CHALMERS |  UNIVERSITY OF GOTHENBURG

Providing Integrity Policies as a Library in Haskell

Master of Science Thesis in the Computer Science Programme

Albert Diserholt

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, Mars 2010

Providing Integrity Policies as a Library in Haskell

Albert Diserholt

Examine: Dr. Koen Claessen
Supervisor: Dr. Alejandro Russo

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden Mars 2010

Abstract

This work describes an extension made to a security library in Haskell that guarantees confidentiality (i.e. that secrets are not leaked). The existence of this library is possible due to Haskell's strong type checker and controlled side-effects. This security library uses monads with type parameters to indicate security levels. The monad guarantees that once data is inside of it, there is no way of getting it out. More precisely stated, the monad can be used to enforce information-flow policies for confidentiality.

This work shows that integrity policies (i.e. policies that avoids data from being intentionally or unintentionally destroyed) can be incorporated as well, and thus defining a richer set of security policies. More precisely, the integrity policies that are incorporated are *Access-Control*, *Data Invariant* and *Information-flow Integrity Policies*. Access-control is implemented using a wrapper data type to define permissions for resources (a variation of an access-control list (ACL)), and uses a security lattice for permissions. Data invariants are enforced at end points (inputs and outputs to a program). This is done by extending the data types for files and checking that the property holds, when an input or output operation is performed. Information-flow integrity policies reuses the same techniques that enforce confidentiality, by simply extending the security lattice with integrity levels.

Confidentiality and integrity are two key aspects of information security, and by incorporating both in a light-weight library, we seek to increase the use of language-based security when writing security critical applications.

A *Password Administrator* is used as an interesting case study. The case study is organized in the modules `Login`, for authenticating, `Reset`, for changing passwords, and `Backup` for backing up and restoring passwords. This application incorporates confidentiality and all the different integrity policies and described above.

Acknowledgment A lot of gratitude goes out to my supervisor Alejandro Russo, for presenting an exciting thesis idea and helping me at every step of the way.

Contents

CHAPTER 1

Introduction

The use of computers has expanded rapidly the last decade. Data and information that used to be stored in paper files and archives are now stored into hard drives. More and more digital devices are sprouting up and interconnecting for fast access and usability. This tendency can leave a questionable sense for security. For example, a web page can save credit card numbers in their database for users' convenience, so they do not have to type it every time when making a purchase. This sensitive information should not be leaked or exploited in any way. Some form of trust has to be placed upon the companies that provide services in order to handle sensitive data properly. Flaws and errors might happen at the code level of the application, which might cause intentional or unintentional leaks, corruption of data, erroneous states, etc. One way of aiding against these types of security breaches is by adopting *language-based security*. Language-based security defends against attacks at the application level by specifying security policies. More specifically, there are security-typed languages and extensions that supports information flow control and other constructs, which restricts how information flows inside programs. There exists programming languages that provide this technology: Jif [1, 2, 3, 4](based on Java), Flow Caml [5, 6](based on Caml) and SPARK [7, 8] (based on Ada). To adopt those languages, programmers require to learn a new language or syntax, and consider trade-offs between security and other aspects, like performance or maintainability. An alternative consists on providing security enforcements as a library, which would allow the use of flexibilities and strengths of the chosen programming language, while using the library's functions to avoid breaches. This report describes a security library for Haskell.

Haskell has a strong type checker and controlled side effects, which makes

it a good candidate for a security library. To aid the process, modules are distinct between trusted and untrusted. A trusted module describes the policies and resources accessible to the untrusted modules. Untrusted modules are further divided into non-malicious and potentially malicious. A non-malicious module would typically be written by programmers with no intention to do harm, for example, programmers at the same company. A potentially malicious module would typically be written by programmers that cannot be trusted, for example, programmers from different companies. The untrusted modules must not get access to the implementation details of security types or certain functions, otherwise security could be compromised. Instead, the untrusted modules get access to a subset of functions and resources through data abstraction. With a correct set up from the trusted module, the untrusted ones are not able to violate security.

