Information Flow Enforcement in Monadic Libraries

Dominique Devriese
K.U.Leuven, Belgium
dominique.devriese@cs.kuleuven.be

Frank Piessens
K.U.Leuven, Belgium
frank.piessens@cs.kuleuven.be

Abstract

In various scenarios, there is a need to expose a certain API to client programs which are not fully trusted. In cases where the client programs need access to sensitive data, confidentiality can be enforced using an information flow policy. This is a general and powerful type of policy that has been widely studied and implemented.

Previous work has shown how information flow policy enforcement can be implemented in a lightweight fashion in the form of a library. However, these approaches all suffer from a number of limitations. Often, the policy and its enforcement are not cleanly separated from the underlying API, and the user of the API is exposed to a strongly and unnaturally modified interface. Some of the approaches are limited to functional APIs and have difficulty handling imperative features like I/O and mutable state variables. In addition, this previous work uses classic static information flow enforcement techniques, and does not consider more recent dynamic information flow enforcement techniques.

In this paper, we show that information flow policies can be enforced on imperative-style monadic APIs in a modular and reasonably general way with only a minor impact on the interface provided to API users. The main idea of this paper is that we implement the policy enforcement in a monad transformer while the underlying monadic API remains unaware and unmodified. The policy is specified through the lifting of underlying monad operations.

We show the generality of our approach by presenting implementations of three important information flow enforcement techniques, including a purely dynamic, a purely static and a hybrid technique. Two of the techniques require the use of a generalisation of the `Monad` type class, but impact on the API interface stays limited. We show that our technique lends itself to formal reasoning by sketching a proof that our implementation of the static technique is faithful to the original presentation. Finally, we discuss fundamental limitations of our approach and how it fits in general information flow enforcement theory.

Categories and Subject Descriptors D.4.6 [*Security and Protection*]: Information flow controls

General Terms Security, Languages

Keywords information flow enforcement, monads, monad transformer, parameterised monads

TLDI'11, January 25, 2011, Austin, Texas, USA.

1. Introduction

In various scenarios, there is a need to expose a certain API to a program without fully trusting it. Examples are JavaScript applications running in a web browser, plug-ins in user applications or even independent programs in a general-purpose OS. In order to protect against faulty or malicious use of the API, policies are defined establishing what constitutes valid use of the API. Such a policy is then enforced statically (type systems or other compile-time analysis) or dynamically (run-time monitoring).

Information flow policies are an important general class of policies. Given an ordered set of security levels, and a classification of all input and output operations at certain security levels, such a policy mandates that input at a certain security level must not in any way influence output at lower security levels. In the common case with only two security levels, secure information flow mandates that confidential information does not leak to public outputs.

It has been shown [13, 19, 22] that enforcement of information flow policies can be implemented in a lightweight fashion in the form of a library. However, these approaches suffer from a number of limitations. Often, the policy and its enforcement are not cleanly separated from the underlying API, and the user of the API is exposed to a strongly and unnaturally modified interface. Some of the approaches are limited to functional APIs and have difficulty handling imperative features like I/O and mutable state variables. In addition, all of these libraries use a form of static analysis (either at compile-time or at runtime). None of them show how to implement the dynamic information flow enforcement techniques that are increasingly being studied in information flow research [2, 4, 12, 21].

In this paper, we present a general technique for enforcing information flow policies in monadic libraries in Haskell. The enforcement of the policy occurs in a monad transformer, wrapping the monadic API of the underlying library, which remains unaware and unmodified. Policies are specified through the lifting of base monad operations into the transformed monad. We show that our technique is suited for implementing a wide range of enforcement techniques, including dynamic, static and hybrid ones. We show that this is possible while maintaining most of the "look and feel" of the underlying monadic library API.

Contrary to previous work, we do not limit ourselves to information flow enforcement techniques based on static analysis. We present implementations of three important information flow enforcement techniques, including a purely dynamic, a purely static and a hybrid technique. The two latter techniques require some form of static analysis of user code, which we implement using a generalisation of the $Monad$ concept, called parameterised monads [9]. This allows us to type the monadic operations $return$, $(\gg)$ (sequence) and $(\gg\!=)$ (bind) differently, allowing us to perform the static analysis we need without relying on explicit developer-provided casts.

In section 7, we provide a formal result and a proof sketch demonstrating the faithfulness of our account of Volpano et al.'s classic type system-based enforcement [24], which we implement in section 6. We show that we can translate programs in a model language extended with output statements to calculations in our monad transformer, where the typing judgements of a straightforward extension of Volpano et al.'s type system correspond in a precisely defined manner to the types of our transformed monad calculations. We do not expect corresponding results for our other implementations to be significantly harder to formulate and prove.

Finally, we look into the limits of our approach. In the context of dynamic information flow enforcement, we can think of our approach as a type of monitor where the monitorable events are the monadic sequence ($\gg$) and bind ($\gg=$) operators. We discuss that for at least one recent dynamic information flow enforcement technique [2], these monitorable events do not suffice, making it impossible to implement the technique as a monad transformer.

In summary, the contributions of this paper are:

- We propose an approach to implementing information flow policy enforcement for monadic libraries. Contrary to existing work, it allows for a clean separation between policy enforcement and the underlying API, and does not impose unnatural modifications of the interface provided to users of the library.
- We provide evidence for the generality of the approach by presenting implementations of three information flow enforcement techniques, including a dynamic, a static and a hybrid technique, where existing libraries only work with static analysis. We are not aware of any previous implementations of dynamic information flow enforcement as a library.
- We give a formal statement of the faithfulness of our account of a static enforcement technique to the original presentation, and provide a proof sketch.
- We discuss the limitations of our approach, in which all compile-time or run-time interception takes place at the level of monadic operations, making it unsuitable for at least one dynamic enforcement technique [2].

2. The basic idea

In order to explain the basic design of our library, we show an example of a monadic API on which we want to enforce an information flow policy.

2.1 An E-mail Client Example

Our example API starts from a simplistic scenario where an e-mail application provides a plug-in infrastructure allowing user- or institution-provided plugins to intercept e-mails while they are being sent. Plug-ins are allowed to modify e-mails (e.g. in order to add quotes or disclaimers at the bottom of the e-mail), send the mail to multiple recipients (e.g. to the original recipient and to an archiving system). In order to do this, they are allowed to retrieve external resources (e.g. a company standard disclaimer or a quotes database), identified by URLs.

In order to prevent malicious plug-ins from leaking users' private information through resource requests, an information flow policy is defined. The result of any $readMail$ operation is considered confidential information, and it is only allowed to influence $sendMail$ operations, which are considered as output at the confidential security level (note that malicious plug-ins could send e-mails to other recipients than the original, but we assume for simplicity that this is prevented in some other way). Invocations of $downloadResource$ and the URLs passed to them are considered public, as they can be observed on the network or by the resource server owners. The result of $downloadResource$ calls is public information. With this classification of base monad operations at security levels, an information flow policy is defined, stating that the contents of a user e-mail must not in any way influence requests for external resources. In further sections, we show how to specify and enforce this policy using different enforcement monad transformers.

```
class (Monad m) ⇒ EmailM m where
  readMail :: m String
  sendMail :: String → m ()
  downloadResource :: String → m String
instance EmailM IO where
  sendMail = putStrLn ∘ ("Sending mail: "++)
  readMail = putStr ": " ≫ getLine
  downloadResource = (≫return "prefix; ") ∘
    putStrLn ∘ ("Downloading resource: "++)
instance (EmailM m) ⇒ EmailM (StateT a m) where
  sendMail = lift ∘ sendMail
  readMail = lift readMail
  downloadResource = lift ∘ downloadResource
```

Figure 1. An example API presented to plug-in developers in an e-mail application, along with an example implementation in the IO monad. We also provide classic automatic lifting of the $EmailM$ operations through the state monad transformer.

The API presented to plug-in developers is shown in Figure 1. The API is defined in the $EmailM$ type class. This type class extends the Monad type class and provides $readMail$, $sendMail$ and $downloadResource$ operations. The $readMail$ operation returns the contents of a mail input by the user, the $sendMail$ operation ships off a mail to be sent and the $downloadResource$ operation takes the URL of a resource and returns its contents. The figure also shows a model implementation in the IO monad and an automatic lifting of the $EmailM$ type class through a $StateT$ monad transformer (similar to how this is done with e.g. the standard $MonadReader$ type class).

2.2 Information flow enforcement in a Monad transformer

The basic observation behind our implementation approach for information flow enforcement libraries is that in a monadic API, a lot can be learned about the program's information flow from the invocations of $Monad$ operations $\gg$ and $\gg=$. For example, when two calculations a and b are sequenced ($a \gg b$), we are sure that the result value of a can not influence computation b. On the contrary, when a monadic calculation a is bound to a continuation f ($a \gg= f$), we can interpret this as a generalised form of a branching statement, where the result value of a determines the calculations that will be executed next.

