Encoding Information Flow in Haskell

Peng Li
University of Pennsylvania
lipeng@cis.upenn.edu

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

Abstract

This paper presents an embedded security sublanguage for enforcing information-flow policies in the standard Haskell programming language. The sublanguage provides useful information-flow control mechanisms including dynamic security lattices, run-time code privileges and declassification, without modifying the base language. This design avoids the redundant work of producing new languages, lowers the threshold for adopting security-typed languages, and also provides great flexibility and modularity for using security-policy frameworks.

The embedded security sublanguage is designed using a standard combinator interface called arrows*. Computations constructed in the sublanguage have static and explicit control-flow components, making it possible to implement information-flow control using static-analysis techniques at* run time*, while providing strong security guarantees. This paper presents a concrete Haskell implementation and an example application demonstrating the proposed techniques.*

1. Introduction

Language-based information-flow security has a long, rich history with many (mostly theoretical) results [11]. This prior work has focused mainly on the problems of static program analysis for a wide variety of computation models and policy features. Often these analyses are presented as type systems whose soundness is justified by some form of *noninterference* result. The approach is compelling because programming-language techniques can be used to specify and enforce security policies that cannot be achieved by conventional mechanisms such as access control and encryption. Two full-scale language implementations have been developed: Jif [6, 4] by Myers *et al.* is a variant of Java, and Flow Caml [13, 10] by Simonet *et al.* is an extension of Caml.

However, despite this rather large (and growing!) body of work on language-based information-flow security, there has been relatively little adoption of the proposed techniques—two success stories are the "taint-checking mode" available in the Perl language and the use of information-flow analysis to separate low-integrity components from high integrity components built in the SparkAda [3] language.

One important reason why these domain-specific security-typed languages have not been widely applied is that, because the information-flow policies are intended to apply in an end-to-end fashion, the whole system has to be written in the new language. However, it is expensive to build large software systems in a new language. Doing so can be justified only if the benefit of using the new language outweighs the cost of migrating to the new language—including the costs of retraining programmers and the time and expense necessary to port existing libraries and other code bases.

Moreover, in practice, it may well be the case that only a small part of the system (maybe only a few variables in a large program) has information-flow security requirements. Although the system may be large and complex, the secret information flow in the system may not be completely unmanageable. In such cases, it is probably more convenient to use the programming language that best fits the primary functionality of the system rather than its security requirements, and manage security issues by traditional means such as code auditing and careful software engineering practices. There is a language adoption threshold based on the ratio of *security requirements* to *functionality requirements*, and this threshold is very high.

1.1. Embedded security sublanguages

This paper presents a different approach to enforcing information-flow security policies. Rather than producing a new language from scratch, we show how to encode traditional information-flow type systems using the general features of existing, modern programming languages. In particular, we show how the *abstract datatype* and *typeclass* features found in the general-purpose language Haskell can be used to build a module that effectively provides a security-typed sublanguage embedded in Haskell itself. This sublanguage can interoperate smoothly with existing Haskell code while still providing strong information-flow security guar-

antees. Importantly, we do not need to modify the design or implementation of Haskell itself—we use features in its standard (but advanced) type system.

Our approach eliminates the adoption threshold for systems implemented in Haskell: such systems can be made more secure without completely rewriting them in a new language. The implementation can be a fine-grained mixture of normal code and security-hardened code (variables, data and computations over secure data). The programmer needs to protect only sensitive data and computation using a software library, which enforces the information-flow policies throughout the entire system and provides end-to-end security goals like noninterference.

Another benefit of our approach is flexibility. A specialized language like Jif must pick a fixed policy framework in which the security policies are expressed. Considering the plethora of possible features present in the literature with regards to how to express the label lattice, declassification options [12], dynamic policy information [14, 17], etc., it is unlikely that any particular choice of policy language will be suitable for all programs with security concerns. By contrast, since it is much easier to build a library module than to build a new language, it is conceivable that different programs would choose to implement entirely different policy frameworks. Our embedded sublanguage approach is *modular* in the sense that it provides an interface through which the programmer can choose which policy framework and type system to use for specific security goals. In this paper we sketch one possible policy framework that illustrates one particular choice of label lattice, declassification mechanism, and support for dynamic policies, but others could readily be implemented instead.

Although we use Haskell's advanced type system and helpful features like the ability to overload syntax, studying how to encode information-flow policies in the context of Haskell can point to how similar efforts might be undertaken in more mainstream languages like Java. Also, since the features we use are intended to be "general purpose," they are more likely to find a home in a mainstream language than the less widely applicable security types. Evidence of this can be found, for example, in Sun's recent addition of parametric polymorphism, a key component of our approach, to Java.

1.2. Overview of technical development

There are two key technical challenges in embedding a useful security-typed sublanguage in Haskell. The first problem is that enforcing information-flow policies requires static analysis of the control flow graph of embedded programs—purely dynamic enforcement mechanisms are generally too conservative in practice. The second problem is representing the policy information itself—depending on the desired model, the policy information might be quite complex, perhaps depending on information available only at run time.

Our solution to the first problem is to use *arrows* [5]. Intuitively, arrows provide an abstract interface for defining embedded sublanguages that support standard programming constructs familiar to programmers: sequential composition, conditional branches, and loops. Haskell provides convenient syntactic sugar for writing programs whose semantics are given by an arrow implementation.

To address the second problem, we use Haskell's *typeclass* mechanism to give an interface for security lattices. Programs written in the embedded language can be parameterized with respect to this interface. Moreover, the embedded language can easily be given security-specific features such as a declassification operation or run-time representation of privileges for access-control checks.

In both cases, we make use of Haskell's strong type system to guarantee that the abstractions enforcing the security policies are not violated. This encapsulation means that it is not possible to use the full power of the Haskell language to circumvent the information-flow checks performed by the embedded language, for example.

Before diving into the details of how the embedded language is implemented, it is useful to see how we imagine these techniques being used in practice. The next section gives some example programs that illustrates how a secure embedded language and the Haskell program can be smoothly integrated, explains some background about Haskell syntax, and considers the issues with enforcing the desired security properties. Section 3 describes the arrow interface and explains how it corresponds to the example. Section 4 presents the detailed implementation of an arrow-based sublanguage for information-flow control. Section 5 discusses some limitations and caveats with this approach and describes some future work. Section 6 concludes.

