A Library for Light-Weight Information-Flow Security in Haskell

Alejandro Russo Koen Claessen John Hughes
Chalmers University of Technology, Gothenburg, Sweden
{russo,koen,rjmh}@chalmers.se

Abstract

Protecting confidentiality of data has become increasingly important for computing systems. *Information-flow* techniques have been developed over the years to achieve that purpose, leading to special-purpose languages that guarantee information-flow security in programs. However, rather than producing a new language from scratch, information-flow security can also be provided as a library. This has been done previously in Haskell using the arrow framework. In this paper, we show that arrows are not necessary to design such libraries and that a less general notion, namely monads, is sufficient to achieve the same goals. We present a *monadic library* to provide information-flow security for Haskell programs. The library introduces mechanisms to protect confidentiality of data for pure computations, that we then easily, and modularly, extend to include dealing with side-effects. We also present combinators to dynamically enforce different declassification policies when release of information is required in a controlled manner. It is possible to enforce policies related to *what*, by *whom*, and *when* information is released or a combination of them. The well-known concept of monads together with the light-weight characteristic of our approach makes the library suitable to build applications where confidentiality of data is an issue.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages

General Terms Security, Languages

Keywords Information-flow, Declassification, Library, Monad

1. Introduction

Protecting confidentiality of data has become increasingly important for computing systems. Often, software is so complex that it is hard to see if a program can be abused by a malicious person to gain access to private data. This is important when developing software oneself, and becomes increasingly more important if one is forced to trust other people's code.

Information-flow techniques have been developed over the years to achieve this kind of protection. For example, as a result, two main stream compilers, Jif (based on Java) and Flowcaml (based on Ocaml) have been developed to guarantee information-flow security in programs.

However, it is a very heavy-weight solution to introduce a new programming language for dealing with information-flow. In this work, we explore the possibility of expressing restrictions on information-flow as a library rather than a new language.

We end up with a light-weight monadic approach to the problem of expressing and ensuring information-flow in Haskell. Code that exhibits information flows that are disallowed will be ill-typed and rejected by the type checker. Our approach is general enough to deal with practical concepts such as secure reading and writing to files (which can be generalized to capture any information exchange with the outside world) and declassification (a pragmatic way of allowing controlled information leakage (Sabelfeld and Sands 2005)).

Our library might be used in scenarios where we want to incorporate in our programs some code written by outsiders (untrusted programmers) to access our private information. Such code can be also allowed to interact with the outside world (for example by accessing the web). We would like to have a guarantee that the program will not send our private data to an attacker. A slightly different, but related, scenario is where we ourselves write the possibly unsafe code, but we want to have the help of the type checker to find possible security mistakes.

Li and Zdancewic (Li and Zdancewic 2006) have previously shown how to provide information-flow security also as a library, but their implementation is based on *arrows* (Hughes 2000), which naturally requires programmers to be familiar with arrows when writing security-related code. In this work, we show that arrows are not necessary to design such libraries and that a less general notion, namely monads, is sufficient to achieve very similar goals.

1.1 Motivating example

Consider a machine running Linux with the default installation of the Shadow Suite (Jackson 1996) responsible to store and manage users' passwords. In this machine, file `/etc/passwd` contains information regarding users such as user and group ID's, which are used by many system programs. This file must remain world readable. Otherwise, simple commands as `ls -l` stop working. Passwords are set in the file `/etc/shadow`, which can only be read and written by root. From now on, we refer to the passwords stored in this file as *shadow passwords*. Programs that verify passwords need to be run as root. From the security point of view, this requirement implies that very careful programming practices must be followed when creating such programs. For instance, if a program running as root has a shell escape, it is not desirable that such shell escape runs with root privileges. The process to verify a password usually consists of taking the input provided by the user, applying some cryptographic algorithms to it, and comparing the result of that with the user's information stored in `/etc/shadow`. Observe that an attacker can encrypt a dictionary of common passwords offline and then, given some file `/etc/shadow`, try to guess users' passwords by checking matches. This attack is known as an *offline dictionary attack* and is one of the most common methods for gain-

Haskell'08, September 25, 2008, Victoria, BC, Canada.

ing or expanding unauthorized access to systems (Narayanan and Shmatikov 2005). In order to obtain the content of `/etc/shadow`, the attacker needs to obtain root privileges, which is not impossible to achieve (Local Root Exploit 2008). Given these facts, we can conclude that there are mainly two security problems with shadow passwords: programs require having root privileges to verify passwords and offline dictionary attacks. We start dealing with these problems by firstly limiting the access to the password file. With this in mind, we assume that information stored in `/etc/shadow` is only accessible through an API. The following Haskell code shows an example of such API.

```
data Spwd = Spwd { uid :: UID, cypher :: Cypher }

getSpwdName :: Name -> IO (Maybe Spwd)
putSpwd     :: Spwd -> IO ()
```

Data type `Spwd` stores users' identification number (`uid::UID`) and users' password (cypher :: Cypher). For a simple presentation, we assume that passwords are stored as plain text and not cyphers. Function `getSpwdName` receives a user name and returns his (her) password if such user exists. Function `putSpwd` takes a register of type `Spwd` and adds it to the shadow password file. This API is now the only way to have access to shadow passwords. We can still be more restrictive and require that such API is only called under root privileges, which is usually the case for Unix-like systems. Unfortunately, this restriction does not help much since attackers could obtain unauthorized root access and then steal the passwords. However, by applying information-flow techniques to the API and programs that use it, it is possible to guarantee that passwords are not revealed while making possible to verify them. In other words, offline dictionary attacks are avoided as well as some requirements as having root privileges to verify passwords. In Section 3.3, we show a secure version of this API.

1.2 Contributions

We present a *light-weight library* for *information-flow security* in Haskell. The library is *monadic*, which we argue is easier to use than arrows, which were used in previous attempts. The library has a pure part, but also deals with *side-effects*, such as the secure reading and writing of files. The library also provides novel and powerful means to specify *declassification policies*.

1.3 Assumptions

In the rest of the paper, we assume that the programming language we work with is a controlled version of Haskell, where code is divided up into *trusted* code, written by someone we trust, and *untrusted* code, written by the attacker. There are no restrictions on the trusted code. However, the untrusted code has certain restrictions; certain modules are not available to the untrusted programmer. For example, all modules providing IO functions, including exceptions (and of course `unsafePerformIO`) are not allowed. Our library will reintroduce part of that functionality to the untrusted programmer in a controlled, and therefore, secure way.

2. Non-interference for pure computations

Non-interference is a well-known *security policy* that preserves confidentiality of data (Cohen 1978; Goguen and Meseguer 1982). It states that public outcomes of programs do not depend on their confidential inputs.

In imperative languages, information leaks arise from the presence of *explicit* and *implicit* flows inside of programs (Denning and Denning 1977). Explicit flows are produced when secret data is placed explicitly into public locations by an assignment. Implicit flows, on the other hand, use control constructs in the language in order to reveal information. In a pure functional language, however, this distinction becomes less meaningful, since there are no assignments nor control constructs. For example, a conditional (if-then-else) is just a function as any other function in the language. In a pure language, all information-flow is explicit; information only flows from function arguments to function results.

```
newtype Sec s a

instance Functor (Sec s)
instance Monad (Sec s)

sec  :: a -> Sec s a
open :: Sec s a -> s -> a
```

Figure 1. The Sec monad

To illustrate information leaks in pure languages, we proceed assuming that a programmer, potentially malicious, needs to write a function `f :: (Char,Int) -> (Char,Int)` where characters and integers are considered respectively secret and public data. We assume that attackers can only control public inputs and observe public results when running programs, and can thus only observe the second component of the pair returned by function `f`. For simplicity, we also assume that type `Char` represents ASCII characters.