Security measurements are considered from two points of view: *Confidentiality* and *Integrity*. Confidentiality ensures that sensitive information is kept secret and what are the allowed disclosing mechanisms (declassification). Integrity seeks to prevent the intentional or unintentional destruction of data. More specifically, integrity policies can be classified into three categories: *Access control*, *Data invariant* and *Information-flow policies*[9]. Access-control policies govern access to data (e.g. an application can have read permission to a file, but cannot be able to write to it). Data invariant policies govern the meaningfulness of data (e.g. a credit card number must adhere to certain restrictions to be correct), and lastly information-flow policies restrict the ways untrustworthy data can flow inside programs (e.g. a vehicle's entertainment system should not affect the cruise control system, while the cruise control system should be able to shut down the entertainment system in case of emergency).

Contributions The aim of this work is to extend the security library by Russo et al.[10] to include integrity policies. To do that, we explore what kind of integrity policies can be provided via a library. The difference between non-malicious and potentially malicious modules is discussed, to clearly shows limitations and restrictions. Access control, data invariant and information-flow policies are combinable and thus being able to express a richer set of security policies. There will be focus on maintaining the library light-weight, modular and convenient to use, since the nature of a library makes it easier to be adopted by programmers. We show that access-control can be implemented using a simple data type and reusing the security lattice, that data-invariants can be implemented as simple properties at end points (inputs and outputs), and that information-flow integrity policies can be implemented reusing the security lattice.

This report is structured as follows: Chapter 2 goes deeper into the work done by Russo et al.[10]. Chapter 3 explains integrity policies and describes the implementation of integrity in the library and the case study. Chapter 4 discusses the current and future work, and Chapter 5 draws conclusions.

CHAPTER 2

Confidentiality as a library in Haskell

This chapter describes the security library by Russo et al.[10], which enforces confidentiality policies. Some small modifications are made to the library, and reasons behind those changes will be detailed.

2.1 Non-Interference

As described in the introduction, confidentiality is about secrecy and preventing disclosure of sensitive information. One security policy that the library focuses upon is *non-interference*. Non-interference is an information-flow policy for confidentiality, which prevents leaking secret data into public channels. Non-interference policies address two types of leaks; *explicit* and *implicit* flows. Explicit flows leak sensitive information by assigning secret data into public variables, while implicit flows leak data indirectly, by performing branching instructions based on secrets. In a pure functional language, however, this distinction becomes less meaningful, since there are no assignments or control constructs. For example, an if-then-else condition is treated as a function where the information only flows from the function's arguments to the function's results.

To illustrate an explicit flow in Haskell, assume that the function `f :: (Int, Int) -> (Int, Int)` accepts a tuple as an argument, expecting the first component of the pair to be secret and the second one to be public. Function `f` performs some computations on the secret and public inputs, and returns a tuple as a result. In the resulting tuple, the first and second components are considered secret and public, respectively. The public input can affect the secret output, but the secret input should not affect the public output. Otherwise, the non-interference policy is violated. Assuming that a

```
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 2.1: The Lattice module

potentially malicious programmer writes function `f`, then the two following implementations of function `f` adhere to the non-interference policy:

```
f  (i, j) = (i+1, j-1)
f' (i, j) = (i+j, j)
```

In contrast, there is nothing preventing the programmer to write any of the following functions:

```
f  (i, j) = (i, i)
f' (i, j) = (i, if i > 5 then 0 else 1)
```

The new versions of functions `f` and `f'` leak information about the secret, thus violating the non-interference policy.

2.2 Security Lattice

A lattice of security levels is used to model how information can flow inside programs[11]. The lattice defines an ordering relationship, $\sqsubseteq$, based on which direction information can flow. In this case, information is allowed to flow from lower to higher positions in the lattice. The relation $l_1 \sqsubseteq l_2$ describes that information associated with the security level l_1 can flow into entities at the security level l_2. For simplicity, only two security levels are