2. Example use of the embedded language

This section presents the features of our secure embedded sublanguage using code examples. We start by encoding a security lattice and use simple program fragments to show how information-flow policies can be specified in programs. Then, we use a larger application to introduce declassification and policy enforcement mechanisms.

2.1. Encoding Security Lattices

The security sublanguage provides a generic interface for defining security labels as first-class values. A security lattice can be implemented by the programmer using the `Lattice` typeclass:

```
class (Eq a) => Lattice a where
    label_top    :: a
    label_bottom :: a
    label_join   :: a -> a -> a
    label_meet   :: a -> a -> a
    label_leq    :: a -> a -> Bool
```

The above definitions says values of type **a** can be used as security labels if **a** supports the usual set of label operations. Since we use standard Haskell values to represent security labels, there is no limitation on the expressiveness of security policies: the programmer has the freedom to choose any implementation for security labels. For example, the following code implements a simple security lattice of three points:

```
data TriLabel = LOW | MEDIUM | HIGH
instance Lattice TriLabel where
  label_top = HIGH
  label_bottom = LOW
  label_join x y=if x `label_leq` y then y else x
  label_meet x y=if x `label_leq` y then x else y
  label_leq LOW _ = True
  label_leq MEDIUM LOW = False
  label_leq MEDIUM _ = True
  label_leq HIGH HIGH = True
  label_leq HIGH _ = False
```

We could have implemented a policy language as complex as the full *decentralized label model* [7], but for simplicity and ease of presentation, we will use this lattice definition throughout the paper.

2.2. Programming with information-flow policies

The security sublanguage defines a polymorphic abstract data type **Protected a** to represent a computation of result type **a**. Internally, such protected computation is associated with its information-flow policies, which represent the security levels of inputs and outputs.

The sublanguage provides a set of primitive operations for constructing computations of protected types. The constructor of the **Protected** type is held abstract to the user program, so protected computations cannot be freely opened: they can only be accessed using sublanguage primitives. The simplest operation is **pure**, which converts a standard Haskell computation to a protected computation. The sequencing operation **>>>** composes two protected computations together by connecting the output of the first computation to the input of another.

By default, protected computations constructed by **pure** have no information-flow constraints. Interesting policies can be specified by using the **tag** operation. The following function **tag_val** takes an arbitrary computation **x** and a security label **l** as inputs, converts **x** to a protected closure using **pure**, and composes it with an information-flow annotation using **>>>** and **tag**. The output is a protected computation with an output label **l**.

```
tag_val :: a -> TriLabel -> Protected a
tag_val x l = pure (\_ -> x) >>> tag l
cH = tag_val 3 HIGH
cM = tag_val 4 MEDIUM
cL = tag_val 5 LOW
```

Using **tag_val**, we can define **cH, cM, cL** as protected values with different information-flow policies. The following shows some computation using these protected values:

```
t1 = liftA2 (+) cL cM
t2 = liftA2 (*) cH cM
t3 = proc () -> do
{       h <- cH -< ();
        if h>3 then do { x <- cM -< ();
                         returnA -< x;
                       }
               else do { x <- t1 -< ();
                         returnA -< x;
                       }
}
```

The **liftA2** function is a generic arrow operation that can be used in our sublanguage to convert any standard binary operator to an operator on protected types: **t1** is the sum of **cL** and **cM**, **t2** is the product of **cH** and **cM**. The sublanguage syntax in the definition of **t3** is more complex, but it suffices to say it represents the computation **if cH>3 then cM else t1**.

The security sublanguage rigorously captures both explicit and implicit information flows in protected computations. When the above code is executed, **t1** will have label **MEDIUM**, while **t2** and **t3** will have label **HIGH**. The control flow of the protected computation is represented using operations provided by the sublanguage, and these operations keep track of the information flow policies and constraints incrementally during the construction of protected computations.

Any protected computation can be used together with the **tag** operation to restrict the information flow. The following function takes a protected computation **c** as argument and requires the output of **c** to be no higher than **MEDIUM**:

```
expects_medium :: Protected a -> Protected a
expects_medium c = c >>> tag MEDIUM
```

Now, we can use this function with protected computations:

```
success1 = expects_medium t1
failure2 = expects_medium t2
failure3 = expects_medium t3
```

The first computation **success1** is fine, because **t1** has label **MEDIUM**. The second and the third both violate the information-flow policies, because **t2** and **t3** both have label **HIGH** while **MEDIUM** is expected: there is information flow from **HIGH** to **MEDIUM**.

So, what will happen if policies are violated? Crucially, Haskell uses the lazy evaluation strategy by default. None of the above computations are actually performed until their results are needed. The security sublanguage is designed so that protected computations cannot be evaluated unless their information-flow policies have been checked by a run-time mechanism. Therefore, if the information-flow policies are violated, none of the protected computation will be performed, but a run-time error will be generated when the program tries to use the protected computations. Although this is a dynamic information-flow tracking system, it performs static analysis on the control-flow graphs of protected computations and provides strong security guarantees.

2.3. Declassification and code privileges

Our security sublanguage supports declassification mechanisms similar to the one in Jif. Protected computation with high security levels can be declassified to low levels using the **declassify** operation. However, the code needs sufficient privilege to perform this operation. The sublanguage provides an abstract type **Priv** to represent the security level of the user at run-time. **Priv** can be passed to functions as a "capability" argument. The code privilege is needed to verify information-flow policies and access protected computations.

The sublanguage is designed so that any common values of type **a** can be converted to protected values of type **Protected a**. However, protected computations have abstract types and cannot be used outside the sublanguage. Whenever we want to access such computation outside the sublanguage, we are at an "end-point" of the secure computation, and we need to certify that the computation satisfies its information-flow policies. The **cert** operation takes a protected computation and verifies its security policies: (1) the output type must be **LOW**, and (2) the current code privilege must be sufficient to perform the declassification operations in the protected computation. The protected computation is suspended by lazy evaluation, and they will not start executing unless the **cert** operation succeeds.