The next observation we make is that intercepting monadic operations in a base monad is precisely what Haskell's $Monad$ transformers allow us to do. And with this second observation, the basis of our approach is clear: we will implement information flow enforcement techniques as monad transformers and use the information we can obtain from intercepted $Monad$ operations to approximate the flow of information in a program.

There is one important problem with this approach however, namely that the calls to $Monad$ operations do not always correctly represent the actual information flow in a program. For example, in the program $\mathbf{do}\ x \leftarrow getLine; return\ ()$, it would be wrong to assume that the result of $getLine$ influences the result of the combination. Nevertheless, a Haskell compiler will "desugar" the example to $getLine \gg= \lambda x \rightarrow return\ ()$, and our monad trans-

```
badflow = do sec ← getSec
             putSecret sec
             pub ← getPub
             putPublic $ pub ++ ", done."

goodflow = do sec ← getSec
              putSecret sec
          >> do pub ← getPub
              putPublic $ pub ++ ", done."
```

Figure 2. An example requiring non-standard use of the do-notation to provide proper scoping information for use in the information flow enforcement. In the expression *badflow* it will not be clear to the enforcement transformer that the information coming from *getSec* is not used in the call to *putPublic*. The scoping in the expression *goodflow* makes this clear.

formers will have to assume that the result value of *getLine* is used by the function $\lambda x \rightarrow return\ ()$, leading to an overapproximation of the information flow in the program. The solution we will adopt to this problem is that we impose the requirement on users of the API to accurately reflect the flow of sensitive information in their programs through a proper scoping of variables. For example the example above should instead be written as **do** *getLine*; *return* () or $getLine \gg return\ ()$. In this case, the use of $\gg$ instead of $\gg=$ allows the enforcement technique to detect that the result of the left-hand side calculation is not used in the other one. Figure 2 shows a more complicated example where two separate do-statements are explicitly sequenced to correctly reflect the example's information flow in the structure of the *Monad* operation calls.

Note that a consequence of this approach is that we do not treat a sequence of two calculations $ma \gg mb$ as identical to $ma \gg= const\ mb$, contrary to what Haskell programmers are used to. However, we want to point out that sematically identical constructs are often treated differently by information flow enforcement techniques. Consider how $x := s - s$ would be treated differently from $x := 0$ by most techniques or how **if** *sec* **then** $x :=$ true **else** $x :=$ false would be treated differently from $x :=$ *sec* by Austin and Flanagan's enforcement technique [2]). This imprecision is inherent to enforcement techniques employing the syntactic structure of programs, rather than their actual behaviour. Devriese and Piessens have recently presented a fundamentally different technique which does not suffer from this problem [4].

2.3 Telling the good from the bad

Figures 3 and 4 show examples of plug-ins using the *EmailM* interface. Both employ explicit sequencing of two separate do-statements, as discussed in the previous section. We will use these examples throughout this paper to demonstrate our implementation.

The first example (*plugin1*) shows a plug-in which prefixes the user mail with the contents of an external resource, appends a newline and sends the mail. No information about the mail leaks to the resource server, so this plug-in respects the information flow policy. On the other hand, *plugin2* will only download the external resource if a certain property holds for the user e-mail, thus violating the policy: the owner of the resource server can deduce that the user sent a mail satisfying the *isInteresting* property. Note that in this case, information is only leaked because the request for the external resource is issued in a branch on secret data, which is sometimes called an "implicit" leak in the information flow literature. If we replace the if-then-else block with $downloadResource\ \$\ res\ ++\ \texttt{"?q="}\ ++\ escape\ m$, then there is an "explicit" leak of information.

```
res :: String
res = "http://example.com/res.txt"
plugin1 :: (EmailM m, MonadState String m) ⇒ m ()
plugin1 = do m ← readMail
             put $ m ++ "\n"
          >> do p ← downloadResource res
             m ← get
             sendMail $ p ++ m
```

Figure 3. Example plug-in *plugin1*, respecting the information flow policy.

```
isInteresting :: String → Bool
isInteresting = elem "Haskell" ∘ words
plugin2 :: (EmailM m, MonadState String m) ⇒ m ()
plugin2 = do m ← readMail
             put $ m ++ "\n"
          >> do m ← get
             if isInteresting m
               then downloadResource res
               else return ""
             sendMail m
```

Figure 4. Example plug-in *plugin2*, which does not respect the information flow policy, because it leaks information about the user mail to the external resource server.

With these two examples, we have set the objective for the rest of this text. In the next sections, we implement three different monad transformers, implementing information flow enforcement techniques to be able to detect illegal information flows, such as the one in *plugin2* without interfering with correct plug-ins like *plugin1*.

3. Dynamic information flow enforcement with security level ascriptions

The first information flow enforcement technique we implement is Sabelfeld and Russo's simple flow-insensitive dynamic enforcement [21]. They employ a fixed programmer-provided assignment of security levels to variables. With this information, they can deduce a security level for each expression, and track the security level of all information in the program. In order to prevent implicit leaks, they additionally keep a stack of security levels for the "program counter", representing the maximum security level of all information that determined the currently executing branch to be chosen. This allows them to detect implicit information leaks when public output is produced in a branch whose execution was determined by confidential information. It's a simple technique, so we do not go further into the details, but we think they will quickly become clear in our implementation.

Before we start, we need a type to represent security levels at run-time.

$$\textbf{data}\ DynSL = L \mid H\ \textbf{deriving}\ (Show, Eq, Ord)$$

In order to implement Sabelfeld and Russo's monitor in a monad, we represent a calculation in the monad as a function of type $DynSL \rightarrow m\ (DynSL, a)$. The calculation takes as input the

current security level of the program counter, executes in the base monad and returns not just the value, but also the security level of the calculated result.

```
newtype DynFlowSabT m a = DynFlowSabT {
  runDynFlowSabT :: DynSL → m (DynSL, a)
  }
```

It is interesting to note the similarity with a StateT transformer monad carrying the program counter security level around as its state. This similarity does not carry through however in the instance of the *Monad* type class.

```
instance (Monad m) ⇒
  Monad (DynFlowSabT m) where
    return v = DynFlowSabT $ λpc → return (pc, v)
    x ≫ y = DynFlowSabT $ λpc →
      runDynFlowSabT x pc ≫
      runDynFlowSabT y pc
    x ≫= f = DynFlowSabT $ λpc →
      do (l, v) ← runDynFlowSabT x pc
         let xo = f v
         runDynFlowSabT xo $ max pc l
```

The essence here is the security levels of the result of combined calculations and the security level of the program counter passed to the components. We make an essential difference between the sequence (≫) and bind (≫=) operators. The first will simply drop the result of a left hand side calculation and not pass it on to the right hand side, so the result of the combined calculation is considered at the same security level as the right hand side result. Both are executed at the same program counter level. The bind operation is more complicated, as the value of the left calculation can determine not just the result of the right calculation, but can even result in a completely different calculation to be combined. Therefore we need to augment the program counter security level passed to the second calculation to at least the security level of the result of the first calculation. This perfectly reflects the rules for branching constructs in Sabelfeld and Russo's monitor. The return operation simply returns the program counter security level it is passed as the security level for its result.

With these ~20 lines of code, our implementation of Sabelfeld's monitor is almost ready. What we still need is a custom lifting operation which will enforce the security policy on a base monad calculation. It is important to note that two security levels need to be specified. First of all, if a base monad operation produces output visible at a certain security level, then we need to make sure it is not executed when the program counter is at a higher security level, because that would constitute an information leak. In addition, if the base monad calculation produces a result value, then we need to assign a correct security level to it, so that it cannot be leaked in the rest of the program. We will call these two security levels respectively the "input" (*li*) and "output" (*lo*) security levels and require them to be specified when a base monad operation is lifted.

```
liftAt :: (Monad m) ⇒ DynSL → DynSL → m a →
    DynFlowSabT m a
liftAt li lo c =
  let levelError pc =
    "Output at level " ++ show li ++ " with " ++
    "program counter at level " ++ show pc ++ "!"
  in DynFlowSabT $ λpc →
    if pc ⩽ li then do v ← c; return (lo, v)
               else fail $ levelError pc
```