```
newtype Sec s a

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

Figure 2.2: The `Sec s` monad

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

class Attacker s where
    public :: Sec s a -> a
```

Figure 2.3: Combinators for `Sec s`

considered: H (high or secret) and L (low or public). The non-interference policy then allows data to flow from low to high (written as $L \sqsubseteq H$), but disallows flows from high to low (written as $H \not\sqsubseteq L$).

In the library by Russo et al.[10], security levels are defined as singleton types and the security lattice is then encoded using the type class `Less`, as shown in Figure 2.1. The type class `Less` is flexible enough to encode more complex security lattices by simply defining class instances of allowed flows. The role of method `less` is explained in Section 2.6.

2.3 A Security Monad for Pure Computations

To keep track of how information flows inside a program, a monad is used in order to make a distinction between data at different security levels. The abstract type `Sec s` in Figure 2.2 is a monad[12, 13] and also a functor[14]. A value of type `Sec s a` indicates that the computed value of type `a` is at the security level `s`. Besides the return and bind combinators, the `Sec s` monad involves the combinators described in Figure 2.3. The combinator `up` is used to capture the fact that information can flow from lower to higher security levels in the security lattice. Observe that `up` requires a `Less` instance from `sl` to `sh`. The class `Attacker s` is introduced by us in this work, to indicate the observable capabilities of the attacker. The method `public` allows the attacker to have access to data inside the `Sec s` monad, providing that data with confidentiality level `s` can be observed by the attacker. For the two point lattice (described in Figure 2.1), the attacker can only observe data at security level `L`, which is captured by creating an instance of `Attacker L`. In [10], restrictions regarding exporting constructors for security levels are imposed, instead of the `Attacker s` class, and the reasoning behind the

```
module SecIO where

newtype SecIO s a

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

Figure 2.4: The `SecIO s` monad

```
data File s = MkFile FilePath
```

Figure 2.5: The `File s` constructor

change is detailed in Section 4.3.

To ensure that function `f` is non-interferent, we modify its signature by:

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

Since `Sec H` is an abstract type, the only way to write `f` adheres to the non-interference policy:

```
f (si, j) = ((\i -> i+p) `fmap` si, j+3)
```

There is no way of adding the protected variable `si` to the public output `j` without returning a value of type Sec H, since it is impossible to go outside of the monad (with the restrictions imposed in Section 2.6). This can be formally stated that type `Sec` guarantees a non-interference property.

2.4 A Security Monad for Side Effects

The `Sec s` monad is not enough to deal with side effects. When reading a file containing secrets and writing to another file, the computation will have the type `IO (Sec H (IO ()))`, which quickly becomes unmanageable and unusable. Also, using standard IO poses no restrictions on the side effects that can be executed. The type `Sec H (IO ())` allows the inner `IO` to leak information about the secret.

A new monad, called `SecIO s` as shown in Figure 2.4, is introduced to guarantee that side-effects are safe to perform. Similar to `Sec s`, this monad is an abstract data type. Intuitively, the type `SecIO s a` represents computations that are allowed to write to entities of security label `s` or higher, and where `a` has confidentiality level, at least, `s`.

```
value :: Sec s a -> SecIO s a

readFileSecIO  :: File s -> SecIO s' (Sec s String)
writeFileSecIO :: File s -> String -> SecIO s ()
```

Figure 2.6: Methods for `SecIO s`

The `SecIO` module also defines entities with security labels `s`. For brevity, only file resources are shown and considered in this paper, but other entities function in a similar fashion. For example, files are defined as shown in Figure 2.5. The data type `File s` represents files which content has confidentiality level s.

Additional combinators, detailed in Figure 2.6 are introduced to perform controlled side-effects inside `SecIO s`. The combinator `value` transforms pure computations into side-effect ones. The method `readFileSecIO` reads a file with security label `s`, and then returns `SecIO s'` with any level for `s'`, and the string read protected in the security monad with security label `s`. The method `writeFileSecIO` writes the string to the file with security label `s`, and returns `SecIO s` with the security level of the file.

2.5 Implementation Details of SecIO

This section briefly covers the implementation details found within the `SecIO s` module, more specifically `readFileSecIO` and `writeFileSecIO`. Later chapters build upon the basic implementation of these methods to extend them with integrity policies.

The `SecIO s` type is defined as such:

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

The type `SecIO s` includes a security monad wrapped inside an `IO` monad. The `IO` is used to perform safe side effects by `SecIO s`, as the following implementation will show:

```
readFileSecIO :: File s -> SecIO s' (Sec s String)
readFileSecIO (MkFile f) =
   MkSecIO $
   return `fmap` readFile f

writeFileSecIO :: File s -> String -> SecIO s ()
writeFileSecIO (MkFile f) s =
   MkSecIO $
   return `fmap` writeFile f s
```

In the implementation, both `readFileSecIO` and `writeFileSecIO` call the prelude functions `readFile` and `writeFile`, respectively. The result is then wrapped inside the security monad and then the `SecIO s` monad.