If a programmer writes the code

```
f (c, i) = (chr (ord c + i), i+3)
```

then the function is non-interferent and preserves the confidentiality of `c`; the public output of `f` is independent of the value of `c` [1]. If a programmer instead writes

```
f (c, i) = (c, ord c)
```

then information about `c` is revealed, and the program is not non-interferent! Attackers might try to write less noticeable information leaks however. For instance, the code

```
f (c, i) = (c, if ord c > 31 then 0 else 1)
```

leaks information about the printability of the character `c` and therefore should be disallowed as well.

In this section, we show how monads can be used to avoid leaks and enforce the non-interference property for pure computations.

2.1 The Sec monad

In order to make security information-flow specific, we are going to make a distinction at the type level between *protected* data and *public* data. Protected data only lives inside a special *monad* (Wadler 1992). This security monad makes sure that only the parts of the code that have the right to do so are able to look at protected data.

In larger programs, it becomes necessary to talk about several *security levels* or *areas*. In this case, values are not merely protected or public, but they can be protected by a certain security level `s`.

Take a look at Fig. 1, which shows the API of an abstract type, `Sec`, which is a functor and a monad. There are two functions provided on the type `Sec`; `sec` is used to protect a value, and `open` is used to look at a protected value. However, to look at a protected value of type `Sec s a`, one needs to have a value of type `s`. Restricting access to values of different such types `s` by means of the module system allows fine control over which parts of the program can look at what data. (For this to work, `open` needs to be strict in its second argument.)

For example, if we define a security area `H` in the following way:

[1] Function `chr` returns an exception when the received argument does not represent an ASCII code. By observing occurrences of exceptions or computations that diverge, an attacker can deduce some information about secrets. However, we only consider programs that terminate successfully.

```
module Lattice where

data L = L
data H = H

class Less sl sh where
  less :: sl -> sh -> ()

instance Less L L where
  less _ _ = ()

instance Less L H where
  less _ _ = ()

instance Less H H where
  less _ _ = ()
```

Figure 2. Implementation of a two-point lattice

```
data H = H
```

then we can model the type of the function `f` given in the beginning of this section as follows:

```
f :: (Sec H Char, Int) -> (Sec H Char, Int)
```

The first, secure, example of `f` can be programmed as follows:

```
f (sc,i) = ((\c -> chr (ord c + i)) `fmap` sc,i+3)
```

However, the other two definitions can not be programmed without making use of `H` or breaking the type checker.

So, for a part of the program that has no means to create non-bottom values of a type `s`, direct access to protected values of type `Sec s a` is impossible. However, computations involving protected data are possible as long as the data stays protected. This can be formalized by stating that type `Sec` guarantees a *non-interference* property. For any type `A`, and values `a1, a2 :: A`, a function

```
f :: Sec H A -> Bool
```

will produce the same result for arguments `a1` and `a2`. See (Russo et al. 2008a) for more details.

We will later show the implementation of the type `Sec` and its associated functions.

2.2 Security lattice

Valid information flows inside of programs are determined by a *lattice on security levels* (Denning 1976). Security levels are associated to data in order to establish its degree of confidentiality. The ordering relation in the lattice, written $\sqsubseteq$, represents allowed flows. For instance, $l_1 \sqsubseteq l_2$ indicates that information at security level l_1 can flow into entities of security level l_2.

For simplicity, in this paper, we will only use a two-point lattice with security levels `H` and `L` where `L` $\sqsubseteq$ `H` and `H` $\not\sqsubseteq$ `L`. Security levels `H` and `L` denote secret (*high*) and public (*low*) information, respectively. The implementation of the lattice is shown in Figure 2. Type class `Less` encodes the relation $\sqsubseteq$ and security levels are represented as singleton types (Pierce 2004). The role of `less` is explained in Section 4. Public information is characterized by the security level `L`. Constructor `L` is then publicly available so that data at security level `L` can be observed by anyone, which also includes attackers.

As explained earlier, attackers must have no access to the constructor `H`. In Section 4, we describe how to achieve such restriction.

Finally, to capture the fact that valid information flows occur from *lower* (`L`) to higher (`H`) security levels, we introduce the function

```
up :: Less sl sh => Sec sl a -> Sec sh a
```

The function `up` can be used to turn any protected value into a protected value at a higher security level. The implementation of `up` will be shown later.

3. Non-interference and side-effects

The techniques described in Section 2 do not perform computations with side-effects. The reason for that is that side-effects involving confidential data cannot be executed when they are created inside of the monad `Sec s`.

Even if we allowed a restricted and secure form of file reading and writing in the IO-monad, that would still not be enough. For example, if we, read information from file A, and depending on the value of a secret, want to write either to a file B or file C, we would obtain a computation of type `IO (Sec H (IO ()))`. It is easy to see that these types quickly become unmanagable, and, more importantly, unusable.

In this section, we show how we can augment our security API to be able to deal with *controlled* side-effects while still maintaining non-interference properties.

In this paper, we concentrate how to provide an API that allows reading and writing protected data from and to files. For this to work properly, files need to contain a security level, so that only data from the right security level can be written to a file. We assume that the attacker has no way of observing what side-effects were performed, other than through our API. (The attacker, so to say, sits within the Haskell program and has no way of getting out[2].)

The ideas for reading and writing files can be extended to deal with many other controlled IO operations, such as creating, reading and writing secure references, communicating over secure channels, etc. We will however not deal with the details of such operations in this paper.

3.1 Secure files

We model all interactions with the outside world by operations for reading and writing files (Tanenbaum 2001). For that reason, we decide to include secure file operations in our library. We start by assigning security levels to files in order to indicate the confidentiality of their contents. More precisely, we introduce the abstract data type `File s`. Values of type `File s` represent names of files whose contents have security level `s`. These files are provided by the trusted programmer. We assume that attackers have no access to the internal representation of `File s`. In Section 4, we show how to guarantee such assumption.

A first try for providing secure file operations is to provide the following two functions:

```
readSecIO  :: File s -> IO (Sec s String)
writeSecIO :: File s -> Sec s String -> IO ()
```

These functions do not destroy non-interference, because they do not open up for extra information-flow between security levels. The data read from a file with security level `s` is itself protected with security level `s`, and any data of security level `s` can be written to a file of security level `s`.

However, the above functions are not enough to preserve confidentiality of data. Take a look at the following program:

```
writeToAFile :: Sec H String -> Sec H (IO ())
writeToAFile secs =
  (\s -> if length s < 10
           then writeSecIO file1 s
           else writeSecIO file2 s) `fmap` secs
```

[2] A situation where the attacker is in league with a hacker who has gotten access to our system, and can for example read log files, is beyond our control and the guarantees of our library.

```
newtype SecIO s a

instance Functor (SecIO s)
instance Monad (SecIO s)

value :: Sec s a -> SecIO s a

readSecIO  :: File s' -> SecIO s (Sec s' String)
writeSecIO :: File s -> String -> SecIO s ()

plug :: Less sl sh => SecIO sh a -> SecIO sl (Sec sh a)
run  :: SecIO s a -> IO (Sec s a)
```

Figure 3. The SecIO monad

Here, `file1, file2 :: File H` is assumed to be defined elsewhere.