The functions *readMail*, *downloadImage*, *sendMail* and *get* and *put*, can now be lifted. The function *downloadResource* produces public effects and a public result, so it is lifted using the function *liftAt L L*. On the contrary, *readMail* produces no publicly visible effects, so it can be executed with a high program counter level, but it produces a confidential result, so we lift with *liftAt H H*. The function *sendMail* is allowed to take confidential input and produces a public (unit value) result, resulting in *liftAt H L*. The String state variable is assigned a confidential value, leading to respectively *H H* and *H L* as input and output security levels for the *get* and *put* functions. Note that because of the workings of the *MonadState* type class, we are limited to a single state variable. We have a version of the code where we remove this restriction and allow the user to allocate extra state variables using a monad transformer version of Launchbury et al.'s *ST* monad [10], but we are not going further into this because of space limitations and because it is unclear to us how and if this can be extended to the technique we will implement in section 5.

```
instance (EmailM m) ⇒
  EmailM (DynFlowSabT m) where
    downloadResource = liftAt L L ∘ downloadResource
    readMail = liftAt H H readMail
    sendMail = liftAt H L ∘ sendMail

instance (MonadState String m) ⇒
  MonadState String (DynFlowSabT m) where
    get = liftAt H H get
    put = liftAt H L ∘ put

runplugin1 = evalStateT
  (runDynFlowSabT plugin1 L) ""
runplugin2 = evalStateT
  (runDynFlowSabT plugin2 L) ""
```

When we now execute the examples in GHCi, we see that our simple monitor works as intended, and succeeds in detecting and preventing the illegal information flow, while not modifying those executions which respect the information flow policy.

```
*Main> runplugin1
: Haskell invented currying?
Downloading resource: http://example.com/res.txt
Sending mail: prefix; Haskell invented currying?

*Main> runplugin2
: Haskell invented currying?
*** Exception: user error (Output at level L
      with program counter at level H!)
*Main> runplugin2
: Some other mail contents...
Sending mail: Some other mail contents...
```

Our first information flow monitor monad transformer is ready. We have defined a basic dynamic information flow monitor as a monad transformer and used a special lifting function to specify security levels for base monad operations. In sections 5 and 6, we show that we can similarly implement two other information flow enforcement techniques. But before we continue, we first need to introduce a technique called parameterised monads, which we require for the implementation of static analysis on the user code.

4. Parameterised Monads

Monads in Haskell are defined through the *Monad* type class, shown in Figure 5. It defines the *return*, *fail*, (≫) and (≫=) operations, and their type signature. For some applications, this type signature is too restrictive. In our case, the problem is that for the two enforcement techniques that we will discuss next, we

```
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
  (>>)  :: m a → m b → m b
  fail  :: String → m a
```

Figure 5. Definition of Monad class and its operations from the standard Haskell Prelude.

```
import qualified Control.Monad as Old
class Sequence m m' m'' | m m' → m'' where
  (>>) :: m a → m' b → m'' b
class Bind m m' m'' | m m' → m'' where
  (>>=) :: m a → (a → m' b) → (m'' b)
class Fail m where
  fail :: String → m a
class Return m where
  returnM :: a → m a
```

Figure 6. The *Sequence*, *Bind*, *Fail* and *Return* type classes, as used in this paper instead of the standard *Monad* class.

need to be able to keep some compile-time data about calculations in information flow enforcing monads. We will represent this data in the type of the monadic calculation, but for this, we need to be able to combine values of different types using the monadic combinators. Because this is impossible with the standard *Monad* type class, we turn to a technique that allows us to bypass this restriction: parameterised monads.

Various forms of parameterised monads have been introduced, but for our purposes, we will use a library developed by Edward Kmett [9]. The basic idea in this library is to use the GHC NoImplicitPrelude [6] Haskell extension to redefine the monadic operators in separate type classes, and to use the GHC MultiParamTypeClasses and FunctionalDependencies extensions to allow different monadic types to be combined and derive the result type from the combined types. We go slightly further than Kmett even, by also separating $(\gg)$ from $(\gg=)$ in separate type classes. Our definitions can be seen in Figure 6.

An important consideration when using these parameterised monads is the impact on type inference. Because monadic calculations of different types can now be combined, the compiler can no longer infer the type of one of the arguments of the $(\gg)$ or $(\gg=)$ operators from the other. This would be especially problematic for the *return* operator, for which the type is almost always determined by the calculations it is sequenced or bound with, so if it were typed as $a \to m\ a$, then explicit programmer type ascription would always be required to determine the type m.

Kmett instead proposes to type *return* as $a \to Identity\ a$ with *Identity* the standard identity monad. In this way, less type inference is required to determine the type of *return* calls, and *return* calculations are now combined with a monad through the the $(\gg)$ and $(\gg=)$ operators with their extended type signature. An old-style *return* call is still available in the new type class *Return* under the name *returnM* and standard definitions for $Bind\ Identity\ m\ m$, $Bind\ m\ Identity\ m$, $Sequence\ m\ Identity\ m$ and $Sequence\ Identity\ m\ m$ are defined using it. The resulting definitions can be seen in Figure 7.

It turns out that in practice, this "trick" can make parameterized monads feel quite natural to programmers used to standard monads

```
return :: a → Identity a
return = Old.return
instance Bind Identity m m where
  lm >>= f = f (runIdentity lm)
instance (Return m, Bind m m m) ⇒
    Bind m Identity m where
  lm >>= (f :: a → Identity a') =
    lm >>= (returnM ∘ runIdentity ∘ f :: a → m a')
instance Bind Identity Identity Identity where
  m >>= f = f $ runIdentity m
instance Sequence Identity m m where
  lm >> rm = rm
instance Sequence Identity Identity Identity where
  lm >> rm = rm
instance (Return m, Sequence m m m) ⇒
    Sequence m Identity m where
  lm >> (rm :: Identity a') =
    lm >> (returnM $ runIdentity rm :: m a')
```

Figure 7. Kmett's definition of the *return* function returning an instance of the *Identity* monad, behaves better w.r.t. type inference. Special instances of the *Bind* and *Sequence* type classes make sure it interacts well with other monads.

in Haskell. Also important in this respect is that GHC supports the do-notation for parameterised monads since version 6.8.3 [5], by simply executing the canonic translation for **do**-blocks in function of whatever versions of *fail*, $(\gg)$ and $(\gg=)$ are in scope.

5. Dynamic enforcement, without annotations.

The second information flow enforcement technique we implement is a hybrid (dynamic, supported by static analysis) flow-sensitive technique developed by Le Guernic et al.[12] The technique is flow-sensitive, indicating that variables can be assigned different security levels throughout the execution of a program. The security levels of variables are inferred by the enforcement technique and do not have to be provided by the programmer.

To present the technique, it is instructive to first consider a more naive attempt at getting rid of the security level annotations. Naively, it seems possible to just infer an appropriate security level for the information stored in a variable every time a value is assigned to it. An appropriate security level could be calculated as the maximum of the level of the assigned value and that of the program counter. We can implement special lifting functions that do this in our DynFlowSabT monad transformer as follows.

```
instance (MonadState (DynSL, Bool) m) ⇒
  MonadState Bool (DynFlowSabT m) where
    get = DynFlowSabT $ const get
    put v = DynFlowSabT $ λpc →
      do put (pc, v)
         return (L, ())
```

Now, this is not very difficult to implement, but unfortunately, this approach is not sound, as can be seen in the following example.

```
leak = do put True
          m ← readMail
          let p = isInteresting m
          if ¬ p then put False else return ()
       >> do p ← get
          if ¬ p
```

```
            then return ""
            else downloadResource res
          return ()
runleak :: (EmailM m) ⇒ m (DynSL, ())
runleak = evalStateT (runDynFlowSabT leak L) (L, False)
```

As we can see in GHCi, the *leak* plugin will download the resource identified by URL *res* if a certain property of a user e-mail holds, thus leaking one bit of secret information and bypassing our information flow policy.

```
*Main> runleak
: Some other mail contents...
(H,())
*Main> runleak
: Haskell invented Currying?
Downloading resource: http://example.com/res.txt
(L,())
```

The reason that our monitor fails to detect the illegal information flow in this example lies in what happens in the first if-statement. There, if p is $False$, then the branch $put\ False$ will be chosen, and our lifted put function will correctly mark the variable as confidential. However, in the case that p is $True$, no assignment takes place, and the security level for the state variable is not updated, even though its new value does depend on confidential data. Afterwards, our monitor does not consider it a problem when we make a request for a public resource in a branch on this variable, because it is marked as non-confidential. One could say that the information leak occurs through the non-assignment of a variable in a conditional branch that is not executed.

This example may seem relatively harmless at first sight, but by adapting the code, the trick can be exploited to consistently leak any amount of information, not just a single bit. In fact, our example is an adaptation from a classical example that plays an essential role in Russo and Sabelfeld's proof of the non-existence of a sound and precise flow-sensitive purely dynamic monitor [18].

Russo and Sabelfeld's result implies that a sound and precise flow-sensitive monitor must use some form of static analysis of user code. The classic example of this solution is the monitor developed by Le Guernic et al. [12]. They propose to tackle the problem by combining the dynamic tracking of state variables' security level with such a static analysis and an additional rule for conditional statements like **if** and **while**. After executing the chosen branch of such a statement, the monitor will in addition update the security level of all variables which could possibly have been assigned in the other branch of the conditional. This set of variables is precalculated using a compile-time static analysis and for all of these variables, the security level is increased to at least the security level of the program counter and the conditional expression. Le Guernic et al. prove that this idea effectively solves the problem and guarantees secure information flow.