2.4. An interactive multi-user application

We use an interactive application to demonstrate the use of these language features. It simulates an online network service, where users can log in and access information. There are only two kinds of users: guests and administrators. Guests have security level **LOW** while administrators have security level **HIGH**. Guests can enter numbers as price bids, while the administrator can log in to see the highest bid. The information-flow policy is that guests are not allowed to know what the highest bid is. To implement this, we maintain the highest bid as a global state **stat** with security level **HIGH**. The following code shows a simple session for guest services.

```
guest_service::Priv->(Protected Int)->IO(Protected Int)
guest_service priv stat = do
{  putStrLn "Enter a number:";
   i <- getNumber;
   let stat' = proc () -> do
     {  x <- stat -< ();
        if i>x then returnA -< i
               else returnA -< x;
     }
   return stat';
}
```

The **guest_service** function takes two arguments: **priv** has type **Priv** and it represents a code privilege passed to this function; **stat** has type **Protected Int** and it is the secret global state. The function returns a new state, which also has type **Protected Int**. The main body of the guest service is written in a standard IO monad; when it runs, it reads a number from input and updates the state if the input **i** is larger than the protected state **stat**.

The body of the **let** expression, however, is the computation written in our embedded sublanguage. The sublanguage implements a standardized combinator interface called *arrows* (explained in the next Section), but the implementation details of these combinators are hidden from the programmer. Instead, the programmer uses Haskell's "**do**-syntax" for arrow computations [8]. The **do**-syntax provides higher-level language control constructs such as variable binding and conditional branches, and the Haskell compiler translates code in the **do**-syntax to the standard arrow operations, which are overloaded by our sublanguage using Haskell typeclasses.

In the inner **do** block, there are two commands. The first command "**x <- stat -< ();**" binds the value of the secret computation **stat** to a local variable **x**, where **x** has type **Int**. Now, **x** can be freely used in any computation, but it cannot escape the scope of the **do**-block. The next command "**if..then..else..**" performs a conditional branching in the sublanguage. The body of the branch "**returnA -< i**" generates the output for the **do**-block. Overall, the computation represented by the "**proc ...do**" block is bound to the variable **stat'** and it is returned as the result.

The **admin_service** implements a similar session for administrator services. It has the same type as **guest_service**.

```
admin_service: Priv->(Protected Int)->IO(Protected Int)
admin_service priv stat = do
{   let low = stat >>> (declassify HIGH LOW) in
    let summary = cert priv low () in
    putStrLn (show summary);

    let stat_new = (pure (\_->0) >>> tag HIGH) in
    return stat_new;
}
```

In contrast to the previous function, **admin_service** uses the combinators provided by the sublanguage directly. In the first **let** expression, it declassifies the protected state to **LOW** level using the **declassify** operation and binds the result to the variable **low**. The variable **low** also has type **Protected Int**, but the security level associated with it is **LOW** after the declassification. Then, it uses the **cert** operation, together with its code privilege **priv**, to access the computation protected in **low**. The result **summary** has an unrestricted **Int** type and thus can be used as common Haskell values. The next line calls the standard Haskell printing function **putStrLn** to send this value to the program output. Finally, it creates a new protected value **stat_new** initialized to 0, sets the security policy of this protected value to be **HIGH**, and returns it as the new global state.

This function uses several combinator operations provided by the security sublanguage. In the first **let** expression, it uses the **declassify** operation to create an empty computation with the information-flow policy **HIGH** → **LOW**. Then, it uses the **>>>** operation to sequentially compose two protected computations **stat** and **declassify** together. Therefore, the result **low** will have the information-flow policy **LOW** for its output. In the last **let** expression, it uses the **pure** operation to convert a common Haskell function to a protected computation. The **tag** operation creates an empty computation with a fixed policy **HIGH** → **HIGH**. When **pure** and **tag** are composed together using **>>>**, it creates a new protect computation with security level **HIGH**.

```
service_loop::(Protected AuthDB)->(Protected Int)->IO()
service_loop auth_db stat = do
{
  putStrLn "Enter username and password:";
  u <- getLine; p <- getLine;
  let (ident,priv) = authenticate auth_db u p
  ;
  stat_new <- case ident of
    "admin" -> admin_service priv stat;
    "guest" -> guest_service priv stat;
     _ -> do {
               putStrLn "login error";
               return stat;
             }
  ;
  service_loop auth_db stat_new;
}
```

The **service_loop** function is part of the trusted computing base. It authenticate users and dispatch services. On every loop, it reads a user name and a password from the input, and authenticates the user. The **authenticate** function constructs an abstract code privilege depending on the identity of the user. Once a code privilege is available, it is used to execute the corresponding service function.

```
main = do
{
  let auth_db :: (Protected AuthDB) =
       (
         pure (\_-> [("admin","admin",HIGH),
                     ("guest","guest",LOW)] ) >>>
         tag HIGH
       )
      secret_val :: (Protected Int) =
       (
          pure (\_->0) >>>
          tag HIGH
       )
  in
  service_loop auth_db secret_val;
}
```

The top-level **main** function is also part of the trusted computing base. It creates an authentication database and an initial global state, which is tagged by the security level **HIGH**. The combinators **pure, tag, >>>** are similar to those in **admin_service**.

Security guarantees:
How does this code enforce our information-flow policies? Suppose an untrusted programmer adds the following code in **guest_service** to declassify the secret state:

```
let s = cert priv
    (stat >>> declassify HIGH LOW) ()
```

Or, suppose the services are incorrectly dispatched:

```
stat_new <- case ident of
 "admin" -> guest_service priv stat
 "guest" -> admin_service priv stat
```

In such cases, when the program tries to declassify the data and certify the result using the guest privilege, there will be a run-time error. The global state is tagged with the label **HIGH**, but the guests can only acquire **LOW** privileges during the authentication process. The declassification operation requires that the user privilege must be higher than the security level of the data to be declassified. Therefore, the guest cannot declassify the global state to **LOW** and use **cert** to steal the secret state. This provides a security mechanism similar to that of using *run-time principals* [14].

Aside from the authentication process and initial setup of confidential state, the information-flow policies are automatically enforced throughout the system. The **guest_service** and the **admin_service** are very simple in this example, but they can be scaled to more complex services, system states and security policies.

Benefits:
As we have seen, the application program is a fine-grained mixture of normal components written in standard Haskell and secure components constructed using a few special operations from the sublanguage. To those unfamiliar with Haskell, it may be hard to distinguish where the "embedded" language ends and the "base" language begins, but that is part of the point—the programmer has easy access to both the strong security guarantees of the embedded language *and* the full power of Haskell at the same time: all the Haskell language features and software libraries are still available.