The behavior of the above function is indeed dependent on the protected data in its argument, as indicated by the result type. However, only the *side-effects* of the computation are dependent on the data, not the *result value*. Why is this important? Because we assume that the attacker has no way of observing from within the program what these side-effects are! (Unless the attacker can observe the results of the side-effects, namely the change of file contents in either `file1` or `file2`, but that information can only be obtained by someone with the appropriate security clearance anyway.) This assumption is valid for the scenarios described in Section 1.

In other words, since side-effects cannot be observed from within a program, we are going to allow the leakage of side-effects. Our assumption is only true if we restrict the IO actions that the attacker can perform.

3.2 The SecIO monad

To this end, we introduce a new monad, called `SecIO`. This monad is a variant of the regular IO monad that keeps track of the security level of all data that was used inside it.

Take a look at Fig. 3, which shows the API for an abstract type `SecIO`, which is a functor and a monad. Values of type `SecIO s a` represent computations that can securely read from any file, securely write to files of security level `s` (or higher), and look at data protected at level `s` (or lower).

The function `value` can be used to look at a protected value at the current security level. The function `readSecIO` reads protected data from files at any security level, protecting the result as such. The function `writeSecIO` writes data to files of the current security level.

The function `plug` is used to import computations with side-effects at a high level into computations with side-effects at a low level of security. Observe that only the side-effects are "leaked", not the result, which is still appropriately protected by the high security level. This function is particularly suitable to write programs that contain loops that depend on public information and perform, based on secret and public data, side-effects on secret files in each iteration.

These functions together with the return and bind operations for `SecIO s` constitute the basic interface for programmers.

Based on that, more convenient and handy functions can then be defined. For instance,

```
s_read :: Less s' s => File s' -> SecIO s String
s_read file =  do ss <- readSecIO file
                  value (up ss)

s_write :: Less s' s =>
           File s -> String -> SecIO s' (Sec s ())
s_write file str = plug (writeSecIO file str)
```

Observe that `s_read` and `s_write` have simpler types while practically providing the same functionality as `readSecIO` and `writeSecIO`, respectively.

In the next section, we show how to implement the core part of our library: the monads `Sec s` and `SecIO s`. We continue this section with an example that shows how these APIs can be used.

3.3 Developing a secure shadow passwords API

As an example of how to apply information-flow mechanisms, we describe how to adapt the API described in the introduction to guarantee that neither API's callers or the API itself reveal shadow passwords. Specifically, passwords cannot be copied into public files at all. Hence, offline dictionary attacks are avoided as well as the requirement of having root privileges to verify passwords. As mentioned in the introduction, we assume that the contents of `/etc/shadow` is only accessible through the API. For simplicity, we assume that this file is stored in the local file system, which naturally breaks the assumption we have just mentioned (user root has access to all the files in the system). However, it is not difficult to imagine an API that establishes, for example, a connection to some sort of password server in order to get information regarding shadow passwords.

We firstly start adapting our library to include the two-point lattice mentioned in Section 2. We decide to associate security level `H`, which represents secret information, to data regarding shadow passwords. Then, we indicate that file `/etc/shadow` stores secret data by writing the following lines

```
shadowPwds :: File H
shadowPwds  = MkFile "/etc/shadow"
```

We proceed to modify the API to indicate what is the secret data handled by it. More precisely, we redefine the API as follows:

```
getSpwdName :: Name -> IO (Maybe (Sec H Spwd))
putSpwd     :: Sec H Spwd -> IO ()
```

where values of type `Spwd` are now "marked" as secrets [3]. The API's functions are then adapted, without too much effort, to meet their new types. In order to manipulate data inside of the monad `Sec H`, API's callers need to import the library in their code. Since `/etc/shadow` is the only file with type `File H` in our implementation, this is the only place where secrets can be stored after executing calls to the API. By marking values of type `Spwd` as secrets, we restrict how information flows inside of the API and API's callers while making possible to operate with them. In Section 5, we show how to implement a login program using the adapted API.

4. Implementation of monads Sec and SecIO

In this section, we provide a possible implementation of the APIs presented in the previous two sections.

In Fig. 4 we show a possible implementation of `Sec`. `Sec` is implemented as an identity monad, allowing access to its implementation through various functions in the obvious way. The presence of `less` in the definition of function `up` includes `Less` in its typing constrains. Function `unSecType` is used for typing purposes and has no computational meaning. Note the addition of the function `reveal`, which can reveal any protected value. This function is not going to be available to the untrusted code, but the trusted code might sometimes need it. In particular, the implementation of `SecIO` needs it in order to allow the leakage of side-effects.

In Fig. 5 we show a possible implementation of `SecIO`. It is implemented as an IO computation that produces a safe result. As

[3] Values of type `Maybe` are not included inside of `Sec H` since the existence of passwords is linked to the existence of users in the system, which is considered public information.

```
module Sec where

-- Sec
newtype Sec s a = MkSec a

instance Monad (Sec s) where
  return x = sec x

  MkSec a >>= MkSec k =
    MkSec (let MkSec b = k a in b)

sec :: a -> Sec s a
sec x = MkSec x

open :: Sec s a -> s -> a
open (MkSec a) s = s `seq` a

up :: Less s s' => Sec s a -> Sec s' a
up sec_s@(MkSec a) = less s s' `seq` sec_s'
                     where (sec_s') = MkSec a
                           s        = unSecType sec_s
                           s'       = unSecType sec_s'

-- For type-checking purposes (not exported).
unSecType :: Sec s a -> s
unSecType _ = undefined

-- only for trusted code!
reveal :: Sec s a -> a
reveal (MkSec a) = a
```

Figure 4. Implementation of Sec monad

an invariant, the IO part of a value of type `SecIO s a` should only contain unobservable (by the attacker) side-effects, such as the reading from and writing to files.

There are a few things to note about the implementation. Firstly, the function `reveal` is used in the implementation of monadic bind, in order to leak the *side-effects* from the protected IO computation. Remember that we assume that the performance of side-effects (reading and writing files) cannot be observed by the attacker. Some leakage of side-effects is unavoidable in any implementation of the functionality of `SecIO`. Secondly, the definition of the type `File` does not make use of its argument `s`. This is also unavoidable, because it is only by a promise from the trusted programmer that certain files belong to certain security levels. Thirdly, function `plug`, similarly to function `up`, includes `less` and an auxiliary function (`unSecIOType`) to properly generate type constraints.

The modules `Sec`, `SecIO`, and `Lattice` can only be used by trusted programmers. The untrusted programmers only get access to modules `SecLibTypes` and `SecLib`, shown in Fig. 6. They import the three previous modules, but only export the trusted functions. Observe that the type `L` and its constructor `L` are exported, but for `H`, only the type is exported and not its constructor. Method `less` is also not exported. Therefore, functions `up` and `plug` are only called with the instances of `Less` defined in `Lattice.hs`.