Clearly, what is difficult to implement about this technique in our framework, is the static analysis. First, it is not straightforward to perform such a static analysis outside of the compiler, but secondly, we also have to generalise the rules defined by Le Guernic et al. to our monadic interception facilities. For the second problem, it turns out that Le Guernic's technique adapts nicely to our handling of the monadic binding operator $(\gg\!=)$ as a generalised branching construct. For a binding of a calculation m to a continuation f $(m \gg\!= f)$, we will perform a static analysis of f to determine the complete set of variables that can be assigned by it, and update their security levels to at least the level of the conditional expression and the program counter. This update can be sequenced after the continuation, just like Le Guernic et al. execute the updates after the chosen branch of an **if** expression.

In order to solve the first problem (actually performing the static analysis of the continuation f), we apply a simple form of effect typing to our monadic operations. For any variable used, we define a ghost type, and require that it be an instance of the type class $LGStateEffect$. The function $updateLabel$ in this class takes a dummy value of the ghost type and a security level, and returns a calculation that increases the variable's security level to at least the given level. Note how the functional dependency $st \to m$ couples a token to the monad in which the state variable is kept.

```
class LGStateEffect st m | st → m where
  updateLabel :: st → DynSL → m ()
```

Because a piece of code can clearly update more than a single state variable, we need a type-level representation for lists of tokens. We define a type Nil representing the empty list and a type $Cons$ parameterised by a first element of a list and a tail list. Using the GHC MultiParamTypeClasses, FunctionalDependencies, OverlappingInstances, UndecidableInstances and TypeFamilies Haskell extensions, we define type families Add and Merge, respectively adding a $token$ to a list of tokens and merging two lists, removing duplicate entries along the way. We also use the EmptyDataDecls and TypeOperators extensions for some syntactic sugar.

```
data Nil
data Cons a r
type Singleton a = Cons a Nil

class Add a b c | a b → c
instance Add Nil a (Cons a Nil)
instance Add (Cons a r) a (Cons a r)
instance (Add a r r') ⇒ Add (Cons b r) a (Cons b r')

class Merge a b c | a b → c
instance Merge Nil l l
instance (Add l a l', Merge r l' l'') ⇒
  Merge (Cons a r) l l''
```

We can now instantiate the LGStateEffect class for lists of effects (using the ScopedtypeVariables Haskell extension).

```
instance LGStateEffect Nil Identity where
  updateLabel _ _ = return ()
instance (LGStateEffect t m, LGStateEffect t' m',
      Sequence m m' m'') ⇒
  LGStateEffect (Cons t t') m'' where
    updateLabel tok l =
      updateLabel (⊥ :: t) l ≫
      updateLabel (⊥ :: t') l :: m'' ()
```

The $DynFlowLGT$ monad transformer is identical to the previous $DynFlowSabT$, except for the annotation with the effect type tag:

```
data DynFlowLGT t m a = DynFlowLGT {
  runDynFlowLGT :: DynSL → m (DynSL, a)
}
```

With this, we define the function $otherBranchExec$, which returns a calculation updating the security level for all variables for which assignments occur in a block of code to at least a given security level.

```
otherBranchExec :: forall t m mt a ∘
  (LGStateEffect t mt)
  ⇒ DynFlowLGT t m a → DynSL → mt ()
otherBranchExec _ l = updateLabel (⊥ :: t) l
```

The monadic combinators $(\gg)$ and $(\gg\!=)$ now perform several tasks. In their type signature, they propagate and merge the effect type tags. Their implementation performs the same function as for

the *DynFlowSabT* transformer, and in addition the combinator ($\gg\!=$) employs the *otherBranchExec* function to update security levels for all variables that can possibly assigned in its right hand side continuation. Evidently, for the propagation of effect tokens, these definitions depend crucially on the parameterized monads introduced before.

```
instance (Sequence m m' m'', Merge t t' t'') ⇒
  Sequence (DynFlowLGT t m) (DynFlowLGT t' m')
           (DynFlowLGT t'' m'')
  where (DynFlowLGT ex) ≫ (DynFlowLGT ey) =
    DynFlowLGT $ λpc → ex pc ≫ ey pc
instance (Bind m mt'' m'', Merge t t' t'',
  Bind m' mt' mt'',
  Sequence mt' Identity mt',
  LGStateEffect t mt, LGStateEffect t' mt') ⇒
  Bind (DynFlowLGT t m) (DynFlowLGT t' m')
    (DynFlowLGT t'' m'') where
  (DynFlowLGT x) ≫= f =
    DynFlowLGT $ λpc →
      do (ll, lv) ← x pc
         let rm = f lv
         result ← runDynFlowLGT rm $ max pc ll
         otherBranchExec rm $ max pc ll
         return result
instance (Return m) ⇒
    Return (DynFlowLGT t m) where
  returnM v = DynFlowLGT $ λpc → returnM (pc, v)
```

State lifting functions are now identical to the unsafe versions defined above, except that their results are tagged with the state token needed for the static analysis. Non-state operations are lifted as before, and get the empty list *Nil* as effect tag, indicating they do not affect any state.

```
liftGet :: m (DynSL, s) → DynFlowLGT t m s
liftGet g = DynFlowLGT $ const g

liftPut :: (Sequence m Identity m,
  LGStateEffect t m) ⇒
  ((DynSL, s) → m ()) → s
    → DynFlowLGT t m ()
liftPut p v = DynFlowLGT $ λpc →
  do p (pc, v)
     return (L, ())

liftAt :: (Fail m, Bind m Identity m) ⇒
  DynSL → DynSL → m a → DynFlowLGT Nil m a
liftAt li lo c =
  let levelError pc =
    "Output at level " ++ (show li) ++ " with " ++
    "program counter at level " ++ (show pc) ++ "!"
  in DynFlowLGT $ λpc →
    if pc ⩽ li
      then do v ← c; return (max pc lo, v)
      else fail $ levelError pc
```

The definitions of the API functions need to be changed because their type becomes more complicated with the effect annotations, and type inference is less powerful in the context of parameterised monads. Indeed, API providers need to more strongly fix the types of lifted base monad operations, to avoid requiring more type ascription from plug-in writers. Here, we reuse the names of the lifted operations, so that we can keep our previous examples unmodified (only a recompilation required). We use the old definitions through a qualified import under the prefix *Old*.

```
type BaseMonad = StateT (DynSL, String) IO
type MonadStack t a = DynFlowLGT t BaseMonad a

readMail :: MonadStack Nil String
readMail = liftAt H H Old.readMail
sendMail :: String → MonadStack Nil ()
sendMail = liftAt H L ∘ Old.sendMail
downloadResource :: String → MonadStack Nil String
downloadResource = liftAt L L ∘ Old.downloadResource
```

For the state operations, we need to define an effect token ghost type, include the effect token in the type of the put operation, and instantiate the LGStateEffect type class. Again, we reuse the previously used *get* and *put* names.

```
data VarAToken

get :: MonadStack Nil String
get = liftGet Old.get
put :: String → MonadStack (Singleton VarAToken) ()
put = liftPut Old.put
instance (MonadState (DynSL, s) m, Bind m m m) ⇒
  LGStateEffect VarAToken m where
  updateLabel _ pc = do
    (l, v) ← Old.get :: m (DynSL, s)
    Old.put (max l pc, v) :: m ()
```

We can now recompile and execute our example plug-ins *plugin1* and *plugin2* again. Note that we need to replace the call to *return* by a call to *returnM* because of the new type for *return* in our parameterised monads, and recompile the programs with the new definitions of the API.

```
plugin1 = do m ← readMail
             put $ m ++ "\n"
          ≫ do p ← downloadResource res
             m ← get
             sendMail $ p ++ m
plugin2 = do m ← readMail
             put $ m ++ "\n"
          ≫ do m ← get
             if isInteresting m
               then downloadResource res
               else returnM ""
             sendMail m
runplugin1 :: IO (DynSL, ())
runplugin1 = evalStateT
  (runDynFlowLGT plugin1 L) (L, "")
runplugin2 :: IO (DynSL, ())
runplugin2 = evalStateT
  (runDynFlowLGT plugin2 L) (L, "")
```

When we execute these examples in GHCi, we get the expected results, this time without having required programmer-provided security level annotations for state variables.

```
*Main> runplugin1
: Haskell invented currying?
Downloading resource: http://example.com/res.txt
Sending mail: prefix; Haskell invented currying?

*Main> runplugin2
: Haskell invented currying?
*** Exception: user error (Output at level L
      with program counter at level H!)
*Main> runplugin2
: Some other mail contents...
Sending mail: Some other mail contents...
```

Similarly, for an omitted adaptation of the *leak* example from the beginning of this section, we see that Le Guernic's solution also detects the illegal information flow in that example:

```
*Main> runleak
: Haskell invented currying?
*** Exception: user error (Output at level L
      with program counter at level H!)
*Main> runleak
: Some other mail contents...
(H,())
```

Unfortunately however, Le Guernic's technique has only been shown to support lexically scoped stack variables, without aliasing. Features like heap data or even references cannot easily be supported, because they make it impossible in general to statically analyse the set of variables that could potentially be assigned by a branch of a conditional statement. In the context of real world API's in real world languages, we think this restriction can prove to be fairly strong and make the technique unusable.