2.6 The Security Library's API

The API provided by the library (shown in Figures 2.1, 2.2 and 2.4) is almost sufficient to write interesting programs. The modules `Sec`, `SecIO` and `Lattice` should only be imported by trusted programmers, while untrusted programmers only are allowed to import the module called `SecLib`. `SecLib` includes functions from the previous three modules, but only exports safe functions, and is defined as follows:

```
module SecLib (
   -- Sec module
     Sec
   , up
   -- Attacker
   , public
   -- SecIO module
   , SecIO
   , value
   , plug
   -- Files
   , File
   , readFileSecIO
   , writeFileSecIO
   -- Lattice module
   , L (..)
   , H (..)
   , Less ()
   )
where

import Lattice
import Sec
import SecIO
```

It is important that the method `less` for the type class `Less` is not exported. Otherwise untrusted code would be able to alter the security lattice and thus violating, for example, the non-interference policy. Only the import list has to be examined to ensure that untrusted modules are safe. More specifically, the following modules should not be imported:

- `Sec`, `SecIO` and `Lattice`
- any module that provides exception handling, for example `Control.Monad.Exception`
- any module that provides unsafe extensions, for example `System.IO.Unsafe` (unsafePerformIO)

A typical example of import relationships between modules are depicted in Figure 2.7. There, the module `MainX` is trusted, and has the duty to initiate the program and call functions in the module `X`, which is non-malicious or potentially malicious. Module `X` imports `SecLib`, and uses the secure functions which are exported there. When properly setting up confidentiality and integrity policies in module `MainX`, such policies can be enforced[1]. The programmers for module `X` is free to use feature of the programming language when considering the implementation (e.g. recursion, function calls, type annotations, etc).

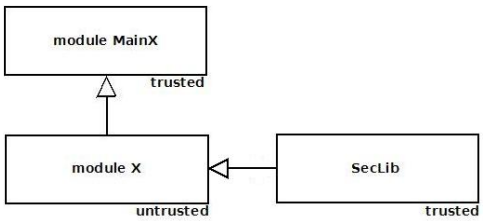


Figure 2.7: Module Relationships

2.7 A Motivating Example

To show how to use the library, we present here the partial implementation of a password administrator. A program that wishes to verify passwords usually needs to be run by root. A lot of care has to be taken when writing such a program, so no passwords are leaked. For example, in Linux[15], the file */etc/passwd* contains information about the users: for example their ID and user name. This file is essential and used by many system programs. The file */etc/shadow* contains passwords and can only be read and written with root permissions. If the file was readily available for anyone to read, it would be subject to off-line dictionary attacks[16] (using cyphered passwords and encrypting dictionary words in an attempt of finding a match). Assuming that the only way of accessing the `shadow` file is through the library, we show that dictionary attacks become impossible, thus making root privileges unnecessary when dealing with the `shadow` file.

First thing to define is the access to the files:

```
module Files (passwdFile, shadowFile) where
-- Trusted module
```

[1] Information-flow integrity policy in potentially malicious sources do not provide the desired guarantees. More information can be found in Section 4.2

```
-- Untrusted module Spwd
getSpwdName :: File L -> -- Passwd file
               File H -> -- Shadow file
               Name -> SecIO L (Maybe (Sec H Spwd))

putSpwd     :: File H -> -- Shadow file
               Spwd -> SecIO H ()
```

Figure 2.8: The Password Administrator API

```
passwdFile :: File L -- "/etc/passwd"
shadowFile :: File H -- "/etc/shadow"
```

The file `passwdFile` represents the file `/etc/passwd` and has the security level `L`, since the content is *public*. The file `shadowFile` represents the file `/etc/shadow` and has the security level `H`, since the content is *secret*.

Some data types are defined to represent the data stored in the files:

```
module SpwdData (Spwd, Name) where
-- Trusted module

type UID  = Int
type Name = String
type Cypher = String
data Spwd = Spwd { uid :: UID, cypher :: Cypher }
```

A `Spwd` data type represents a value stored inside the `shadowFile`, which consists of a user id and a cypher string. Names inside the `passwdFile` is represented as strings.

