forkRef}\ f \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ () \mid \bot}$$

Figure 20. Typing rule for `forkRef`

when the guard of the branch involves some secrets, these combinators must no yield control to other threads. We introduce two more combinators to the *underlying arrow*: `beginAtomic` and `endAtomic`. When placed like `beginAtomic >>> f >>> endAtomic`, they leave without any effect the combinators `waitForYield` and `yieldControl` appearing in `f`. Therefore, program `f` executes until completion without yielding control to other threads. We then modify the implementation of combinators related with branchings in order to include `beginAtomic` and `endAtomic` when the condition of the branch depends on secrets. The Implementation of `beginAtomic` and `endAtomic` is given in Figure 19. Observe that `beginAtomic` and `endAtomic` count how many computations branching on secret are nested. Combinators `waitForYield`, `yieldControl`, `beginAtomic` and `endAtomic` need to be pairwise to properly work.

Dynamic thread creation is introduced by the new arrow combinator `forkRef`. It takes a computation as argument and spawns it in a new thread with an exception handler. If the new thread raises an exception, the handler forces all the program to finish, reducing the bandwidth of leakings due to no termination. The typing rule for `forkRef` is shown in Figure 20. Observe that the returned value of f is discarded since f will be run in another thread.

7 Case Study: Online Shopping

In order to evaluate the flexibility of the arrow combinators and techniques proposed in Sections 3, 4, and 6, we implemented a case study of an online shopping server. Basically, the server processes transactions related to buying products. It receives information from the network and spawn different threads to perform purchases for each client. For simplicity, we assume that there is only one product to buy and that the only information provided by clients are their names, billing addresses, and credit card numbers composed of 16 digits. We also assume that there are security levels `HIGH` and `LOW` for secret and public information, respectively. Our library guarantees, in this example, that the confidentiality of credit card numbers is preserved.

The server program consists of three components: `protectData`, `purchase`, and `showPurchase`. Component `protectData` receives information from clients and determines that credit card numbers are the only secrets in the system. The implementation of `protectData` consists of a few lines of code that apply combinator `tag` to its input. We consider this component as part of the trusted computing based. Component `purchase` simulates buying products. Moreover, it copies the client credit card number and the rest of his/her information into two different databases, respectively. We simulate the access to these databases with references to different lists of data. Component `showPurchase` retrieves information from the database with public information and shows it on the screen (a public channel).

The online shopping server can be modified to execute malicious code that exploits internal timing covert channels. An attack similar to (2) can be implemented if no countermeasures are taken. However, such an attack only reveals one bit of the secret. In order for the attacker to obtain complete credit card numbers, it is necessary to magnify the attack by introducing a loop. Then, one bit of the secret is leaked in each iteration of the loop. The implementation of this attack reveals a credit card number in about two minutes [2]. Surprisingly, it was quite straightforward to leak the sixteen digits of a credit card number even though we have no information about the run-time environment. This shows how feasible and dangerous are internal timing leaks in practice.

Our malicious code concatenates credit card numbers after the billing addresses of clients. Thus, credit card numbers can be displayed on the screen by just invoking `showPurchase`. To illustrate that, we consider a client with the credit card number `9999999999999999`. We run the attack several times obtaining different leaked credit card numbers (see Figure 21). These numbers differ in at most three bits from the binary representation of the secret. This imprecision comes from the lack of knowledge about the run-time environment, in particular, the lack of knowledge about scheduler policies. Scheduler policies are important for an internal timing attack to succeed. Nevertheless, by repeatedly running the attack and taking the most frequent boolean values in each position, it is possible to obtain the value of the secret with very high confidence. Observe that the secret and the inferred secret are the same in Figure 21.

We repeatedly ran the malicious code mentioned above but with the countermeasures described in Section 6. In this opportunity, the leaked credit card number was always 0. In other words, the attack did not succeed. There is an obvious overhead introduced by restricting the scheduler in the run-time environment to behave like a round-robin one. However, this is acceptable since only small parts of a system need to manipulate secrets and therefore be written using our library.

[2] Every experiment was run on a laptop Pentium M 1.5 GHz and 512 MB RAM.

Secret: 1000111000011011110010011011111100000011111111111111

Run No.	**Leaked credit card number**	**Time(Sec.)**
1	1010111110011111111110111011111100000111111111111111	27
2	1100111000011011110111011011110100000011111111111111	27
3	1010111000011011111010011011111100000011111111111111	28
4	1000111001011011110010011011111100000011111111111011	28
5	1000111000011011110010011011111101000011111111111111	29

Inferred Secret: 1000111000011011110010011011111100000011111111111111

Figure 21. Results produced by the malicious code

8 Conclusions

We have presented an extension to Li and Zdancewic's library to consider secure programs with reference manipulation and concurrency. Moreover, a case study has been implemented to evaluate the techniques proposed in this work. It reveals that internal-timing leaks are feasible and dangerous in practice and shows how our library properly repairs them. To the best of our knowledge, this is the first tool that supports information-flow security and concurrency, and the first case study implemented that involves concurrent programs and information-flow policies.