6. Static enforcement

The third and last information flow enforcement technique we implement in this text is the classic compile-time flow-insensitive type-system based enforcement developed by Volpano in 1996 [24]. Like Sabelfeld's dynamic technique (see section 3), this technique requires programmer-provided security level annotations for variables and uses this to type all (pure) expressions at a certain security level (written $e : l$). A subtyping rule states that if $e : l$ and $l < l'$ then $e : l'$. In addition to that, statements' types represent the lowest security level on which their effects are visible. For example, an assignment s to a variable annotated with security level l will be typed $s : l\,\mathrm{cmd}$. Contrary to expression types, these statement types are subject to a contravariant subtyping, i.e. if $s : l\,\mathrm{cmd}$ and $l > l'$ then $s : l'\,\mathrm{cmd}$.

Using this typing information, enforcing secure information flow becomes relatively easy. In assignment or output statements, explicit flows are prevented by requiring the assigned expression to be associated to the variable's security level or higher. In branching statements, implicit flows are prevented by requiring the security level of the conditional expression to be lower or equal to the effect levels of any of its branches. We discuss all of this in more detail in section 7.

In this section, we implement this type system in a monad transformer, in such a way that the transformed monad behaves identically to the original at runtime, but the information flow type system is checked at compile-time. Making the security types part of the transformed monad calculations' types and providing custom types for the monad operations (as parameterized monads), allows us to implement the type checking in a relatively cheap way, making the Haskell compiler do most of the work for us.

Clearly, the first thing we need to do to achieve this, is to represent security levels at the type level. We use the Haskell extensions MultiParamTypeClasses, EmptyDataDecls, TypeFamilies, TypeOperators and UndecidableInstances to provide both powerful, convenient and elegant type-level computation primitives for working with our security level types. The *UndecidableInstances* extension is not generally accepted in the Haskell community, but we cannot avoid it for the static analysis we want to perform. We define the different levels we need as empty data types, and define a type class $\cdot \geqslant \cdot$. We also define type families $\cdot \wedge \cdot$ and $\cdot \vee \cdot$ (commonly known as join and meet operators using terminology from mathematical lattices), representing respectively a least upper bound and greatest lower bound of two security levels.

$$
\begin{array}{l}
\textbf{data}\ H \\
\textbf{data}\ L \\
\\
\textbf{class}\ ll \geqslant lr \\
\textbf{instance}\ H \geqslant H \\
\textbf{instance}\ H \geqslant L \\
\textbf{instance}\ L \geqslant L \\
\\
\textbf{type family}\ ll \wedge rl \\
\textbf{type instance}\ H \wedge sl = H \\
\textbf{type instance}\ sl \wedge H = H \\
\textbf{type instance}\ L \wedge L = L \\
\\
\textbf{type family}\ ll \vee rl \\
\textbf{type instance}\ H \vee H = H \\
\textbf{type instance}\ sl \vee L\ \ = L \\
\textbf{type instance}\ L \vee sl = L
\end{array}
$$

Before we continue, there is a slight extension that we need to make to Volpano et al.'s type system. Contrary to their model language, our monadic calculations can behave as both an expression and a statement. A base monadic calculation $m\ v$ produces a result of type v, but can also produce side effects in the background. This means that in the type for our transformed monad, we need to track both a covariant *result security level* representing the security level of the result of the calculation, and a contravariant *effect security level* representing the lowest level at which it produces side effects. We write this as $StaticFlowT\ lr\ le\ m\ a$ where m is the base monad, v the result type, lr the result security level and le the effect security level. Note that this elegant extension of Volpano et al.'s type system is not our contribution, but was previously investigated by Crary et al. [3] (see section 8). We look further into the correspondence between the classic types and ours in section 7.

Except for this annotation with security level types, our transformed monads are identical to the underlying base monad:

$$
\begin{array}{l}
\textbf{newtype}\ StaticFlowT\ lr\ le\ m\ v = StaticFlowT\ \{ \\
\quad runStaticFlowT :: m\ v \\
\quad \}
\end{array}
$$

We make our $StaticFlowT$ monad transformer an instance of the $MonadTrans$ class, so that a base monad operation can be lifted using the standard *lift* function.

$$
\begin{array}{l}
\textbf{instance}\ MonadTrans\ (StaticFlowT\ lr\ le)\ \textbf{where} \\
\quad lift\ c = StaticFlowT\ c
\end{array}
$$

Note how the lifting operation is fully polymorphic in the security levels. The idea here is that the caller of the lift function can simply use explicit type ascription to annotate base monad operations with correct security levels. We demonstrate this for the e-mail application plug-in API.

$$
\begin{array}{l}
\textbf{type}\ BaseMonad = StateT\ String\ IO \\
\textbf{type}\ MonadStack\ lr\ le\ a = \\
\quad StaticFlowT\ lr\ le\ BaseMonad\ a \\
\\
readMail :: MonadStack\ H\ L\ String \\
readMail = lift\ Old.readMail \\
sendMail :: String \to MonadStack\ L\ H\ () \\
sendMail = lift \circ Old.sendMail \\
downloadResource :: String \to MonadStack\ L\ L\ String \\
downloadResource = lift \circ Old.downloadResource
\end{array}
$$

For state variables, we lift *put* and *get* operations similarly.

$$
\begin{array}{l}
get :: MonadStack\ H\ H\ String \\
get = lift\ MS.get \\
put :: String \to MonadStack\ L\ H\ () \\
put = lift \circ MS.put
\end{array}
$$

The final task in implementing our static information flow enforcement monad transformer is defining the monadic operations:

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Bind}\ \mathit{lm}\ \mathit{rm}\ \mathit{fm}, \mathit{rle} \geqslant \mathit{llr}, \\
\quad \mathit{llr} \wedge \mathit{rlr} \sim \mathit{flr}, \mathit{lle} \vee \mathit{rle} \sim \mathit{fle}) \Rightarrow \\
\quad \mathit{Bind}\ (\mathit{StaticFlowT}\ \mathit{llr}\ \mathit{lle}\ \mathit{lm}) \\
\qquad (\mathit{StaticFlowT}\ \mathit{rlr}\ \mathit{rle}\ \mathit{rm}) \\
\qquad (\mathit{StaticFlowT}\ \mathit{flr}\ \mathit{fle}\ \mathit{fm})\ \textbf{where} \\
\quad (\mathit{StaticFlowT}\ \mathit{le}) \gg\!= f = \\
\quad\quad \mathit{StaticFlowT}\ \$\ \textbf{do}\ x \leftarrow \mathit{le}; \\
\quad\quad\quad\quad\quad\quad \mathit{runStaticFlowT}\ \$\ f\ x \\
\textbf{instance}\ (\mathit{Sequence}\ \mathit{lm}\ \mathit{rm}\ \mathit{fm}, \mathit{lle} \vee \mathit{rle} \sim \mathit{fle}) \Rightarrow \\
\quad \mathit{Sequence}\ (\mathit{StaticFlowT}\ \mathit{llr}\ \mathit{lle}\ \mathit{lm}) \\
\qquad\quad (\mathit{StaticFlowT}\ \mathit{rlr}\ \mathit{rle}\ \mathit{rm}) \\
\qquad\quad (\mathit{StaticFlowT}\ \mathit{rlr}\ \mathit{fle}\ \mathit{fm})\ \textbf{where} \\
\quad (\mathit{StaticFlowT}\ \mathit{le}) \gg (\mathit{StaticFlowT}\ \mathit{re}) = \\
\quad\quad \mathit{StaticFlowT}\ \$\ \mathit{le} \gg \mathit{re} \\
\textbf{instance}\ (\mathit{Return}\ m) \Rightarrow \\
\quad\quad \mathit{Return}\ (\mathit{StaticFlowT}\ \mathit{lr}\ \mathit{le}\ m)\ \textbf{where} \\
\quad \mathit{returnM} = \mathit{StaticFlowT} \circ \mathit{returnM}
\end{array}
$$

These are intimidating type signatures, but before looking at them, it is interesting to first look what the code actually does. Interestingly, all that it does is pass on the monadic operations to the underlying monad. This is clearly what we want, since this is a purely compile-time enforcement technique, not performing any work at run-time.

Looking at the type signatures, we see that for the monadic sequencing operation $\gg$, the result security level of the left hand side calculation is thrown away together with the result itself, and the right-hand side result security level is used as the result security level of the calculation. Slightly less evident, the effects of the combined calculation are considered at the meet (minimum) of the effect levels of the calculations being combined. This is natural, since an observer at that security level can observe at least one of the effects.