In order to check that a module is safe with respect to information-flow, the only thing we have to check is that it does not import trusted modules, in particular:

- `Sec` and `SecIO`
- any module providing exception handling, for example `Control.Monad.Exception`,
- any module providing unsafe extensions, for example `System.IO.Unsafe`

```
module SecIO where
import Lattice
import Sec

-- SecIO
newtype SecIO s a = MkSecIO (IO (Sec s a))

instance Monad (SecIO s) where
  return x = MkSecIO (return (return x))

  MkSecIO m >>= k =
    MkSecIO (do sa <- m
                let MkSecIO m' = k (reveal sa)
                m')

-- SecIO functions
value :: Sec s a -> SecIO s a
value sa = MkSecIO (return sa)

run :: SecIO s a -> IO (Sec s a)
run (MkSecIO m) = m

plug :: Less sl sh => SecIO sh a -> SecIO sl (Sec sh a)
plug ss_sh@(MkSecIO m)
     = less sl sh `seq` ss_sl
       where
         (ss_sl) = MkSecIO (do sha <- m
                               return (sec sha))
         sl      = unSecIOType ss_sl
         sh      = unSecIOType ss_sh

-- For type-checking purposes (not exported).
unSecIOType :: SecIO s a -> s
unSecIOType _ = undefined

-- File IO
data File s = MkFile FilePath

readSecIO :: File s' -> SecIO s (Sec s' String)
readSecIO (MkFile file) =
  MkSecIO ((sec . sec) `fmap` readFile file)

writeSecIO :: File s' -> String -> SecIO s ()
writeSecIO (MkFile file) s =
  MkSecIO (sec `fmap` writeFile file s)
```

Figure 5. Implementation of SecIO monad

5. Declassification

Non-interference is a security policy that specifies the absence of information flows from secret to public data. However, real-word applications release some information as part of their intended behavior. Non-interference does not provide means to distinguish between intended releases of information and those ones produced by malicious code, programming errors, or vulnerability attacks. Consequently, it is needed to relax the notion of non-interference to consider *declassification* policies or intended ways to leak information. In this section, we introduce run-time mechanisms to enforce some declassification policies found in the literature.

Declassification policies have been recently classified in different dimensions(Sabelfeld and Sands 2005). Each dimension represents aspects of declassification. Aspects correspond to *what*, *when*, *where*, and by *whom* data is released. In general, type-systems to enforce different declassification policies include different features, e.g rewriting rules, type and effects, and external analysis (Myers and Liskov 2000; Sabelfeld and Myers 2004; Chong and Myers 2004). Encoding these features directly into the Haskell type system would considerably increase the complexity of our library. For

```
module SecLibTypes ( L (..), H,  Less () ) where
import Lattice


module SecLib
 (   Sec, open, sec, up
   , SecIO, value, plug, run,
   , File, readSecIO, writeSecIO, s_read, s_write
 )
where

import Sec
import SecIO
```

Figure 6. Modules to be imported by untrusted code

the sake of simplicity and modularity, we preserve the part of the library that guarantees non-interference while orthogonally introducing run-time mechanisms for declassification. More precisely, declassification policies are encoded as programs which perform run-time checks at the moment of downgrading information. In this way, declassification policies can be as flexible and general as programs! Additionally, we provide functions that automatically generate declassification policies based on some criteria. We call such programs *declassification combinators*. We provide combinators for the dimensions *what*, *when*, and *who* (*where* can be thought as a particular case of *when*). As a result, programmers can combine dimensions by combining applications of these combinators.

5.1 Escape Hatches

In our library, declassification is performed through some special functions. By borrowing terminology introduced in (Sabelfeld and Myers 2004), we call these functions "escape hatches" and we represent them as follows.

```
type Hatch s s' a b = Sec s a -> IO (Maybe (Sec s' b))
```

Escape hatches are functions that take some data at security level `s`, perform some computations with it, and then probably return a result depending if downgrading of information to security level `s'` is allowed or not. Arbitrary escape hatches can be included in the library depending on the declassification policies needed for the built applications. In fact, escape hatches are just functions. Types `IO` and `Maybe` are present in the definition of `Hatch s s' a b` in order to represent run-time checks and the fact that declassification may not be possible on some circumstances. By placing `Maybe` outside of monad `Sec s'`, the fact that declassification is possible or not is public information and programs can thus take different actions in each case. Consequently, it is important to remark that declassification policies should not depend on secret values in order to avoid unintended leaks (we give examples of such policies later). Otherwise, it would be possible to reveal information about secrets by inspecting the returned constructor (`Just` or `Nothing`) when applying escape hatches.

As mentioned in the beginning of the section, we include some *declassification combinators* that are responsible for generating escape hatches. The simplest combinator creates escape hatches that always succeed when downgrading information. Specifically, we define the following combinator.

```
hatch :: Less s' s => (a -> b) -> Hatch s s' a b
hatch f = \sa -> return(Just(return(f (reveal sa))))
```

Basically, `hatch` takes a function and returns an escape hatch that applies such function to a value of security level `s` and returns the result of that at security level `s'` where `s'` $\sqsubseteq$ `s`. Observe how the function `reveal` is used for declassification.

The idea is that the function `hatch` is used by trusted code in order to introduce a controlled amount of leaking to the attacker. Note that it is possibly dangerous for the trusted code to export a *polymorphic* escape hatch to the attacker! A polymorphic function can often be used to leak an unlimited amount of information, by for example applying it to lists of data. In general, escape hatches that are exported should be monomorphic.

5.2 The What dimension

In general, type systems that enforce declassification policies related to "what" information is released are somehow conservatives (Sabelfeld and Myers 2004; Askarov and Sabelfeld 2007; Mantel and Reinhard 2007). The main reason for that is the difficulty to statically predict how the data to be declassified is manipulated or changed by programs. Inspired by quantitative information-theoretical works (Clark et al. 2002), we focus on "how much" information can be leak instead of determining exactly "what" is leaked. In this light, we introduce the following declassification combinator.

```
ntimes :: Int -> Hatch s s' a b -> IO (Hatch s s' a b)
ntimes n f
 = do ref <- newIORef n
      return (\sa -> do k <- readIORef ref
                        if k <= 0
                           then do return Nothing
                           else do writeIORef ref (k-1)
                                   f sa )
```

Essentially, `ntimes` takes a number `n` and an escape hatch `h`, and returns a new escape hatch that produces the same result as `h` but that can only be applied at most `n` times. To achieve that, the combinator creates the reference `ref` to the number of times (`n`) that the escape hatch (`h`) can be applied. Every application of the escape hatch then checks if the maximum number of allowed applications has been reached by observing the condition `k <= 0`. Additionally, every application of the escape hatch also reduce the number of possible future applications by executing `writeIORef ref (k-1)`. The generated escape hatch returns `Nothing` if the policy is violated as a manner to avoid leaking more information than intended. Inspecting if the result of applying an escape hatch is `Nothing` or not can be considered as a covert channels by itself when happening inside of computations related to confidential data. Fortunately, escape hatches applied inside of computations depending on secrets are never executed. For instance, if we try to apply an escape hatch inside of some secret computation, it will have the type `Sec H (IO (Maybe (Sec L b)))` for some type `b`. Declassification is performed inside of the `IO` monad and it is not possible to extract `IO` computations from the monad `Sec H` unless than another escape hatches is declared to release `IO` computations. Therefore, escape hatches must be introduced to release pure values rather than side-effecting computations, which seems to be the case for most applications.

Note that the function `ntimes` is safe to be exported to the attacker, since it only restricts the use of existing hatches.