3. The arrows interface

As described in the introduction, *arrows* can be thought of as an interface, called a type class in Haskell parlance, for defining embedded sublanguages. Haskell actually provides several related type classes that each provide refinements to the basic arrow functionality.[1]

The following code specifies the simplest **Arrow** type class. Here, **a** is an abstract type of arrows with input type **b** and output type **c**. This arrow type class supports only three operations: **pure**, **(>>>)**, and **first**.

```
class Arrow a where
  pure  :: (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b, d) (c, d)
```

1 The related references [5, 9, 8] in the bibliography can be used for more detailed studies of arrows.

Basic blocks and compositions

Intuitively, the **pure** operation lifts a Haskell function of type **b -> c** into the arrow; such lifted functions serve as the "basic blocks" of the control-flow graphs constructed via arrow combinators. The infix operation **>>>** provides sequential (horizontal) composition of computations, and **first** provides parallel (vertical) composition of computations.

An instance of the **Arrow** type class is required to satisfy a set of axioms that specify coherence properties between the operations. For example, **>>>** is required to be associative. We omit the complete description of the arrow axioms here; for our purposes, there is only one interesting case to consider, and it is discussed in Section 5.

The simplest instance of **Arrow** is Haskell's function arrow constructor **(->)** itself: every function of type **b -> c** is also an arrow computation **(->) b c**.

Representing conditionals and loops

The basic **Arrow** interface does not provide the ability to construct conditional computations—it can construct only control-flow graphs that represent straight-line code with no branches. Two other type classes refine arrows by permitting conditional branches and loops.

The **ArrowChoice** type class provides an operation called **left** that extends the **Arrow** interface with the ability to perform a one-sided branch computation depending on the arrow's input value. In Haskell, the type **Either b d** describes a value that is a tagged union or option type.

```
class Arrow a => ArrowChoice a where
 left :: a b c -> a (Either b d) (Either c d)
```

Using **left** and the other arrow primitives, the following operations can be implemented to construct different kinds of conditional computations:

```
right :: a b c -> a (Either d b) (Either d c)
(+++) :: a b c -> a b' c'->a (Either b b') (Either c c')
(|||) :: a b d -> a c d -> a (Either b c) d
```

ArrowLoop provides the embedded language with a loop construct sufficient for encoding **while** and **for** loops. Intuitively, the **loop** operator feeds the **d** output of the arrow back into the **d** input of the arrow, introducing a cycle in the control-flow graph:

```
class Arrow a => ArrowLoop a where
 loop :: a (b, d) (c, d) -> a b c
```

The benefit of having this operation is that recursive computations can be constructed as a finite combination of arrow components. We will use this property in Sec. 4.5.

Translating the do-syntax

Programming directly with the arrow operations is sometimes cumbersome, because arrows require a point-free programming style. The **do**-syntax for arrows [8] provides syntactic sugar for arrow programming, such as arrow abstraction, arrow application, sequential composition, conditional branching and recursion. Internally, the Haskell compiler[2] translates the **do**-syntax used in the embedded sublanguage into the basic arrow operations. For example, conditional statements are translated to **pure, >>>** and **|||** operators, using the following rule:

```
⟦ proc p -> if e then c1 else c2 ⟧ =
  pure (\p -> if e then Left p else Right p)
     >>> ⟦ proc p -> c1 ⟧ ||| ⟦ proc p -> c2 ⟧
```

This rule translates the **if** command in the sublanguage. The **if** construct in the translated code is the conditional expression in the base Haskell language. The sublanguage syntax "**proc p->**" provides an arrow abstraction that binds the arrow input to the variable **p**.

The compiler is able to resolve this syntax overloading by using type information: the type **Protected** mentioned in the example code informs the compiler to use the definition of **|||** given by our **FlowArrow** instance of the **ArrowChoice** type class (described below).

3.1. Control flow in arrow sublanguages

The Haskell programming language itself provides branching, looping, and other control-flow constructs, so one might wonder why it is necessary to re-implement all of these features in the embedded sublanguage. Compared to Haskell's full control-flow mechanisms (which also include function calls and exceptions, for example), the arrow type classes are actually quite impoverished. The arrow interfaces isolate the base language (Haskell) and the sublanguage (arrows): by design, the control flow constructs in the base language cannot be directly used to represent the control flow of the sublanguage.

This separation property is crucial for the security analysis of arrow-based sublanguages. If an arrow implements only the operations in the **ArrowChoice** type class, conditional branches on arrow computations can only be implemented using the given arrow operations **left**, **right**, **(+++)**, and **(|||)**. By keeping the arrow implementation abstract, the programmer is forced to use these arrow operations for writing conditional branches, because there is no other way to manipulate the interface.

Therefore, by designing the arrow interface with limited control-flow primitives, the control-flow graph of an arrow computation is determined by the composition of primitive arrow operations. In other words, arrows can force computations to be constructed with static and explicit control-flow structures. This makes it possible to completely analyze the information flow in an arrow-based sublanguage

2 We use the Glasgow Haskell Compiler [1].

before running the computation. A more permissive interface to the sublanguage (such as provided by *monads* for example) would allow base language branches to leak information about supposedly protected data.

4. An embedded language for information-flow control

This section presents the design of our secure embedded sublanguage using the arrows interface. The design uses the structure of *arrow transformers*, which allows arrow sublanguages to be composed in a modular, layered fashion.

4.1. Encoding flow types and constraints

The design in this paper assumes arrow computations are purely functional and uses a simple information-flow type system for arrow computations. A computation has an input security label l_1 and an output security label l_2. The typing judgment has the form

$$\Phi \vdash c : l_1 \rightarrow l_2$$

where c is a purely functional computation, $l_1 \rightarrow l_2$ is the *flow type* assigned to c, and Φ is a list of label constraints. The type system is presented in Figure 1.

The sublanguage flow type is encoded using the Haskell **`Flow`** data type:

```
data Flow l = Trans l l | Flat
```

1. **`Trans`** l_1 l_2 specifies a security type $l_1 \rightarrow l_2$.
2. **`Flat`** means the input and output can be given the same arbitrary label. It specifies a security type $l \rightarrow l$, where the label l can be determined by constraints in the context.

The label constraints are encoded using the **`Constraint`** data type:

```
data Constraint l = LEQ l l | USERGEQ l
```

1. **`LEQ`** l_1 l_2 represents a direct ordering between two labels: $l_1 \sqsubseteq l_2$.
2. **`USERGEQ`** l represents the constraint $l \sqsubseteq$ **`user`**. It requires that the run-time code privilege, which is represented as a label, be at least l. This will be used in Sec. 4.4 when we implement declassification.