The monadic binding operation $\gg\!=$ has an even more complicated type. The result security level of the combined calculation here is the join (maximum) of the levels of both calculations' results, since both results can influence the combined result. The effect security level is, as before the meet (minimum) of both effect levels. In addition to this, the $\gg\!=$ type signature also performs a check to see if the bind is actually allowed. This is done using the constraint $\mathit{rle} \geqslant \mathit{llr}$, which makes sure that information at a certain security level can not be used to produce effects at a lower security level.

We don't repeat the *plugin1* and *plugin2* examples, since they look exactly the same as in section 5. We do need to recompile them, because of the new definitions of the API functions, and we need to compile each separately, because one is supposed to give a compilation error and the other one isn't.

$$
\begin{array}{l}
\mathit{runplugin1} :: \mathit{IO}\ () \\
\mathit{runplugin1} = \mathit{evalStateT}\ (\mathit{runStaticFlowT}\ \mathit{plugin1})\ \texttt{""}
\end{array}
$$

Running the *runplugin1* program in GHCi, the program is accepted. The compiler statically determines that the program respects the information flow policy. Its execution is unmodified and does not incur any run-time overhead.

```
*Main> runplugin1
: Safe info flowing through...
Downloading resource: http://example.com/res.txt
Sending mail: prefix; Safe info flowing through...
```

The *runplugin2* example now gives the following compile-time type error:

```
No instance for (GT L H)
  arising from a do statement at (...)
Possible fix: add an instance declaration for
  (GT L H)
In a stmt of a 'do' expression: y <- getA
In the second argument of '(>>)', namely
    'do { y <- getA;
          if y /= 0
          then putInt y else returnM () }'
(...)
```

We can still improve the error message a little, by playing a standard trick on the compiler.

$$
\begin{array}{l}
\textbf{data}\ \mathit{IllegalInformationFlow} \\
\textbf{class}\ \mathit{Failure}\ a \\
\textbf{instance}\ (\mathit{Failure}\ \mathit{IllegalInformationFlow}) \Rightarrow L \geqslant H
\end{array}
$$

The improved error message has the same error location info, but starts like this:

```
No instance for (Failure IllegalInformationFlow)
...
```

With this, our third and last information flow enforcement monad transformer is also ready and working correctly on at least the two examples we started with.

7. Proof

In this text, we do not provide a proof of the correctness of all for the three information flow enforcement techniques we implement. What we do provide, in this section, is a formal result about our implementation of the static enforcement technique in section 6. What we demonstrate is that a model language and type system similar to the one from Volpano et al.'s presentation [24] can be translated into our implementation and that under this translation, types on both ends correspond in a precise manner that we will define below. It is important to note that we do not prove the semantic correctness of our translation, but we consider that given a semantically correct set of base monad state and input/output functions, this correctness is at least credible given the relative simplicity of the translation.

Figure 8 shows the syntax of the simple model imperative programming language we will use. The language is the one we used previously in our paper about an information flow enforcement technique called secure multi-execution [4]. It is not the one used by Volpano et al., but it is similar except for the added basic input and output statements. Values denoted i and o represent identifiers of input and output channels, respectively.

Figure 9 shows an version of the classic type system, adapted to our situation. These typing rules assume a fixed, user-provided assignment of variables to security levels Γ, and fixed classifications σ_{in} and σ_{out} of input and output channels into security levels. We further assume an extension of the user-provided typing for variables to a typing for all expressions. An expression of type $e : l$ denotes an expression containing information at security level l. The covariant subtyping for expression types indicates that information at a certain security level can be used wherever more confidential information can. A statement typing $s : l\,\mathrm{cmd}$ denotes that the statement can produce effects visible on security level l. The contravariant subtyping for statement types indicates that statements with effects at a given level can be used wherever statements with lower level effects can.

Figure 11 now shows the translation function ϕ, translating a program $P : l\,\mathrm{cmd}$ in our model language to a calculation of type $\mathit{StaticFlowT}\ \ l\ \ L\ \ \mathit{BaseMonad}\ \ ()$ (see theorem 1 below). For a program P using a set of state variables

$$command ::= x := e \mid \; c; c \mid \textbf{ if } e \textbf{ then } c \textbf{ else } c \mid \textbf{ while } e \textbf{ do } c \mid \textbf{ skip } \mid \textbf{ input } x \textbf{ from } i \mid \textbf{ output } e \textbf{ to } o$$

Figure 8. Command syntax of a simple model imperative language. We assume a standard syntax for effect-free expressions e.

$$\frac{\Gamma(x) = l \quad \Gamma \vdash e : l}{\Gamma \vdash x := e : l\,\mathrm{cmd}} \tag{1}$$

$$\frac{\Gamma \vdash c : l\,\mathrm{cmd} \quad \Gamma \vdash c' : l\,\mathrm{cmd}}{\Gamma \vdash c; c' : l\,\mathrm{cmd}} \tag{2}$$

$$\frac{\Gamma \vdash e : l \quad \Gamma \vdash c : l\,\mathrm{cmd} \quad \Gamma \vdash c' : l\,\mathrm{cmd}}{\Gamma \vdash \textbf{if } e \textbf{ then } c \textbf{ else } c' : l\,\mathrm{cmd}} \tag{3}$$

$$\frac{\Gamma \vdash e : l \quad \Gamma \vdash c : l\,\mathrm{cmd}}{\Gamma \vdash \textbf{while } e \textbf{ do } c : l\,\mathrm{cmd}} \tag{4}$$

$$\frac{\Gamma(x) = l \quad \sigma_{\mathrm{in}}(i) = l}{\Gamma \vdash \textbf{input } x \textbf{ from } i : l\,\mathrm{cmd}} \tag{5}$$

$$\frac{\Gamma \vdash e : l \quad \sigma_{\mathrm{out}}(o) = l}{\Gamma \vdash \textbf{output } e \textbf{ to } o : l\,\mathrm{cmd}} \tag{6}$$

$$\Gamma \vdash \textbf{skip} : l \tag{7}$$

$$\frac{\Gamma \vdash e : l \quad l \leqslant l'}{\Gamma \vdash e : l'} \tag{8}$$

$$\frac{\Gamma \vdash c : l\,\mathrm{cmd} \quad l \geqslant l'}{\Gamma \vdash c : l'\,\mathrm{cmd}} \tag{9}$$

Figure 9. Adapted information flow enforcement typing rules, based on those developed by Volpano et al [24]. Typing rules for expressions are assumed, reflecting the maximum security level of any variable used in the expression. Also assumed is a security level typing Γ for variables, based on programmer-provided variable security level ascriptions.

$\{v_1, \ldots, v_n\}$, we assume that the variables are assigned security levels $\{l_{v_1}, \ldots, l_{v_n}\}$. We assume that properly typed Haskell values $getV_i :: StaticFlowT\ H\ l_{v_i}\ BaseMonad\ Int$, $putV_i :: Int \to StaticFlowT\ l_{v_i}\ L\ BaseMonad\ ()$ are defined. We assume that P uses input and output channels $\{i_1, \ldots, i_{n_i}\}$ and $\{o_1, \ldots, o_{n_o}\}$, classified at security levels $\{l_{i_1}, \ldots, l_{i_n}\}$ and $\{l_{o_1}, \ldots, l_{o_n}\}$ and that properly typed Haskell values $readI_j :: StaticFlowT\ H\ l_{i_j}\ BaseMonad\ Int$ and $writeO_i :: Int \to StaticFlowT\ l_{o_i}\ L\ BaseMonad\ ()$ are defined. Finally, the translation assumes a function $Vars$ returning the set of variables used in a given expression, and a translation function for expressions ϕ_{ex}, translating references to variables into references of correspondingly named Haskell values. With all of these, ϕ will translate the given program P in a straightforward manner. Maybe the only unintuitive feature of this translation is the use of helper function *upcastLeft* from Figure 10, which make sure the **if**-statement typechecks by upcasting the lowest typed of the two branch statements. **while**-statements are translated to a recursive monadic expression.