As an example of how `ntimes` can be used, we write a login program that uses the secure shadow password API described in Section 3.3. It is not possible to write such program without having means for declassification. The login program must release some information about users' passwords: if access is granted, then the attacker knows that his input matches the password, otherwise he knows that it does not. We present the program in Figure 7. Module `Policies` introduces declassification policies for our login program and states that a shadow password can be compared by equality at most three times. This module is trusted and must not be imported by untrusted code. Otherwise, attackers can create an unrestricted number of escape hatches in order to leak secrets!

```
module Policies ( declassification ) where
import SecLibTypes ; import Declassification
import SpwdData

declassification
 = ntimes 3 (hatch (\(spwd,c) -> cypher spwd == c))
   :: IO (Hatch H L (Spwd, String) Bool)


module Main ( main ) where
import Policies
import Login

main = do match <- declassification
          login match


module Login ( login ) where
import SecLibTypes ; import SecLib
import SpwdData ; import Spwd
import Maybe

check :: (?match :: Hatch H L (Spwd, Cypher) Bool)
         => Sec H Spwd -> String -> Int -> String
            -> IO ()
check spwd pwd n u =
  do acc <- ?match ((\s -> (s, pwd)) `fmap` spwd)
     if (public (fromJust acc))
               then putStrLn "Launching shell..."
               else do putStrLn "Invalid login!"
                       auth (n-1) u spwd

auth 0 _ spwd = return ()
auth n u spwd = do putStr "Password:"
                   pwd <- getLine
                   check spwd pwd n u

login match
  = do let ?match = match
       putStrLn "Welcome!"
       putStr "login:"
       u <- getLine
       src <- getSpwdName u
       case src of
            Nothing    -> putStrLn "Invalid user!"
            Just spwd  -> auth 3 u spwd
```

Figure 7. Secure login program

Module `SecLibTypes`, described in Section 4, is extended to include type definitions related to declassification as, for instance, `Hatch s s' a b`. Module `Declassification` introduces declassification combinators (e.g. `ntimes`). These modules are part of our trusted base. Module `Declassification` must not be imported by untrusted code for the same reasons given for module `Policies`. Modules `SpwdData` and `Spwd` respectively include the data type declaration of `Spwd` and the API described in Section 3.3. Module `Main` extracts declassification policies defined in `Policies` and pass them to the `login` function. In general, this module determines what functions are called from untrusted code in order to run the program. In this case, it determines that `login` must be called to perform the login procedure. Since the module imports module `Policies`, it also belongs to the trusted base. The most interesting module is `Login`. This module does not belong to our trusted base and therefore it may contain code written by possibly malicious programmers. Because declassification policies can be applied at any part of the untrusted code, we place them into implicit parameters (Lewis et al. 2000). Implicit parameters can be thought as some kind of global variables and they are declared by

```
module Bid ( bid ) where

obtainBid :: FilePath -> IO Int
obtainBid file = do s <- readFile file
                    return (read s :: Int)

bid  = do putStrLn "Bid system!"
          putStrLn "-----------"
          putStrLn ""
          putStrLn "Obtaining the bids..."
          a <- obtainBid "bidA"
          -- writeFile "bidB" (show (a+1))
          b <- obtainBid "bidB"
          putStrLn (if a > b then "A wins!"
                             else "B wins!")
```

Figure 8. Insecure bidding system

writing variable names starting with the symbol ?. Module `Login` contains three functions: `check`, `auth`, and `login`. Function `check` takes the password `spwd :: Sec H Spwd` stored in the system for the user `u :: String` and checks, by applying the escape hatch placed in `?match`, if the user's input `pwd :: String` matches the password stored in the field `cypher` of `spwd`. Assuming that is possible to perform the declassification described by `?match`, variable `acc` stores if the access is granted or not. We assume that untrusted code has access to the function `public :: Sec L a -> a` to extract the public values from monad `Sec L`. In the example, `public` is applied to values returned by `?match`. If the access is denied, `check` might give another chance to the user by calling the function `auth`. Function `auth` is responsible to ask the user's password and validates it at most `n` times. Function `login` asks for the user name and checks that the user is registered in the system by calling the function `getSpwdName` from the secure shadow password API.

Since program in Figure 7 type-checks, it respects the declassification policies defined in module `Policies`, i.e. the password can be compared for equality only three times. To illustrate that, we place our selves in the role of the attacker and modify function `check` to call `auth n u spwd` instead. As a result, it would be now possible to try as many passwords as the user wants and thus increasing the amount of information leak by unit of time. Observe that this situation is particularly dangerous when passwords have short length as PIN numbers in ATMs. Nevertheless, if we try to run the modified code, we get the message `*** Exception: Maybe.fromJust: Nothing` after the user tries more than three times to check if the access can be granted or not.

5.3 The When dimension

As a motivating example for handling this dimension, we can consider the scenario described in (Chong and Myers 2004) of a sealed auction where each bidder submits a single secret bid in a sealed envelope. Once all bids are submitted, the envelopes are opened and the bids are compared. The highest bidder wins. One security property that is important for this program is that no bidder knows any of the other bids until all the bids have been submitted. Program in Figure 8 simulates this process for two bidders: `A` and `B`. We represent envelopes as files. Function `obtainBid` opens an envelope and extracts the bid. The rest of the program is self-explanatory. It is possible to incorrectly implement the auction protocol by mistake or intentionally. For instance, if we uncommented the line in Figure 8, the program uses the bid from user `A` to make user `B` the winner. However, no information about `A`'s bid must be available until `B` submits his (her) own bid.

The library introduces the when dimension by associating events in the system that indicates at which time release of in-

formation may occur. For instance, "releasing a software key may occur after the payment has been confirmed". Inspired by (Broberg and Sands 2006), we implement boolean flags called *flow locks* [4] that, when open, allow downgrading of information.

Flow locks are introduced by the following combinator.

```
when :: Hatch s s' a b ->
        IO (Hatch s s' a b, Open, Close)
when f = do ref <- newIORef False
            return (\sa -> do b <- readIORef ref
                              if b then f sa
                                   else return Nothing
                   , writeIORef ref True
                   , writeIORef ref False)
```

Basically, `when` takes an escape hatch `h` and returns a new escape hatch that produces the same result as `h` but that has associated a flow lock to it. The combinator creates the reference `ref` to an initially close flow lock represented as `False`. The returned escape hatch can only be applied when the associated flow lock is open (i.e. the corresponding boolean flag is set to `True`). Observe that, by inspecting the value of `b`, every application of the escape hatch checks that the flow lock is open before declassifying information. The combinator also returns computations to open and close the lock, which respectively have type `Open` and `Close`. These computations must be only used by trusted code. Otherwise, the attacker can execute them at any time in the untrusted code and thus ignoring the events that indicate when declassification may occur. `Open` and `Close` are just synonymous type declarations for `IO ()`.