The purpose of using constraints Φ is to implement late binding of the security lattice. The type system collects the label constraints when secure computations are constructed from individual components. Such constraints are checked when the secure computation is accessed through the policy enforcement mechanism, namely, the **`cert`** operation. This design makes it possible to use dynamic security lattices and also helps when implementing declassification.

4.2. Encoding typing judgments and rules

The abstract datatype **`FlowArrow`** defines our secure embedded language by implementing the arrow interfaces described above:

```
data FlowArrow l a b c = FA
    { computation :: a b c
    , flow        :: Flow l
    , constraints :: [Constraint l] }
```

A value of type **`FlowArrow l a b c`** is a record with three fields. The **`computation`** field encapsulates an arrow of type **`a b c`** that is the underlying computation. The **`flow`** field specifies the security levels for the input and output of the computation. The **`constraints`** field stores the list of flow constraints Φ when the arrow computation is constructed from smaller components.

`FlowArrow` encodes an information-flow typing judgment for an effect-free arrow computation, using the encoding of flow types and constraints we just defined. The typing judgment $\Phi \vdash c : l_1 \rightarrow l_2$ is represented by the value:

$$\texttt{FA}\ c\ \texttt{(Trans}\ l_1\ l_2\texttt{)}\ \Phi$$

`FlowArrow` uses a generic design of arrow transformers and it is parameterized by many types:

1. The type l of security labels. **`(FlowArrow`** l**`)`** is an arrow transformer.
2. The effect-free arrow **`a`** we are transforming from. The simplest and most common case of **`a`** is the function arrow **`(->)`**. The result **`(FlowArrow l a)`** is also an arrow.
3. The input type **`b`** and output type **`c`**.

The arrow **`a`** must have no side-effects. Although **`FlowArrow`** is a generic arrow transformer, we do require that the arrow **`a`** represent a purely functional computation where information flows from one end to the other, so the information-flow types in the form of $l_1 \rightarrow l_2$ will make sense. For the rest of the paper, the reader can assume **`a`** is the function arrow **`(->)`** for ease of understanding. The type **`(Protected a)`** in Sec. 2 is actually an abbreviation for a **`FlowArrow`** type:

```
type Protected a = FlowArrow TriLabel (->) () a
```

`(FlowArrow l a)` is an arrow, so we can use **`FlowArrow`** as a sublanguage to represent computation. At the same time, **`(FlowArrow l a)`** also encodes a typing judgment, so we can verify the information-flow policies for the computation. Essentially, the implementation of **`(FlowArrow l a)`** is a type-checker: each arrow operation implements a typing rule for that operation. For standard arrow operations on **`a`**, **`FlowArrow`** lifts them by (1) running the original operation on the **`computation`** fields of arguments, and (2) computing the flow types and constraints using such information from arguments.

$\boxed{\Phi \vdash c : l_1 \to l_2}$

$$\frac{}{\emptyset \vdash \textbf{pure}\ f : l \to l} \quad \text{PURE}$$

$$\frac{\Phi_1 \vdash c_1 : l_1 \to l_2 \qquad \Phi_2 \vdash c_2 : l_3 \to l_4}{\Phi_1 \cup \Phi_2 \cup \{l_2 \sqsubseteq l_3\} \vdash c_1 \ \textbf{>>>}\ c_2 : l_1 \to l_4} \quad \text{SEQ}$$

$$\frac{\Phi \vdash c : l_1 \to l_2}{\Phi \vdash \textbf{op}\ c : l_1 \to l_2} \quad \text{ONE}$$

$$\frac{\Phi_1 \vdash c_1 : l_1 \to l_2 \qquad \Phi_2 \vdash c_2 : l_3 \to l_4}{\Phi_1 \cup \Phi_2 \vdash c_1 \ \textbf{op}\ c_2 : l_1 \sqcap l_3 \to l_2 \sqcup l_4} \quad \text{PAR}$$

$$\frac{\Phi \vdash c : l_1 \to l_2}{\Phi \cup \{l_2 \sqsubseteq l_1\} \vdash \textbf{loop}\ c : l_1 \to l_2} \quad \text{LOOP}$$

$$\frac{}{\emptyset \vdash \textbf{tag}\ l : l \to l} \quad \text{TAG}$$

$$\frac{}{\{l_1 \sqsubseteq \textbf{user}\} \vdash \textbf{declassify}\ l_1\ l_2 : l_1 \to l_2} \quad \text{DECL}$$

$\boxed{\mathbb{L}, l_{user} \succ c : l_{in} \to l_{out}}$

$$\frac{\Phi \vdash c : l_1 \to l_2 \qquad l_{in} \sqsubseteq l_1 \qquad l_2 \sqsubseteq l_{out} \qquad \mathbb{L} \vdash \Phi[l_{user}/\textbf{user}]}{\mathbb{L}, l_{user} \succ c : l_{in} \to l_{out}} \quad \text{CERT}$$

Figure 1: Information-flow type system implemented by the secure sublanguage

```
instance (Lattice l, Arrow a) =>
                    Arrow (FlowArrow l a) where
  pure f = FA { computation = pure f   -- PURE --
              , flow = Flat
              , constraints = [] }
  (FA c1 f1 t1) >>> (FA c2 f2 t2) =    -- SEQ --
      let (f,c) = flow_seq f1 f2 in
           FA { computation = c1 >>> c2
              , flow = f
              , constraints = t1 ++ t2 ++ c }
  first (FA c f t) =                   -- ONE --
      FA { computation = first c
         , flow = f
         , constraints = t }
  (FA c1 f1 t1) &&& (FA c2 f2 t2) =    -- PAR --
      FA { computation = c1 &&& c2
         , flow = flow_par f1 f2
         , constraints = t1++t2 }
  (FA c1 f1 t1) *** (FA c2 f2 t2) =    -- PAR --
      FA { computation = c1 *** c2
         , flow = flow_par f1 f2
         , constraints = t1++t2 }

instance (Lattice l, ArrowChoice a) =>
             ArrowChoice (FlowArrow l a) where
  left (FA c f t) =                    -- ONE --
     FA { computation = left c
        , flow = f
        , constraints = t }
  (FA c1 f1 t1) +++ (FA c2 f2 t2) =    -- PAR --
     FA { computation = c1 +++ c2
        , flow = flow_par f1 f2
        , constraints = t1++t2 }
  (FA c1 f1 t1) ||| (FA c2 f2 t2) =    -- PAR --
     FA { computation = c1 ||| c2
        , flow = flow_par f1 f2
        , constraints = t1++t2 }

instance (Lattice l, ArrowLoop a) =>
             ArrowLoop (FlowArrow l a) where
 loop (FA c f t) =                     -- LOOP --
   let t' = constraint_loop f in
     FA { computation = loop c
        , flow = f
        , constraints = t ++ t'
        }
   where
   constraint_loop Flat = []
   constraint_loop (Trans l1 l2) = [LEQ l2 l1]

flow_seq::Flow l->Flow l->(Flow l, [Constraint l])
flow_seq (Trans l1 l2) (Trans l3 l4)=
         (Trans l1 l4, [LEQ l2 l3])
flow_seq Flat f2 = (f2,[])
flow_seq f1 Flat = (f1,[])
flow_par :: (Lattice l)=>Flow l->Flow l->Flow l
flow_par (Trans l1 l2) (Trans l3 l4) =
    Trans (label_meet l1 l3) (label_join l2 l4)
flow_par Flat f2 = f2
flow_par f1 Flat = f1
```