We now come to the central theorem of this section. In order to understand this correspondence, it is important to remark that our typing scheme assigns a single type to any monadic calculation, whereas Volpano et al. allow subtyping. For example, a calculation $StaticFlowT\ L\ l\ m\ ()$ corresponds to a statement s such that $s : l'\,\mathrm{cmd}$ for all $l' \leqslant l$. This is taken into account in the formulation of the next theorem.

$$\begin{aligned}
&upcastLeft :: StaticFlowT\ l\ L\ bm\ () \to \\
&\quad StaticFlowT\ l'\ L\ bm\ () \to \\
&\quad StaticFlowT\ (Min\ l\ l')\ L\ bm\ () \\
&upcastLeft\ a\ b = MkStaticFlowT\ (unMkStaticFlowT\ a)
\end{aligned}$$

Figure 10. Utility function *upcastLeft* for use in the translation function ϕ in Figure 11.

$$\frac{\begin{array}{c} c = \textbf{if } e \textbf{ then } c_{\mathrm{true}} \textbf{ else } c_{\mathrm{false}} \qquad Vars(e) = \{v_{i_1}, \ldots, v_{i_n}\} \\ cond = \phi_{ex}(e) \qquad m_{\mathrm{true}} = \phi(c_{\mathrm{true}}) \qquad m_{\mathrm{false}} = \phi(c_{\mathrm{false}}) \end{array}}{\begin{array}{rl} \phi(c) = & getV_{i_1} \gg\!= \lambda v_{i_1} \to \ldots \to getV_{i_n} \gg\!= \lambda v_{i_n} \to \\ & \textbf{if } cond \textbf{ then } upcastLeft\ m_{\mathrm{true}}\ m_{\mathrm{false}} \\ & \textbf{else } upcastLeft\ m_{\mathrm{false}}\ m_{\mathrm{true}} \end{array}} \tag{1}$$

$$\frac{m_1 = \phi(c_1) \qquad m_2 = \phi(c_2)}{\phi(c_1; c_2) = m_1 \gg m_2} \tag{2}$$

$$\frac{}{\phi(\textbf{skip}) = returnM\ () :: StaticFlowT\ H\ L\ BaseMonad\ ()} \tag{3}$$

$$\frac{\begin{array}{c} c = \textbf{while } e \textbf{ do } c_{\mathrm{loop}} \qquad Vars(e) = \{v_1, \ldots, v_n\} \\ cond = \phi_{ex}(e) \qquad m_{loop} = \phi(c_{\mathrm{loop}}) \end{array}}{\begin{array}{rl} \phi(c) = & fix\ \$\ \lambda self \to \\ & getV_{i_1} \gg\!= \lambda v_{i_1} \to \ldots \to getV_{i_n} \gg\!= \lambda v_{i_n} \to \\ & \textbf{if } cond \textbf{ then } m_{loop} \gg self \textbf{ else } returnM\ () \end{array}} \tag{4}$$

$$\frac{val = \phi_{ex}(e) \qquad Vars(e) = \{v_1, \ldots, v_n\}}{\begin{array}{rl} \phi(v_i := e) = & getV_{i_1} \gg\!= \lambda v_{i_1} \to \ldots \to \\ & getV_{i_n} \gg\!= \lambda v_{i_n} \to putV_i\ val \end{array}} \tag{5}$$

$$\frac{val = \phi_{ex}(e) \qquad Vars(e) = \{v_1, \ldots, v_n\}}{\begin{array}{rl} \phi(\textbf{output } e \textbf{ to } o_i) = & getV_{i_1} \gg\!= \lambda v_{i_1} \to \ldots \to \\ & getV_{i_n} \gg\!= \lambda v_{i_n} \to writeO_i\ val \end{array}} \tag{6}$$

$$\frac{c = \textbf{input } vi \textbf{ from } i_j}{\phi(c) = readI_j \gg\!= putV_i} \tag{7}$$

Figure 11. Translation function ϕ from our model language to calculations in the transformed monad.

Theorem 1. *Suppose a program P as before, using variables $\{v_1, \ldots, v_n\}$. Suppose $\{l_{v_1}, \ldots, l_{v_n}\}$, $getV_i$, $putV_i$, $\{i_1, \ldots, i_{n_i}\}$, $\{o_1, \ldots, o_{n_o}\}$, $\{l_{i_1}, \ldots, l_{i_n}\}$, $\{l_{o_1}, \ldots, l_{o_n}\}$, $readI_j$ and $writeO_i$, all as before.*

Then we have that $\phi(P)$::$StaticFlowT\ l'\ L\ BaseMonad\ ()$ if and only if $P : l\,\mathrm{cmd}$ for all $l' \geqslant l$.

Proof sketch. "only if": induction on the syntactic structure of P.

- Suppose $P = \textbf{if } e \textbf{ then } c_{\mathrm{true}} \textbf{ else } c_{\mathrm{false}}$, with $e : l$, $c_{\mathrm{true}}, c_{\mathrm{false}} : l\,\mathrm{cmd}$. By the induction hypothesis, we know that $\phi(c_{\mathrm{true}}) = m_{\mathrm{true}} :: StaticFlowT\ l'\ L\ BaseMonad\ ()$, $l' \geqslant l$ and $\phi(c_{\mathrm{false}}) = m_{\mathrm{false}} :: StaticFlowT\ l''\ L\ BaseMonad\ ()$, $l'' \geqslant l$. Case (1) in Figure 11, the typing of the $(\gg\!=)$ operator, the types of the $getV_i$, and the assumption on the typing of expressions (that it is determined by the security levels of the

variables used), together with Haskell typing rules, give us that $\phi(c)$::$StaticFlowT\ l'''\ L\ BaseMonad\ ()$ with $l''' \geqslant l$.

- Suppose $P = c_1; c_2$, with $c_1 : l\,\mathrm{cmd}$ and $c_2 : l\,\mathrm{cmd}$. We know that $m_1 = \phi(c_1)$::$StaticFlowT\ l'\ L\ BaseMonad\ ()$, $m_2 = \phi(c_2)$::$StaticFlowT\ l''\ L\ BaseMonad\ ()$, for some $l' \geqslant l$ and $l'' \geqslant l$. Case (2) in Figure 11, the type of the $(\gg)$ operator and the Haskell typing rules tell us that $\phi(P)$::$StaticFlowT\ l'''\ L\ BaseMonad\ ()$ with $l''' \geqslant l$.
- The cases for **skip**, **while**, **output**, **input** are similar or straightforward.

For the "if" direction, we also work by induction on the syntactic structure of P.

- Suppose now that $P =$ **if** e **then** c_{true} **else** c_{false}, that $\phi(P)$::$StaticFlowT\ l\ L\ BaseMonad\ ()$. Take l' and l'' such that $\phi(c_{\mathrm{true}})$::$StaticFlowT\ l'\ L\ BaseMonad\ ()$ and $\phi(c_{\mathrm{false}})$::$StaticFlowT\ l''\ L\ BaseMonad\ ()$. By induction, case (1), the type rules for **if**, we know l' is the same type as l'', that $c_{\mathrm{true}}, c_{\mathrm{false}} : l'''\,\mathrm{cmd}$ for all $(Min\ l'\ l'') \geqslant l'''$. Because the security level of e is determined by the variables used in it, because of the types of the $getV_i$ and the $(\gg\!=)$ operator, we get that $P : l''''\,\mathrm{cmd}$ for all $l \geqslant l''''$.
- Suppose $P = c_1; c_2$, with $m_1 = \phi(c_1)$, $m_2 = \phi(c_2)$. Suppose that $m_1 :: StaticFlowT\ l'\ L\ BaseMonad\ ()$ and $m_2 :: StaticFlowT\ l''\ L\ BaseMonad\ ()$. The type of the $(\gg)$ operator and Haskell typing rules then give us that $\phi(P)$::$StaticFlowT\ l\ L\ BaseMonad\ ()$ with $l \sim Min\ l'\ l''$. Induction gives us that $c_1 :: l'''\,\mathrm{cmd}$, $c_2 :: l'''\,\mathrm{cmd}$ for $l' \geqslant l'''$ and $l'' \geqslant l'''$ and the typing rules in our model language then ensure that $P :: l'''\,\mathrm{cmd}$ for all $l \geqslant l'''$
- **skip**, **while**, **output**, **input** are again similar or straightforward.

□

8. Discussion and Related Work

8.1 The techniques we use

Parameterised monads [1, 9] have already been shown to serve many purposes. Kiselyov uses parametrised monads for typing polymorphic delimited continuations [8]. Harrison and Hook also apply monad transformers for information flow security but in a different manner. Also, our use of parameterised monads and monad transformers in the context of policy enforcement on Haskell libraries is not new. Pucella describes an interesting implementation of session types using these two techniques [17].

8.2 The Monad operations as monitorable events in an information flow monitor

In a theoretical work about fundamental limits of run-time policy enforcement mechanisms, Ligatti et al. [14] discuss how the set of events in the executing programs that can be observed by the monitor is one of the factors that determines the power of the enforcement mechanisms. In this respect, our run-time enforcement monad transformer from section 3 can be regarded as a monitor receiving only applications of the monadic bind $(\gg\!=)$ and sequence $(\gg)$ operations as monitorable events. We think we have shown in this text that monitoring this set of events provides us with enough information about information flow in a program to be able to enforce relevant forms of secure information flow, and that we have been able to encode three important enforcement techniques from the literature. Nevertheless, we also want to point out that for at least one other enforcement technique, we do not believe such an implementation is possible.