We can then implement a secure bidding system. We firstly define our security lattice composed by the security levels `A`, `B`, and `L`, where `L` $\sqsubseteq$ `A` and `L` $\sqsubseteq$ `B`. Security levels `A` and `B` are respectively associated to information coming from users `A` and `B`, while `L` denotes public information. We implement these security levels as singleton types with constructors `A :: A`, `B :: B`, and `L::L`. The described security lattice is very simple and therefore we omit details about its implementation. The secure bidding system is shown in Figure 9. At first glance, it might seem that this implementation is much more complex than the insecure one. However, the module `Bid`, the core of the bidding system, has approximately the same size as before. The rest of the modules are related to properly setting up the security level of different resources in the program as well as the corresponding declassification policies. Module `Files` declares the security level `A` and `B` for the files that store the bids of users `A` and `B`, respectively. Module `Policies` defines the escape hatches `ha` and `hb` to release information that belongs to users `A` and `B`, respectively. Computations `openA` and `closeA` (`openB` and `closeB`) open and close the flow lock associated to `hA`(`hB`), respectively. As mentioned before, the opening and closing of locks are produced by trusted code. In this case, the opening of locks happens when bids are read from files. We then place function `obtainBid` in the trusted module `Main`. We also adapt such function to read files at security level `s` and return their contents, but opening the flow lock received as argument. Function `main` obtains the escape hatches from `declassification` and defines trusted function responsible for opening flow locks. Function `obtainBidA` (`obtainBidB`) reads the bid of user `A` (`B`) and opens the lock for releasing the bid of user `B` (`A`). Differently from the insecure version in Figure 8, function `bid` receives as arguments escape hatches and functions to obtain bids. Module `Bid` is written by the attacker or possibly

[4] The notion presented here about flow locks is not exactly the same that is introduced in Broberg and Sands's paper. For instance, their work can statically check if a program respects the declassification policies determined by the flow locks. Moreover, the state of the locks is not related with the state of programs at all. We differ from these two points due to the dynamic nature of our approach. However, the intuitive idea of allowing downgrading of information when locks are open is preserved in our implementation.

```
module Files ( bidAF, bidBF ) where
import Sec (secret, File (File) ) ; import Lattice

bidAF :: File A
bidAF = MkFile "bidA"

bidBF :: File B
bidBF = MkFile "bidB"


module Policies ( declassification ) where
import SecLibTypes ; import Declassification

declassification
 = do (pA :: Hatch A L Int Int, openA, closeA)
        <- when (hatch id)
      (pB :: Hatch B L Int Int, openB, closeB)
        <- when (hatch id)
      return (pA, openA, closeA, pB, openB, closeB)


module Main ( main ) where
import Policies ; import Files ; import SecLib
import Bid

obtainBid :: File s -> Open -> IO (Sec s Int)
obtainBid file open
  = do sec <- run (do r <- s_read file
                      return (read r :: Int))
       open
       return sec

main = do (hA, openA, closeA,
           hB, openB, closeB) <- declassification
          let obtainBidA = obtainBid bidAF openB
              obtainBidB = obtainBid bidBF openA
          bid hA obtainBidA hB obtainBidB


module Bid ( bid ) where
import SecLibTypes ; import SecLib

bid hA obtainBidA hB obtainBidB
 = do putStrLn "Bid system!"
      putStrLn "-----------"
      putStrLn ""
      putStrLn "Obtaining the bids..."
      bidA   <- obtainBidA
      -- Just cheat <- hA bidA
      bidB   <- obtainBidB
      Just seca <- hA bidA
      Just secb <- hB bidB
      putStrLn(if (public seca) > (public secb)
                  then "A wins!"
                  else "B wins!")
```

Figure 9. Secure bidding system

malicious programmer. In this module, function `bid` obtains the bids to later compare them. In order to compare bids, they need to be extracted from values of type `Sec A Int` and `Sec B Int` through the escape hatches `ha` and `hb`, respectively. It is then not possible to determine which bid is the highest before obtaining all for them. For instance, if we uncommented the line in function `bid`, we obtain a program that tries to release the bid from user `A` before getting the bid for user `B`, which is clearly a non-desirable behavior for the auction system. However, if we run the program, we get the message `*** Exception: Maybe.fromJust: Nothing` since the flow lock associated to release `A`'s bid is not open. In order to open it, we firstly need to get `B`'s bid!

To illustrate why flow locks may need to be closed, we take the example on step further by thinking of a bidding system that allows the users to bid more than once. In this case, function `bid` is called several times and flow locks related to `hA` and `hB` must be closed between each call. Otherwise, all the flow locks are open at the second call of `bid`, which allows bids to be released at any time. It is not difficult to imagine this implementation by considering that function `main` calls computations `closeA` and `closeB` before each call of `bid`.

For simplicity, we considered an auction system with only two users. However, it is possible to use flow locks when more users are present in the auction. Indeed, we can create escape hatches that are associated to as many flow locks as users. In order to do that, we can compose `when` with itself as many times as users we have in the system. In this way, the escape hatch obtained in the end is associated to as many flow locks as users. Then, when a user submits its bid, his corresponding flow lock is open.

Attackers can still write programs that wrongly implement the auction system. For instance, we can write a program that makes user `A` the winner all the time by just replacing the `if-then-else` in Figure 9 by `putStrLn "The user A wins!"`. However, user `A` is going to be the winner because the program is not implemented correctly, but not because the program "cheated" by inspecting `B`'s bid. Correctness of programs are stronger properties than those ones captured by declassification policies.

5.4 The Who dimension

In the *Decentralized Label Model* (DLM) (Myers and Liskov 1997, 1998, 2000) data is marked with a set of principals who owns the information. While executing a program, the code is authorized to act on behalf of some set of principals known as *authority*. Then, declassification makes a copy of the released data and marked it with the same principals as before the downgrading but excluding those ones appearing in the authority of the code. We do not consider situations where some principals can act on behalf of others.

Similarly to (Li and Zdancewic 2006), we adapt the idea of DLM to work on a security lattice. Authorities are assigned with a security level l in the lattice and they are able to declassify data at that security level. To achieve that, we introduce a declassification combinator that checks the authority of the code before applying an escape hatch. As indicated in (Broberg and Sands 2006), DLM can be expressed using flow locks. Fortunately, our implementation is also suitable for that. More precisely, we have the following declassification combinator.

```
data Authority s = Authority Open Close

who :: Hatch s s' a b -> IO (Hatch s s' a b, Authority s)
who f = do (whof, open, close) <- when f
           return (whof, Authority open close)

certify :: s -> Authority s -> IO a -> IO a
certify s (Authority open close) io =
  s `seq` (do open ; a <- io ; close ; return a)
```

Combinator `who` takes an escape hatch an returns another escape hatch that is associated with a flow lock. The main idea here is that the flow lock is open when the code runs under the same authority as the security level appearing as the argument of the escape hatch. The mechanisms to open and close the flow lock are placed inside of the data type `Authority s`. The constructor of this data type is not accessible for attackers. Otherwise, they can avoid the certification process to determine that some piece of code runs under some authority. Such certification process is carried out by the function `certify`. This function takes an element of security type `s`, an `Authority s`, and a computation `IO a`. In Section 4, we explain that constructors that belongs to security levels above the security level of the attacker are not exported. For instance, in the two-point lattice considered so far, attackers can only observe data at security level `L`, and thus constructor `H :: H` is not exported to untrusted modules. This assumption needs to be relaxed in order to consider this dimension for declassification. To certify that some code has authority `s`, we require that such code, possibly malicious, has only access to the constructors for security level `s` and the security level denoting public information. In this way, it is reflected that code running under authority `s` can freely declassify data from security level `s` as expected in DLM. Function `certify` checks that it receives a valid constructor for the security type `s` by applying `seq` to it, and then respectively opens and closes a flow lock before and after running the `IO` computation received as argument. Observe that this function can be freely used by attackers since it requires to provide the right constructor for some security level `s` and only authorities at that level must have it. Therefore, assignments of authorities to pieces of code must be clearly part of the trusted code.