Figure 2: Implementation of arrow operations

The implementation of **FlowArrow** is given in Figure 2. In the definition of each arrow operation, the operation in **FlowArrow** (on the left hand side) is implemented using operations in the arrow **a** (on the right hand side).

The **pure** operation returns a **Flat** flow type and no constraints, because such computations have no information-flow policies on them. It implements the PURE typing rule. **Flat** represents a flow type $l \to l$, but the label l never appears in the implementation—it can always be inferred from context.

The **(>>>)** operation sequentially composes two arrow computations. The flow types and constraints are computed using the **flow_seq** function, which implements the SEQ typing rule.

The **first** and **left** operations implement the ONE typing rule. The **(&&&)**, **(***)**, **(|||)** and **(+++)** operations are parallel compositions and they implement the PAR rule. The **flow_par** function is used to compute parallel composition of flow types.

The **loop** operation is slightly more interesting. It implements the LOOP typing rule. Since **loop** connects the output of a computation back to its input, we generate a constraint to capture this information flow. This requires that the input and the output have the same security level, unless there is declassification inside the computation.

As described in Section 3, the **do**-syntax of the secure sublanguage is translated to these standard arrow operations. Therefore, the typing judgment for code written in the **do**-syntax can be derived by combining the typing rules implemented in these standard arrow operations. For the trans-

lation of conditional commands shown in the end of Sec. 3, the combination of typing rules yields essentially the same constraints as the following COND rule found in conventional information-flow type systems:

$$\frac{\Gamma \vdash e_1 : l_1 \qquad \Gamma \vdash e_2 : l_2 \\ \Gamma \vdash e_3 : l_3 \qquad l_1 \sqsubseteq l_2 \sqcup l_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : l_2 \sqcup l_3}\ \text{COND}$$

It is by this property that the implicit information flow in the **t3** example in Sec. 2.2 can be captured by our type system.

4.3. Policy specification

So far, the typing rules implemented in **FlowArrow** only construct computations from smaller components while composing their information-flow policies. Pure computations are given the $l \to l$ flow type, but we need a way to introduce more interesting flow types. Recall from Sec. 2 that the **tag** operation annotates a computation with a security label. It implements the TAG rule in Figure 1.

```
tag::(Lattice l,Arrow a)=> l -> FlowArrow l a b b
tag l = FA { computation = pure (\x->x)
           , flow = Trans l l
           , constraints = [] }
```

When **tag** is applied to a label l, it creates an arrow that represents an empty step of computation, with the flow type $l \to l$. Intuitively, **tag** inserts a "pipe" in the middle of the computation, with explicit flow types specified on both ends. For example, to annotate the confidential value **secret** with label **HIGH**, we can use the following code which has a flow type **HIGH** $\to$ **HIGH**:

```
pure (\_->secret) >>> tag HIGH
```

To assert that the output of a computation **c** has no confidential information, we can simply use the following code to connect **c** to a "low" pipe:

```
c >>> tag LOW
```

For confidentiality policies, we care about the future of secret computation, i.e. where information will flow to. Therefore, a good pattern for protecting confidentiality is to append **tag** to the output of the computation we want to protect. The design of **tag** and the arrow types are completely symmetrical. If we have labels for integrity policies, we can connect the output of **tag** to computations that require trustworthy data as inputs. This is a bi-directional design that works for both confidentiality and integrity policies.

4.4. Declassification

Declassification is a practical requirement for language-based information-flow control. We need a mechanism to allow information flow from high levels to low levels, but only in controlled ways. The *decentralized label model* (DLM) [7] solves this problem by assigning code with *authority*. Each declassification statement can only weaken the security policy that belongs to the authority of code. In our arrows framework, there is no difficulty of encoding the labels in DLM, but we need a declassification mechanism that takes code authority into account.

For simplicity, we designed the declassification construct and its corresponding flow constraints for simple lattices such as the **TriLabel** lattice implemented in Sec. 2. It is simpler than the declassification mechanism in Jif, but it implements the essence of code authority checking.

```
declassify :: (Lattice l, Arrow a) =>
              l -> l -> FlowArrow l a b b
declassify l1 l2 =
    FA { computation = pure (\x->x)
       , flow = Trans l1 l2
       , constraints = [USERGEQ l1] }
```

The **declassify** operation implements the DECL rule in Figure 1. It is similar to the **tag** operation except that it constructs a "pipe" where the security level of the output is lower than the level of input. Similar to other operations, **declassify** does not check the policies directly, but it creates a constraint which can be checked later. When applied to two label values, **declassify** $l_1\ l_2$ creates an arrow with flow type $l_1 \to l_2$ and a flow constraint **USERGEQ** l_1 stating that the code privilege must be at least l_1. For example, if the code privilege is **HIGH**, it can declassify information from **MEDIUM** to **LOW**. But if the code privilege is **MEDIUM**, it cannot declassify from **HIGH** to **LOW**.