In section 5, we have explained the general problem related to flow-sensitive purely dynamic information flow monitors and Le Guernic's solution to this problem. Austin and Flanagan discuss an alternative [2], purely dynamic solution to the problem based on the "No Sensitive Upgrade" rule. Under this rule, it is not allowed for a variable's security level to be upgraded to a higher level because of an assignment with the program counter at a higher level. Instead, such an assignment will cause the program to be terminated. Our example *leak* on page 5 section 5 would thus be terminated when the assignment $put\ p$ is executed. Austin and Flanagan prove that this new rule fixes the flow-sensitivity problem and ensures secure information flow.

The problem for implementing this technique using our approach can be seen by looking at how the technique handles the following two statements: **if** sec **then** $x := \mathrm{true}$ **else skip** and $x := sec$. Assuming that the variable x was previously assigned a low security level, the no sensitive upgrade rule would terminate the first command. The second command on the other hand is legal, and the monitor would just assign a high security level to the variable x. The problem now becomes apparent if we notice that in a monadic encoding of these examples, we get $getSec \gg\!= \lambda sec \rightarrow$ **if** sec **then** $putX\ True$ **else** $returnM\ ()$ and $getSec \gg\!= putX$ respectively. Notice that both cases reduce to an application of the monadic bind operator, making it fundamentally impossible for monad transformer policy enforcers to make the required distinction. Accordingly, we believe Austin and Flanagan's enforcement technique cannot be implemented in our model.

8.3 The information flow enforcement techniques we implement

The information flow enforcement techniques we implement are classic from the literature. For an overview of existing static techniques, we refer to Sabelfeld and Myers [20] and for dynamic techniques to Le Guernic's PhD thesis [11].

The techniques we implement in this paper are those described by Sabelfeld and Russo [21] (section 3), Le Guernic et al. [12] (section 5) and Volpano et al. [24] (section 6). These techniques are tied in an important way to their handling of imperative features like mutable state and side effects, which is why they are relevant to our policy enforcement over monadic APIs. A small but nevertheless relevant contribution in this paper is the way we have adapted these techniques to handle monadic operations. For all three of these techniques, the extensions followed straightforwardly from considering the monadic binding operator as a generalised branching construct and generalising the rule for the **if** statement. Our presentation of Volpano et al.'s type system unified the linear type hierarchies for expressions and statements into a type lattice for monadic calculations, like in Crary et al.'s proposal [3].

An interesting direction for future research comes from the observation that monads in functional languages can be used for more purposes than the encoding of mutable state variables and side effects. Other features like exceptions and even delimited continuations can also be encoded in it. It would be interesting to investigate the application of techniques from the information flow literature for handling these features [15, 16, 23] by lifting the relevant base monad operations into information flow enforcement monad transformers.

8.4 Other information flow enforcement libraries

Finally, as an example of a Haskell information flow library, it is fundamental to compare our work to other interesting papers in this domain. Li and Zdancewic [13] were the first to propose an information flow enforcement library as a light-weight alternative to performing the enforcement as part of the implementation of a language compiler or interpreter. Li and Zdancewic require the programmer to explicitise control flow by encoding the program using the do-notation for arrows [7]. This is similar in spirit to

how we approximate information flow using the invocations of monad operations in untrusted code. They perform a static analysis of the resulting dependency graph to detect illegal information flow, but they perform this analysis at runtime. This strikes us as a compromise lacking the advantages of both dynamic checking (e.g. determination of security levels during execution, ignoring dead code) and static checking (no run-time performance impact, errors signaled to developer during compilation). A strong point of Li and Zdancewic' approach is that they provide a full formal proof of the soundness of their enforcement. Unfortunately, they do not handle side effects or imperative features like mutable state, but mention it as interesting future work. Their use of the arrows framework requires the API user to work with an unfamiliar abstraction, and yields code that is quite strongly adapted for and as such coupled to their enforcement mechanism.

Tsai, Russo and Hughes [22] improve upon Li and Zdancewic' work by adding support for typing separate components of structured data differently, mutable references and concurrency primitives to the library, extending and modifying Li and Zdancewic' library significantly. These extensions improve the applicability of the library quite a bit, but they still require API users to write their code in the arrow abstraction. Interestingly, their $FlowArrow$ is parameterised by an "underlying arrow" in a similar way as our monad transformers, but afterwards, they remove this generality by instantiating the underlying arrow with the $a \rightarrow IO\ b$ arrow when adding support for references. It would have been more elegant to use some sort of arrow transformer based on something like the ST monad [10]. Unlike Li and Zdancewic, Tsai, Russo and Hughes require a type level representation of security levels (like we do in section 6) to keep track of the security type of the contents of references, and convert back to the dynamic representation a type class when needed in the run-time information flow analysis. This idea is interesting, and could perhaps be used to add support for references in our libraries as well.

Russo, Claessen and Hughes present an information flow enforcement library based on the more widespread monad abstraction instead of arrows [19]. Their approach is simple and corresponds quite strongly to the static enforcement technique we present in section 6. They use the $Sec\ s$ type constructor to protect values at the security level represented by s, and use the $SecIO\ s'$ to represent IO calculations producing side effects visible at security level s'. Their $SecIO\ s\ (Sec\ s'\ v)$ corresponds quite strongly to the type $StaticFlowT\ s'\ s\ IO\ v$ in our implementation.

Like Tsai, Russo and Hughes, they miss the opportunity to parametrise the underlying monad, instead mandating the common IO monad. Their $Sec\ s$ and $SecIO\ s$ type constructors are instances of the $Monad$ type class, but Russo, Claessen and Hughes do not use the parameterised monads technique. Because of this, they require API users to use explicit security level casting functions ($plug$, up) in their programs. This is acceptable, but does not allow the strong separation of information flow enforcement from the underlying API and programs using it that parameterised monads allow us to achieve, thus limiting the modularity of their library. In addition, their use of monadic values ($Sec\ s\ a$) as the values of another monad ($SecIO\ s'$) requires API users to work with monads on two levels, which again leads to unnatural code.

In their text, Russo, Claessen and Hughes subsequently demonstrate an impressive declassification framework using *declassification combinators* and the concept of *escape hatches*. Declassification is something which we have not looked at in this paper, but we think their declassification framework can in fact be translated easily to our implementations. In fact, most of their declassification functions can probably be reused as is by simply lifting them correctly into our transformed monads. This would add important value to our implementations, without much effort.

We think our library improves upon this existing work in a number of respects. First, as we have discussed, our implementations are much more modular, and allow for a much stronger separation between information flow enforcement on the one hand and the underlying API and programs using it on the other. Second, this previous work never featured generality in the specific information flow enforcement technique used. This text, on the other hand, presents implementations of three different classical enforcement techniques. Third, our parameterisation in the base monad is more general and elegant than other libraries' use of the IO monad. Finally, we think that our libraries keep the API presented very natural to the API user, which previous approaches did not achieve, either due to their use of the uncommon arrow abstraction or because of the requirement for working in more than a single monad and using explicit casting functions.

References

[1] R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335–376, 2009.

[2] T. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS Workshop*, pages 113–124. ACM, 2009.

[3] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of functional programming*, 15(02):249–291, 2005.

[4] D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, 2010.

[5] I. Dupree. Ghc ticket 1537 - do notation translation. `http://hackage.haskell.org/trac/ghc/ticket/1537`, 2007. Fixed in GHC 6.8.3.

[6] GHC Team. GHC manual - rebindable syntax and the implicit Prelude import. `http://haskell.org/ghc/docs/6.12.2/html/users_guide/syntax-extns.html#rebindable-syntax`, 2010.

[7] J. Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1-3):67–111, 2000.

[8] O. Kiselyov. Genuine shift/reset in haskell. Mailing list message, December 2007.

[9] E. Kmett. Parameterized monads in haskell. *The Comonad.Reader*, 2007. URL `http://comonad.com/reader/2007/parameterized-monads-in-haskell/`.

[10] J. Launchbury, P. Jones, and L. Simon. Lazy functional state threads. In *PLDI*, page 35. ACM, 1994.

[11] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[12] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *ASIAN*, pages 75 – 89, 2006.

[13] P. Li and S. Zdancewic. Encoding information flow in haskell. In *19th IEEE Computer Security Foundations Workshop, 2006*, page 12, 2006.

[14] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1):2–16, 2005.

[15] A. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241. ACM, 1999.

[16] F. Pottier and V. Simonet. Information flow inference for ML. *TOPLAS*, 25(1):117–158, 2003.

[17] R. Pucella and J. Tov. Haskell session types with (almost) no class. *ACM SIGPLAN Notices*, 44(2):25–36, 2009.

[18] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Computer Security Foundations*, 2010.

[19] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. In *Haskell Symposium*, pages 13–24. ACM, 2008.

[20] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[21] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Perspectives of System Informatics*, 2009.

[22] T.-C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *CSF*, pages 187–202, 2007.

[23] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *CSFW*, pages 156–168, 1997.

[24] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2/3):167–188, 1996.