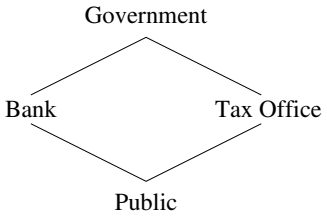

Figure 10. Security lattice

As a motivating example for this dimension, we start consider the security lattice in Figure 10. We have the security levels: Government, Bank, Tax Office, and Public to represent information related to citizens that is used for such entities. Unless that information is made public, banks cannot have access to information stored in the tax office and vice versa. Government, on the other hand, can have full access to the information stored at banks and the tax office, which can be debatable for any real government. However, we made such assumption to simplify the example and rather illustrate how functions `who` and `certify` can be used. We implement the security levels Government, Bank, Tax Office, and Public with the singleton types `G`, `B`, `T`, and `L`, respectively. The described security lattice is very simple and therefore we omit details about its implementation. We assume that the declassification polices are the followings: banks can declassify the status of their accounts (i.e. if an account is open or close), the tax office can release the address of the citizens, and the government can provide information about new immigrants to the tax office as well as revealing results of financial studies related to the economy of the country to the banks. Observe that, for instance, it is possible for the government to declassify some information to a bank, and then the bank divulges that information to the public by opening or closing some accounts. In order to avoid that, a more complex security lattice needs to be encoded. However, for simplicity, we tighten to the lattice in Figure 10. In Figure 11, we give the skeleton of an application that uses these security levels and the mentioned declassification policies. Module `Policies` declares declassification policies constructed by combinator `who`. Accounts, status of accounts, citizens, addresses, immigrants, financial studies, and outcomes of financial studies are represented by data types `Account`, `Status`, `Citizen`, `Address`, `Immigrant`, `Study`, and `Result`, respectively. Functions `status`, `address`, `immigrant`, and `study` have types `Account -> Status`, `Citizen -> Address`, `Immigrant -> Citizen`, and `Study -> Result`, respectively. These functions together with declarations of data types related to the application are placed in the module `Data`. Function `declassification`

```
module Policies ( declassification ) where
import SecLibTypes ; import Declassification
import Data

declassification
 = do (hB :: (Hatch B L Account Status),
       authBank) <- who (hatch status)
      (hT :: (Hatch T L Citizen Address),
       authTax)  <- who (hatch address)
      (hG :: (Hatch G T Immigrant Citizen),
       authG) <- who (hatch inmigrants)
      (hG' :: (Hatch G B Study Result),
       authG') <- who (hatch studies)
      return ((hB, authBank), (hT, authTax),
              (hG, authG), (hG', authG'))

module Bank ( bank ) where
import SecLibTypes ; import SecLib
import Data

bank :: B -> (Hatch B L Account Status,
              Authority B) -> IO ()
bank = ...

module TaxOffice ( taxoffice ) where

import SecLibTypes ; import SecLib
import Data

taxoffice
 :: T -> (Hatch T L Citizen Address, Authority T)
    -> IO ()
taxoffice = ...

module Government ( government ) where
import SecLibTypes ; import SecLib
import Data

government
 :: G -> (Hatch G T Immigrant  Citizen,
    Authority G) -> (Hatch G B Study Result,
    Authority G) -> IO ()
government = ...


module Main ( main ) where
import Policies ; import Lattice
import Bank ; import TaxOffice ; import Government

main
  = do (whohB,  whohT, whohG, whohG')
         <- declassification
       bank B whohB
       taxoffice T whohT
       government G whohG whohG'
       return ()
```

Figure 11. Skeleton for an application

implements the declassification policies described before. Modules `Bank`, `TaxOffice`, and `Government` are untrusted and they might include malicious code. Functions `bank`, `taxoffice`, and `government` receive the escape hatches together with values of type `Authority s` for some corresponding instances of `s`. Observe that `bank`, `taxoffice`, and `government` expects to receive the constructor for security types `B`, `T`, and `G`, respectively. In other words, the authority for `bank`, `taxoffice`, and `government` is set to `B`, `T`, and `G`, respectively. Consequently, it is then possible for those functions to apply `cerfity` with escape hatches that release information at their authority level. Module `Main` sets the authority for each of the given functions while providing the corresponding escape hatches. Observe how constructors `B :: B`, `T :: T`, and `G :: G` are given to functions `bank`, `taxoffice`, and `government`, respectively. Malicious code placed in one function only compromises confidential information related to its authority's security level. For instance, if function `bank` contains malicious code, then confidential information related to the bank may be at risk. However, if `government` is compromised, all the information in the system may be affected. Function `government` should be carefully designed, or perhaps other restrictions regarding the application of the escape hatch must be imposed in this function (see next subsection). This phenomenon also occurs in `DLM` when a process running with the authority of all the principals in the system contains malicious code.

5.5 Combining dimensions

For some application, declassification policies are not so simple as those ones captured by the dimensions of what, when, and who. For those scenarios, the user of the library has basically two options. One one hand, the user can program his own policy, which provides enough flexibility. However, such flexibility could be dangerous when declassification policies are not implemented carefully. For instance, an escape hatch must not decide if declassification is possible by inspecting confidential data. Otherwise, attackers learn information about secrets when applying escape hatches by inspecting if the returned values are `Nothing` or not. On the other hand, users can specify more interesting declassification policies by combining applications of `ntimes`, `when`, and `who` together. For instance, we extend the *what*-policy from the example given in Section 5.2 to consider more dimensions as follows.

```
comb = do h <- ntimes 3
              (hatch (\(spwd,c) -> cypher spwd == c))
          (h', open, close) <- when h
          (h'', auth) <- who h'
          return (h''':: Hatch H L (Spwd, String) Bool,
                  open, close, auth)
```

Observe how `comb` defines an escape hatch that releases information if it is applied in a piece of code with authority `H` when some events that execute `open` happened and information has not been previously released more than three times. Other combinations are also possible. To the best of our knowledge, this is the first implementation of mechanisms to enforce more than one dimension for declassification.

6. Related work

Much previous related work addresses non-interference and functional languages consider reduced programming languages (Heintze and Riecke 1998; Volpano et al. 1996; Volpano and Smith 1997) or require designing compilers from scratch (Pottier and Simonet 2002; Simonet 2003). Rather than implementing compilers, Li and Zdancewic (Li and Zdancewic 2006) show how to provide information-flow security as a library for a real programming language. They provide an implementation for Haskell based on arrows combinators(Hughes 2000), which naturally requires programmers to be familiar with arrows when writing security-related code. Their library still imposes restrictions on what kind of programs can be written. In particular, their approach does not generalize naturally in the presence of side-effects or information composed of data with different security levels. To incorporate these features, the library requires major changes as well as the introduction of new combinators (Tsai et al. 2007).

In this paper, we show that a less general notion, namely monads, is enough to provide information-flow security as a library. We propose a light-weight library ($\sim$ 400 LOC) able to handle side-effecting computations and that requires programmers to be familiar with monads rather than arrows. Moreover, by just placing data

into corresponding `Sec s` monads, our library is also able to handle data composed of elements with different security levels. However, there exists one restriction in our approach w.r.t. to the arrow approach. Since our security levels are represented by types, all of them have to be known statically at compile-time[5], whereas in the arrow approach, they can be constructed at run-time.