4.5. Policy enforcement

Finally, we need to check the flow types and constraints that we have accumulated during the construction of a secure computation. Since we have a declassification mechanism which takes code privilege into account, the code privilege must be provided to check the constraints.

```
data Privilege l = PR l

certify :: (Lattice l) => l -> l ->
           Priv l -> FlowArrow l a b c -> a b c
certify l_in l_out (PR l_user) (FA c f t) =
 if not $ check_levels l_in l_out f then
     error $ "security level mismatch" ++ (show f)
 else if not $ check_constraints l_user t then
     error $ "constraints cannot be met"++(show t)
 else c
```

The judgment $\mathbb{L}, l_{user} \succ c : l_{in} \to l_{out}$ states the security property to be checked: given a security lattice $\mathbb{L}$ and the label l_{user} representing the code privilege, does the arrow computation c satisfy the information-flow policy $l_{in} \to l_{out}$? The CERT rule in Figure 1 checks this security property and the **certify** function implements this rule.

The **certify** operation takes a few arguments:

1. The information-flow types l_{in} and l_{out} that we expect the computation to have. Suppose the flow type of the

secure computation is $l_1 \rightarrow l_2$, **certify** calls another function to verify that $l_{in} \sqsubseteq l_1$ and $l_2 \sqsubseteq l_{out}$.

2. The code privilege l_{user}, under which the computation is performed. The **certify** operation checks all the constraints that come with the secure computation. For any constraints of the form **USERGEQ** l, a check is performed to make sure that $l \sqsubseteq l_{user}$. If any constraint is not satisfied, a run-time error is generated.
3. A **FlowArrow** value that includes the secure computation to be checked. If all the above checks are successful, the embedded secure computation is returned. Note that we stacked the arrow transformer **FlowArrow** on another arrow **a**, this **certify** operation strips **FlowArrow** off and gives back computations in arrow **a**.

Although **certify** is a dynamic enforcement mechanism (it executes as part of the Haskell program), it provides strong security guarantees. When an embedded computation is certified, its whole control structure is examined using an information-flow analysis before any part of embedded computation is performed. This process is like type checking the embedded sublanguage.

Branches over arrow computations can only be constructed using the operators provided in **FlowArrow**.[3] The control structure of a secure computation is independent of the values generated in the computation. Therefore, if the run-time check fails, the failure does not leak information about secrets inside the computation.

A minor caveat is that recursive arrow computations should be constructed using the **loop** operation rather than using standard Haskell recursion. The **certify** function checks the flow types and constraints of the whole computation, so it forces evaluation of all arrow operations used to construct the computation. If the computation is recursively constructed using standard Haskell recursion, **certify** will essentially try to check an infinite control flow graph with an infinite typing derivation, which will exhaust Haskell's stack space and eventually abort the program. In such cases secret information is not leaked, but the secure computation should be re-written using the **loop** operation.

The **certify** interface seems verbose, but it is very flexible to use. For protecting confidentiality, we only care about the security level of the output, so the argument l_{in} can always be **label_bottom**. We also require all secrets be declassified to the lowest security level before reaching the output channel, so we let the argument l_{out} always be **label_bottom**. Thus, we hide the definition of **certify** and define a simpler operation **cert**:

```
cert = certify label_bottom label_bottom
```

3 Importantly, **FlowArrow** does not implement a richer interface such as the **ArrowApply** type class that would make it impossible to analyze the control structure.

4.6. Code privileges

When using **certify**, it is important that the code privilege is correctly specified: untrusted code cannot call **certify** using a code privilege that it does not have. Our solution is to define an abstract data type **Privilege** that internally stores a label as code privilege. The **certify** operation takes values of the abstract type **Privilege** as its input.

```
data Privilege l = PR l
```

The key point is to make the constructor **PR** only available in trusted modules. The program must be organized such that untrusted code can only treat the **Privilege** type abstractly. Privileges can only be created in trusted code and passed to untrusted code as shown in Sec. 2, where the type **Priv** is an abbreviation:

```
type Priv = Privilege TriLabel
```

Developing appropriate design patterns for structuring privileged code is an important task that we leave to future work. An interesting question as with all capability-based authorization mechanisms is how to revoke the privileges passed to untrusted code. If the untrusted code has state and runs under several privileges in different places, it can steal privileges by storing and reusing them. One solution is to encode version numbers in such privileges and have a global state to indicate valid privileges, doing so would require the top level code be inside a monad.

5. Discussion and future work

5.1. Compile time vs. run time

For applications written using the embedded language, there are two stages of type-checking. At compile time, the base language (in this case, Haskell) is type checked and compiled. At run time, the embedded language is type checked before embedded secure computations are executed. Therefore, the information-flow policy violations are not detected until the application is launched. Although the sublanguage uses static analysis techniques and provides similar strong security guarantees, this two-stage mechanism is sometimes not as convenient as specialized languages such as Jif. Each run of the application may only use part of the secure computations, so debugging can be more difficult. Therefore, it is appealing to have a one-stage, compile-time enforcement mechanism for the sublanguage. Such a mechanism can be possible if it is build entirely in the static type system of the host language.

Abadi et. al. developed the *dependency core calculus* (DCC) [2], which uses a hierarchy of monads to model information flow. Tse and Zdancewic [15] translated DCC to System F and showed that noninterference can be translated to a more generic property called *parametricity*, which

states that polymorphic programs behave uniformly for all their instantiations. An intuitive demonstration of this idea is that abstract data types can be used as a protection mechanism to hide high-security information. They presented a Haskell implementation where each security level is encoded using an abstract data type and binding operators are defined to compose computations with permitted information flows. This approach works well for simple lattices, but encoding the security lattice of n points would require $O(n^2)$ definitions for binding operators. This makes it difficult to implement more complex security lattices such as the *decentralized label model*.

The problem with this approach is policy *expressiveness*. The type system of the base language must be expressive enough to encode the syntax and the semantics of security policies. Although Haskell has an expressive type system, it is not clear how to encode more expressive policies directly in the type language — we leave that as a open question to investigate in the future.

5.2. Parallel composition and arrow axioms

We used the arrow interface to build the embedded language, but, does **`FlowArrow`** satisfy all the arrow axioms? A quick check of the arrow axioms [9] shows that the *exchange* axiom does not seem to hold. Let f have the flow type $l_1 \rightarrow l_2$ and g have the flow type $l_3 \rightarrow l_4$. There are two canonical ways to compose f and g in parallel using the **`first`** combinator. They should be equivalent:

`first` f **`>>> pure (id`** $\times$ g**`) = pure (id`** $\times$ g**`) >>> first`** f

However, our **`FlowArrow`** implementation yields the flow type $l_1 \rightarrow l_4$ with constraints $\{l_2 \sqsubseteq l_3\}$ on the left side and $l_3 \rightarrow l_2$ with $\{l_4 \sqsubseteq l_1\}$ on the right. This seems to violate the arrow axioms! Does our implementation make sense?