The extension to the library consists on two different parts. On one hand, we include reference manipulation by considering richer security types than those appearing in Li and Zdancewic's work. This consideration allows us to describe a more precise analysis for information-flow security than that obtained by just considering single security labels as security types. In order to get such analysis, we combine several ideas from the literature in our implementation: singleton types, type-classes in Haskell, and projection functions. On the other hand, supporting concurrency requires dealing with internal-timing attacks. The extension includes a mechanism to close internal-timing covert channels and provides a flexible treatment for dynamic thread creation. Therefore, it is not necessary to modify the run-time environment to obtain secure programs. These achievements are also result of combining several ideas from the literature: round-robin cooperative schedulers and software transactional memories. Similarly to Li and Zdancewic's work, the technical development in this paper is informal, although we have implemented it in Haskell. The type system encoded in `FlowArrowRef` can be mainly justified by following standard techniques to prove non-interference properties [36, 18]. The implementation of the library and the case study is publicly available in [34].

Acknowledgments We wish to thank to the anonymous reviewers and our colleagues in the ProSec group at Chalmers for helpful feedback. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

[1] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, Jan. 2000.

[2] G. Boudol and I. Castellani. Noninterference for concurrent programs. In *Proc. ICALP'01*, volume 2076 of *LNCS*, pages 382–395. Springer-Verlag, July 2001.

[3] G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, June 2002.

[4] R. Focardi and R. Gorrieri. Classification of security properties (part I: Information flow). In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 331–396. Springer-Verlag, 2001.

[5] T. Harris, M. Herlihy, S. Marlow, and S. P. Jones. Composable memory transactions. In *Proc. ACM Symp. on Principles and Practice of Parallel Programming, to appear*, June 2005.

[6] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. European Symp. on Programming*, volume 1782 of *LNCS*, pages 180–199. Springer-Verlag, 2000.

[7] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 81–92, Jan. 2002.

[8] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[9] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. IEEE Computer Security Foundations Workshop*, July 2006.

[10] S. Hunt. Pers generalise projections for strictness analysis (extended abstract). In *Proc. 1990 Glasgow Workshop on Functional Programming*, Workshops in Computing, Ullapool, 1991. Springer-Verlag.

[11] B. Köpf and H. Mantel. Eliminating implicit information leaks by transformational typing and unification. In

FAST'05, volume 3866 of *LNCS*. Springer-Verlag, July 2006.

[12] P. Li and S. Zdancewic. Encoding information flow in haskell. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

[13] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, Jan. 1999.

[14] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. `http://www.cs.cornell.edu/jif`, July 2001–2006.

[15] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[16] B. C. Pierce. *Advanced Topics In Types And Programming Languages*. MIT Press, November 2004.

[17] F. Pottier. A simple view of type-secure information flow in the pi-calculus. In *Proc. IEEE Computer Security Foundations Workshop*, pages 320–330, June 2002.

[18] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, Jan. 2002.

[19] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Proc. Annual Asian Computing Science Conference*, LNCS, Dec. 2006.

[20] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop*, pages 177–189, July 2006.

[21] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. PSI'06*, LNCS. Springer-Verlag, June 2006.

[22] P. Ryan. Mathematical models of computer security—tutorial lectures. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 1–62. Springer-Verlag, 2001.

[23] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.

[24] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, Sept. 2002.

[25] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[26] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proc. European Symp. on Programming*, volume 1576 of *LNCS*, pages 40–58. Springer-Verlag, Mar. 1999.

[27] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[28] A. G. Simon Peyton Jones and S. Finne. Concurrent haskell. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1996.

[29] V. Simonet. Flow caml in a nutshell. In *Graham Hutton, editor, Proceedings of the first APPSEM-II workshop*, pages 152–165, Mar. 2003.

[30] G. Smith. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.

[31] G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proc. IEEE Computer Security Foundations Workshop*, pages 3–13, 2003.

[32] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[33] The GHC Team. The glasgow haskell compiler. Software release. `http://haskell.org/ghc/`.

[34] T. C. Tsai and A. Russo. A library for secure multi-threaded information flow in haskell. Software release. Available at `http://www.cs.chalmers.se/∼russo/tsai.htm`.

[35] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, Nov. 1999.

[36] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[37] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.