Abadi et. al. developed the *dependency core calculus* (DCC) (Abadi et al. 1999) based on a hierarchy of monads to guarantee non-interference. Similarly, `Sec` constructs a hierarchy of monads when applied to security levels `s`. However, DCC uses non-standard typing rules for its *bind* operations while our library just provides instances of the type class `Monad`. Tse and Zdancewic translate DCC to System F and show that non-interference can be stated using the *parametricity* theorem for F (Tse and Zdancewic 2004). They also provide an implementation in Haskell for a two-point lattice. Their implementation encodes each security level as an abstract data type constructed from functions and binding operations to compose computations with permitted flows. The same kind of ideas relies behind `Sec s`, `open`, and `close` (see Section 4). Their implementation requires, at most, $O(n^2)$ definitions for binders for n-points lattices. Since they consider the same non-standard features for binders as in DCC, they provide as many definitions for binders as different type of values produced after composing secure computations. Moreover, their implementation needs to be compiled with the flag `-fallow-undecidable-instances` in GHC. On one hand, our library requires, at most, $O(n^2)$ instantiations on the type class `Less` for n-points lattices, but it does not provide more than one definition for binders nor requires allowing undecidable instances in GHC [6]. DCC and Tse and Zdancewic's approach do not consider computations with side-effects. Moreover, Tse and Zdancewic leaves as an open question how to encode more expressive policies, such as declassification, directly in the type system of Haskell.

Harrison and Hook show how to implement an abstract operating system called *separation kernel* (Harrison and Hook 2005). Programs running under this multi-threading operating system are non-interferent. To achieve that, the authors rely on properties related to monad transformers as well as state and resumption monads. Basically, each thread is represented as an state monad that have access to the locations related to the thread's security level while state monad transformers act as parallel composition. Interleaving and communication between threads is carried out by plugging a resumption monads on top of the parallel composition of all the threads in the system. Non-interference is then enforced by the scheduler implementation, which only allow signaling threads at the same, or higher, security level as the thread that issued the signal. Different from that, our library enforces non-interference by typing. The authors also use monads differently than we do since their goals are constructing secure kernels rather than providing information-flow security as a library. For instance, we do not use state monads, state transformers, or resumption monads since we do not model threads. As a result, our library is simpler and more suitable to write sequential programs in Haskell. It is stated as a future work how to extend our library to include concurrency.

Crary et. al. design a monadic calculus for non-interference for programs with mutable state(Crary et al. 2003). Their language distinguishes between *term* and *expressions*, where terms are pure and expressions are (possibly) effectful computations. The calculus mainly tracks flow of information by inspecting the security levels of effects produced by expressions. Expressions can be included at the term level as an element of the monadic type $\bigcirc_{(r,w)} A$, which denotes a suspended computation where the security level r is an upper bound on the security levels of the store locations that the suspended computation reads, while w is a lower bound on the security level of the store locations to which it writes. Authors introduce the notion of *informativeness* in order to relax some typing rules so that reading and writing into secret store locations can be included in large computations related to public data. A type A is informative at security level r or above if its values can be used or observed by computations that may read data from security level r or above. In our library, the type `SecIO s a` makes the value of type `a` only informative at security level `s`. In principle, the value of type `a` cannot be used anywhere but inside the monad `SecIO s`. Considering a two-point lattice, we introduce the function `plug :: Less L H => SecIO H a -> SecIO L (Sec H a)` to allow reading and writing secret files into computations related to public data. Observe that the function preserves the informativeness of `a` by placing it inside of the monad `Sec H`.

Recently, several approaches have been proposed to dynamically enforce non-interference (Guernic et al. 2006; Shroff et al. 2007; Nair et al. 2007). In order to be sound, these approaches still need to perform some static analysis prior to or at run-time. Authors argue, in one way or another, that their methods are more precise than just applying an static analysis to the whole program. For instance, if there is an insecure piece of dead code in a program, most of the static analysis techniques will reject that program while some of their approaches will not. The reason for that relies in the fact that dead code is generally not executed and therefore not analyzed by dynamic enforcement mechanisms. Our library also combines static and dynamic techniques but in a different way. Non-interference is statically enforced through type-checking while run-time mechanisms are introduced for declassfication. By dynamically enforcing declassification policies, we are able to modularly extend the part of the library that enforce non-interference to add downgrading of information and being able to enforce several dimensions for declassification in a flexible and simple manner. To the best of our knowledge, this is the first implementation of declassification policies that are enforced at run-time and the first implementation that allows combining dimensions for declassifications.

7. Conclusions

We have presented a light-weight library for information-flow security in Haskell. Based on specially designed monads, the library guarantees that well-typed programs are non-interferent; i.e. secret data is not leaked into public channels. When intended release of information is required, the library also provides novel means to specify declassification policies, which comes from the fact that policies are dynamically enforced and it is possible to construct complex policies from simple ones in a compositional manner.

Taking ideas from the literature, we show examples of declassification policies related to what, when, and by whom information is released. The implementation of the library and the examples described in this paper are publicly available in (Russo et al. 2008a). The well-known concept of monads together with the light-weight and flexible characteristic of our approach makes the library suitable to build Haskell applications where confidentiality of data is an issue.

Acknowledgments

We wish to thank to Aslan Askarov, Ulf Norell, Andrei Sabelfeld, David Sands, Josef Svenningsson, and the anonymous reviewers for useful comments and discussions about this work. This work was funded in part by the Information Society Technologies program of the European Commission,

[5] We are investigating the use of polymorphic recursion to alleviate this – this remains future work however.

[6] All the code shown in the paper works with the Glasgow Haskell Compiler (GHC) with the flag `-fglasgow-exts`

Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, January 1999.

A. Askarov and A. Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 53–60, New York, NY, USA, 2007. ACM.

N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In Peter Sestoft, editor, *Proc. European Symp. on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2006.

S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, October 2004.

D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In *QAPL'01, Proc. Quantitative Aspects of Programming Languages*, volume 59 of *ENTCS*. Elsevier, 2002.

E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state, 2003.

D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *Proc. Annual Asian Computing Science Conference*, volume 4435 of *LNCS*, pages 75–89. Springer-Verlag, December 2006.

W. L. Harrison and J. Hook. Achieving information flow security through precise control of effects. In *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 16–30, Washington, DC, USA, 2005. IEEE Computer Society.

N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, January 1998.

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

M. H. Jackson. Linux shadow password howto. Available at `http://tldp.org/HOWTO/Shadow-Password-HOWTO.html`, 1996.

J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: dynamic scoping with static types. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 108–118, New York, NY, USA, 2000. ACM.

P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

P. Li and S. Zdancewic. Arrows for secure information flow. Available at `http://www.seas.upenn.edu/~lipeng/homepage/lz06tcs.pdf`, 2007.

Local Root Exploit. Linux kernel 2.6 local root exploit. Available at `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=465246`, February 2008.

H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In Rocco De Nicola, editor, *European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*, pages 141–156. Springer, 2007. ISBN 978-3-540-71314-2.

A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997.

A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998.

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

S. K. Nair, P. N.D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *The First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)*, September 2007.

A. Narayanan and V. Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 364–372, New York, NY, USA, 2005. ACM.

B. C. Pierce. *Advanced Topics In Types And Programming Languages*. MIT Press, November 2004. ISBN 0262162288.

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

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. Software release and documentation. Available at `http://www.cs.chalmers.se/~russo/seclib.htm`, 2008a.

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. Technical Report. Chalmers University of Technology. To appear., October 2008b.

A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.

A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW '05: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 255–269. IEEE Computer Society, 2005.

P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 203–217, 2007.

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

A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. ISBN 0130313580.

T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, July 2007.

S. Tse and S. Zdancewic. Translating dependency into parametricity. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 115–125, New York, NY, USA, 2004. ACM.

D. Volpano and G. Smith. A type-based approach to program security. In *Proc. TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621. Springer-Verlag, April 1997.

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

P. Wadler. Monads for functional programming. In *Marktoberdorf Summer School on Program Design Calculi*, August 1992.