If we compose f and g naturally using the **`(***)`** operator, we get $l_1 \sqcap l_3 \rightarrow l_2 \sqcup l_4$ with no constraints, which is the least restrictive flow type. The types we get from using **`first`** are both more restrictive than this one. The problem with using **`first`** is that our analysis technique is not fine-grained enough—it reasons about information flow in a syntax-directed, end-to-end fashion that yields imprecise flow types for **`first`** f **`>>> pure (id`** $\times$ g**`)`**. This coarse analysis does not compromise the security guarantee because it always conservative.

To justify that our arrow implementation satisfy the arrow axioms, we would need to give finer semantic interpretations to the flow types and constraints. Intuitively, although both sides of the *exchange* axiom are over-restrictive and have different types, they can be considered equivalent in the sense that they are both *sound*: using such flow types will not lead to acceptance of insecure programs.

The practical ramification of this imprecision is that although soundness is not affected by using **`first`**, programmers are encouraged to use the **`(&&&), (***), (|||), (+++)`** operations directly so that safe programs are more likely to be accepted. The use of **`first, second, left, right`** should be avoided whenever possible. Compared to **`first`**, **`(&&&)`** is a more intuitive operation for parallel composition because it resembles the product morphism in category theory. The prior work on arrows [5] uses **`first`** as the primitive operation because it is simpler and it gives definite evaluation orders.

A consequence of this imprecision is that the security analysis can be too restrictive for arrow computations written in the **`do`**-syntax, because Haskell implements some translation rules using **`first`** instead of **`(&&&)`**. Fortunately, conditional branches are translated using **`(||||)`**, so programmers can still write conditional branches in the natural way. In general, we need a more precise type system to avoid depending on particular implementations of the translation rules of the **`do`**-syntax.

5.3. DLM and practical applications

The declassification mechanism in this paper can be adapted to work with the *decentralized label model* (DLM) [7], where the constraints on code authority are expressed using the act-for relation of principals. We are currently working on the encoding and integration of DLM in the arrows framework. Unlike Jif, where information-flow control are mostly-static, the arrows framework is a run-time mechanism, so the principals, the act-for hierarchy and the security lattice can all be dynamic. Such dynamic policies have long been sought in language-based information-flow security because they address practical requirements.

Once the dynamic DLM is implemented, it will be interesting to see how it works in real applications. An important benefit of our approach is that existing Haskell applications can be enhanced with information-flow control without complete rewriting. The programmers may proceed gradually by changing the representation of secure components while leaving most normal components untouched. It would be ideal if the security-sensitive computation only takes a small portion of the whole program, so information-flow policies can be globally enforced by a few local modifications to the program.

As mentioned in Sec. 4.6, there are still interesting open questions on the protection and revocation of code privileges. The dynamic checking makes debugging more difficult because run-time errors are hard to observe, reproduce and locate. All these problems need to be explored in the contexts of concrete applications.

5.4. Implementing other type systems

Although **`FlowArrow`** is a generic arrow transformer, the type system implemented in **`FlowArrow`** only works with arrows that have no side-effects, because we assign a simple

information-flow type $l_1 \rightarrow l_2$ to such arrow computations. This brings up two questions. First, what arrows can be used besides the function arrow `(->)`? The stream processor [5] is an example: we can use `FlowArrow` to track information flow for stream processors, which map input streams to output streams. But in general, we need to formally state the properties of arrows that can be used with `FlowArrow`. Another question is: how can we modify the type system in `FlowArrow` so that it could work for states and other side-effects? For a specific effect such as memory states, we can extend the type system implemented in `FlowArrow` and lift the special arrow operations such as `get` and `put` to `FlowArrow` while implementing appropriate typing rules. By designing multiple versions of `FlowArrow`, we can implement multiple security type systems, and they can be used in one application at the same time.

The technical development in this paper is informal, although we have implemented it in Haskell. The type system implemented in `FlowArrow` can easily be justified following standard techniques [16]. For more complex type systems, however, justifying correctness often requires formal reasoning. The security goal is usually formalized as *noninterference* properties and the soundness is proved in a language with well-defined formal semantics. To use such a type system in the arrow framework, we must verify that the semantics of the arrow sublanguage matches the semantics of the toy languages used in the type soundness proofs.

6. Conclusion

Using an embedded sublanguage of arrows, end-to-end information-flow policies can be directly encoded and enforced in Haskell using modular library extensions, with a modest overhead of run-time checking. There is no need to modify the Haskell language, and this embedded sublanguage approach permits the information-flow technology to be adopted gradually. The security mechanism is designed to be generic with respect to computations types and security lattices. There is great flexibility in the choice of security policy frameworks; multiple policy frameworks can coexist in the same program. Dynamic information-flow policies can be expressed, yet the security guarantee is as strong as that of static analysis.

Acknowledgments

We thank Benjamin Pierce, Geoffrey Washburn, Stephanie Weirich, the other members of the PL Club at the University of Pennsylvania and the anonymous reviewers for their valuable suggestions and feedback about our work. Funding for this research was provided in part by NSF grants CNS-0346939 and CCF-0524035, and grant N00014-04-1-0725 provided by the ONR.

References

[1] The Glasgow Haskell Compiler. http://www.haskell.org/ghc.

[2] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, January 1999.

[3] Roderick Chapman and Adrian Hilton. Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters*, XXIV(4):39–46, 2004.

[4] Andrew C. Myers et. al. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif.

[5] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[6] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[7] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[8] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[9] Ross Paterson. Arrows and computation. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 201–222. Palgrave, 2003.

[10] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, Portland, Oregon, January 2002.

[11] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[12] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 255–269, 2005.

[13] Vincent Simonet. Flow Caml in a nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Nottingham, United Kingdom, March 2003.

[14] Stephen Tse and Steve Zdancewic. Run-time Principals in Information-flow Type Systems. In *Proc. IEEE Symposium on Security and Privacy*, 2004.

[15] Stephen Tse and Steve Zdancewic. Translating Dependency into Parametricity. In *ACM International Conference on Functional Programming*, 2004.

[16] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[17] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proceedings of the Second Workshop on Formal Aspects in Security and Trust (FAST2004)*, 2004.