The API for accessing the files is shown in Figure 2.8. The method `getSpwdName` expects the `passwd` and `shadow` files as parameters, and a name. The `passwd` file is searched for the name, and if found, it returns the corresponding match in the `shadow` file. The match is returned with security level `H` (since the `shadowFile` has security level `H`), or `Nothing` is returned if no match was found. The returning `SecIO` has security level `L`, which poses no security vulnerability since the interior security monad is protected by level `H`. The method `putSpwd` accepts the `shadow` file, and a `Spwd` data type with a user id and password. The method inserts a new entry in the `shadow` file, or overwrites the old password if the user exists. The returned value is of the type `SecIO H`, since a secret file is written to.

The signatures are slightly altered compared to the example in [10]. The files are sent as parameters, which is due to accommodating for access

control later on in Section 3.1.2. The return type of `IO a` was changed to `SecIO s a` as well, since using `IO` poses no restrictions on the side-effects an untrusted source can perform (this is explained further in Section 2.8.4).

As we assumed earlier, `getSpwdName` is the only way of reading information from the `shadow` password file, and the cyphered password is not leaked or accessible inside the monad, making off-line dictionary attacks impossible. Using the API provided by `SecLib` also means that confidentiality is preserved and the number of possible actions on the `shadow` password file is limited, making root privilege unnecessary (with regards to accessing and writing to the `passwd` and `shadow` files). Other untrusted modules, for example a `Login` module, can import the `Spwd.hs` file and use the two functions, without being able to violate confidentiality.

2.8 Declassification

The non-interference policy specifies that no flow of information is allowed from secret to public data. However, this restriction makes it difficult to implement certain real-world applications, which release specific information as their intended behavior. Taking the above API (in Section 2.7), it is not possible, for example, to create a login prompt. The return type of `getSpwdName` is `Sec H Spwd`, which contains the id and password of the user specified by the `Name` parameter. It is possible to compare a string with the password inside the monad `Sec H`, but the resulting boolean will also have the security level `H`. From here on, there is no manner of publishing the result (if the match succeeded or failed) to the user, due to the nature of the non-interference policy.

Declassifications policies relax the notion of non-interference to allow a controlled leak of information. The chosen approach is a run-time mechanism for declassification, which allows downgrading of information. This mechanism introduces *escape hatches* [17], which are defined in the file `Declassification.hs`. We altered the signature of escape hatches, shown in [10], and the difference is detailed in 2.8.4. Our representation of escape hatches is as follows.

```
type Hatch s s' a b = Sec s a -> SecIO s' b
```

A hatch is a function that takes secure data of security level `s` with type `a`, and produces a secure side-effect computation with security level `s'` and type `b`. This representation of escape hatches allows for a lot of flexibility when defining declassification policies. Only trusted code is allowed to define escape hatches and specify intended release of information. Thus, untrusted code should not import the file `Declassification.hs`.

The library provides combinators to work with escape hatches. The combinators provide means to deal with *what*, *when*, *where* and by *whom* data is

released[18]. The simplest combinator always succeeds when downgrading information, and is defined as follows.

```
hatch :: Less s' s => (a -> b) -> Hatch s s' a b
```

The combinator `hatch` takes a function and returns an escape hatch that applies the given function as long as $s' \sqsubseteq s$.

The remainder of this section briefly explains the different dimensions. The different dimensions are explained further in [10], coupled with examples and code.

2.8.1 The What Dimension

In general, the enforcement for declassification policies mechanisms related to the *what* dimension are somewhat conservative[17, 19, 20]. The reason is that it is difficult to statically predict how the data to be declassified is manipulated or changed by programs. Instead, the focus is shifted towards "how much" information can be leaked. In light of this, a new combinator is added:

```
ntimes :: Int -> Hatch s s' a b -> IO (Hatch s s' a b)
```

Basically, `ntimes` takes a number `n` and a hatch, and returns an escape hatch that can be applied `n` times. The `IO` return type is safe here, since all declassification combinators are for trusted code only.

To illustrate how the `ntimes` combinator works, consider the (untrusted) `Login` module for the password administrator. The escape hatch defined for that modules is as follows:

```
module Policies where
-- Trusted module
import Declassification

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

The hatch `match_passwd` takes a secret `Spwd` parameter, which includes a user's ID and password, and a public `Cypher` parameter, which is a string representing the password the user typed in. Then, the password is extracted from `Spwd` and compared with the `Cypher`, and the resulting boolean is leaked. The `ntimes` combinator assures that `match_passwd` cannot leak information more than three times per program run. Attempting to apply it a forth time will result in a program crash. This is useful to slow down automated scripts attempting to compromise accounts.

2.8.2 The When Dimension

The *when* dimension specifies a declassification policy for when information can be released. This is implemented using a notion similar to flow locks[25]. Flow locks are introduced by the following combinator:

```
when :: Hatch s s' a b -> IO (Hatch s s' a b, Open, Close)
```

Given an escape hatch, `when` returns the escape hatch and two functions to open and close the lock, respectively. Using `Open`, which has the type `IO ()`, will open the flow lock it is associated with, thus making the escape hatch able to declassify information. `Close` can be used in the same fashion to close the lock.

2.8.3 The Who Dimension

In the *Decentralized Label Model* (DLM)[21, 22, 23], data is tagged with an *authority* that owns it. These authorities are assigned security levels in the security lattice[24], and are able to declassify information at that level. A combinator is introduced to check the authority of the executing code before applying an escape hatch.

```
data Authority s = Authority Open Close

who :: Hatch s s' a b -> IO (Hatch s s' a b, Authority s)

certify :: s -> Authority s -> IO a -> IO a
```

The *who* implementation uses a variation of flow locks[25], as seen above. The data type `Authority` has a security level `s` associated with it. The combinator `who` takes an escape hatch and returns the escape hatch and an authority data type for declassifying at level `s`. The returned escape hatch will release information when the authority of the level is authenticated for security level `s`. To prove that an authority has security level `s`, the function `certify` is used. The function requires the constructor for level `s`, and thus proving that it is associated to the security level `s`. The function also takes an authority data type for the escape hatch, an `IO` computation and returns an `IO` computation where the locks for the escape hatch is open. Before `certify` returns, the lock is closed. Declassifying information is only possible in the computation sent to `certify`, therefore the escape hatch should be used in that computation.

For the `certify` method to work, the constructors for security levels has to be hidden (using data abstraction) from modules that are not acting for that authority.

2.8.4 Differences with original implementation

Some changes were introduced to escape hatches. The difference between the implementation in [10] and our implementation is shown below.

```
-- Russo et al. 2008
type Hatch s s' a b = Sec s a -> IO (Maybe (Sec s' b))

-- Our Implementation
type Hatch s s' a b = Sec s a -> SecIO s' b
```

The difference lies in the return type, which was changed from `IO (Maybe (Sec s' b))` to `SecIO s' b`. The reason being that having a return type of `IO` implies that functions using the hatch will also have a return type of `IO`. Allowing a hatch to be used in an untrusted module, implies that the untrusted module has access to the `IO` monad, which poses no restrictions on the IO operations that can be performed. This is particularly problematic in potentially malicious code. For example, the `Login` file can just directly read the file `/etc/shadow` using the primitive Haskell function `readFile`, thus compromising confidentiality. The return type `SecIO s'` assures that side-effects are controlled and do not violate confidentiality. Furthermore, the inner `Maybe` in [10] is used to represent if a declassification is possible under some circumstances. The return value is `Nothing` when, for example, the `ntimes` combinator has already been applied the allowed times, or a flow lock is closed. Since the returned `Maybe` constructor is public, an attacker can inspect the value to determine if it is `Just` or `Nothing`. This means that extra care has to be taken when defining escape hatches, since it is important that declassification policies should not depend on secret values in order to avoid unintended leaks. Otherwise, it would be possible to reveal information about secrets by inspecting the return value when applying escape hatches. We removed the `Maybe` constructor from our implementation, and for simplicity, the program crashes if an escape hatch cannot be applied.

CHAPTER 3

Integrity as a Library

As briefly described in the Introduction (Chapter 1), integrity policies seeks to prevent destruction of data, accidentally or maliciously. A piece of data that has integrity is valid and has some meaning. Integrity can be compromised by, for example, human error when entering data, transmission errors, bugs or viruses. Integrity policies are as important as confidentiality when regarding the whole security aspect of a system, and thus it is natural to include them in a security library.

Integrity is hard to capture and context specific, however, three kind of integrity policies frequently required by programs are identified[9]. More precisely, we distinguish in this chapter access control, data invariants and information-flow policies. Each section will cover how to incorporate each of these kinds of policies in the library. The examples presented in this chapter build upon the password administrator described in Section 2.7. The password administrator example is divided into three modules. Each module is responsible for providing different functionalities. The modules are: *Login* (which performs user authentication), *Reset* (which changes users' passwords) and *Backup* (which creates a copy of the `passwd` and `shadow` files). These modules all benefit from integrity policies as a way of strengthening their security guarantees as shown in Table 3.1. The table shows the modules and the desired policies that are to be enforced for each of them.

3.1 Access Control

Access control policies governs who has access to a resource and in which way. With regards to integrity, this can protect resources from intentional or unintentional modification or destruction, by simply not having the proper

Module	Confidentiality policies	Integrity policies
`Login`	The only acceptable information leak comes from comparing users's input with user's stored password. Otherwise, no passwords must be leaked.	The files `passwd` and `shadow` cannot be written to.
`Reset`	No passwords must be leaked.	The file `passwd` can only be read, while the file `shadow` can be both read and written. Users' passwords must be *difficult* to guess by attackers.
`Backup`	Password must not be leaked unless they are copied and encrypted in the backup file.	The files `passwd` and `shadow` can only be read. Untrustworthy data must not weaken the encryption process. Chosen encryption methods must be strong.

Table 3.1: Confidentiality and integrity policies for the password administrator

permissions of performing certain tasks on given resources.

As an example, we have the policy "module `X` can only read file `F`". Access control should then be responsible that module `X` cannot write into file `F`. Access control policies are often enforced using a form of execution monitors. Access control mechanisms can be classified as mandatory, discretionary, and role-based[9]. Mandatory access control constraints the ability of *subjects* on *objects* in order to enforce confidentiality[26, 27]. These mechanisms are usually over restrictive. Discretionary access control allows users to create and set permissions of resources, and is commonly used by operating systems (Unix-like, Windows, etc). Lastly, role-based access control[28] is more suitable for civilian or governmental organizations where employees have different capabilities depending on their role in the organization. We decided to extend the library to include discretionary access control since it is widely used by operating systems.

3.1.1 Access Control Data Type

A new data type, `AC`, is introduced to restrict access to resources. Every resource can be wrapped by the access control data type, which indicates the allowed permissions on the given resource. An access controlled resource is then created using the following data type and method:

```
data AC p r
```

```
ac :: r -> AC p r
```

Type `AC p r` represents that resource `r` can only allow the operations described by permissions `p`. Permissions are encoded as singleton types. For example, the following singleton types represent reading and writing permissions:

```
data Write = MkWrite
data Read = MkRead
```

We extend the combinators responsible to handle side-effects in `SecIO` to work with access controlled objects. The signature for reading and writing into files is changed to:

```
readFileSecIO  :: AC Read (File s) -> SecIO s' (Sec s String)
writeFileSecIO :: AC Write (File s) -> String -> SecIO s ()
```

These methods are defined in a new set of modules dealing with access control. Their implementation is simply to assert that the right permission is given, and then reroute the file to `readFileSecIO` and `writeFileSecIO` in the `SecIO` module.

Observe that these methods can now be used to guarantee the policy regarding module X:

```
module Files (acFile) where
-- Trusted code, only exports acFile
file :: File H

acFile :: AC Read (File H)
acFile = ac file


module X where
-- Untrusted code
import Files

readFile :: SecIO s' (Sec H String)
readFile = readFileSecIO acFile
```

The function `readFile` reads the `acFile` file and returns the string that is read. Attempting to call the method `writeFileSecIO` on `acFile` will result in a type error.

There are situations when several permissions are needed for the same resource, for example reading and writing to the same file. One way to do

that is to check that a permission is in a list of capabilities. This can be done by using the class `Less` described in Section 2.1. For example, the next instances of `Less` check that some permissions are in a given list of capabilities.

```
instance Less (Read,Write) Read where
   less _ _ = ()

instance Less (Read,Write) Write where
   less _ _ = ()
```

Using the class `Less` allows for greater flexibility and can be used to create lattices for permissions, meaning that permissions can be extended to imply other permissions. For example, when given the write permission, the read permission can be implied by introducing:

```
instance Less Write Read where
   less _ _ = ()
```

The methods for reading and writing to files can now be rewritten as follows:

```
readFileSecIO  :: Less p Read =>
                  AC p (File s) -> SecIO s' (Sec s String)
writeFileSecIO :: Less p Write =>
                  AC p (File s) -> String -> SecIO s ()
```

For files with no need for access control policies, the singleton type `Any` is introduced:

```
data Any = MkAny

instance Less Any s where
   less _ _ = ()
```

Finally, the import restrictions, discussed in Section 2.6, is extended for untrusted modules to not allow importing the access control modules. These modules are:

- `AC.hs`, which includes the data type for the access control and a few methods to work with.
- `ACSecIO.hs`, which defines new methods for `readFileSecIO` and `writeFileSecIO` using access control.

These files should not be directly imported by untrusted modules, since they are not allowed to create access controlled resources. Instead, access controlled resources are created by in a trusted module and that module is imported (as shown with the module `Files` in the example earlier). When several different files are needed by an untrusted module, the responsibility can be placed upon the trusted main module calling the untrusted code to supply the correct files via parameters. Furthermore, the methods `readFileSecIO` and `writeFileSecIO` from `ACSecIO.hs` are included in `SecLib.hs`. The trusted users of the security library can customize `SecLib.hs` to either export the methods from `AcSecIO.hs` or `SecIO.hs`, depending if access control is desired.

3.1.2 Access Control in the Password Administration

The password administration example shown in Figure 2.8 can be extended using access control as follows:

```
getSpwdName :: Less p Rd =>
               AC p (File L) -> -- Passwd file
               AC p (File H) -> -- Shadow file
               Name -> SecIO L (Maybe (Sec H Spwd))
putSpwd     :: Less p Wrt =>
               AC p (File H) -> -- Shadow file
               Spwd -> SecIO H ()
```

The `passwd` and `shadow` files, passed as arguments, are now wrapped inside the `AC` data type. Function `getSpwdName` requires reading permissions for the `passwd` and `shadow` files. Similarly, `putSpwd` requires write permissions to the `shadow` file.

Reviewing the Table 3.1 shows that the modules `Login`, `Reset` and `Backup` benefit from access control policies. The `Reset` module should be able to write to the `shadow` file. This is done by creating an access control data type for the `shadow` file and assigning the `Read` and `Write` permissions to it. It is important that the created `AC` wrapper is only exported to the `Reset` module, and not to other modules. The modules `Login` and `Backup` should import the module that creates the read access for `passwd` and `shadow`. Different `AC` data types have to be created to give different permissions to the same resource in different modules.

3.2 Data Invariants

Data invariant integrity policies place constraints upon data, and certain properties that the data must fulfill in order to be meaningful and correct. Generally speaking, a computable predicate ϕ defines the quality of a piece

of data d. If $\phi(d)$ holds, then d has *good quality*. Property ϕ is a safety property, and there exists techniques to enforce it, e.g. execution monitors and software-based fault isolation. These traditional techniques generally suffer from overhead and scalability issues. As designers of a library, there are difficulties to monitor that ϕ holds at every computational step. Computations that a module `X` performs on a piece of data cannot be tracked unless calls to the library are made. As a consequence, verifying the computation at every step implies calling the library continuously and thus becoming computationally expensive. Instead, an alternative approach is to verify ϕ at programs' end points, i.e. inputs and outputs. By focusing on data invariants at end points, it is possible to capture some interesting integrity policies required in practice while keeping the overhead at reasonable levels. The library described in Section 2 provides access to end points for files through reading and writing operations. By extending the `File` data type, data invariants at end points can be enforced. We denote ϕ_i to predicates that are checked when inputs are performed. Similarly, ϕ_o denotes the predicate that is checked when outputs are performed. Both data invariants ϕ_i and ϕ_o are specified by trusted code. For simplicity, programs crash when an invariant is violated.

3.2.1 Data Invariant Implementation

One way of implementing a data invariant is as follows:

```
type DataInvariant d = (d -> IO Bool)
```

Data invariants are program specific, and the type `d -> IO Bool` allows to investigate programs' states when considering the quality of `d`. A data invariant that is not fulfilled will return `False`, which will cause the program to terminate before an input operation returns the result, or before an output operation is performed.

The data types for resources are extended to handle data invariant policies. The constructor for files (show in Figure 2.5) is extended as follows:

```
data File s = MkFile FilePath
              (DataInvariant String) -- input
              (DataInvariant String) -- output
```

The `DataInvariant`s represent ϕ_i and ϕ_o, respectively.

A new constructor is introduced to create files with data invariants as tautologies:

```
makeFile :: FilePath -> File s
makeFile fp = MkFile fp (return True) (return True)
```

For trusted programmers, two methods are introduced for setting up ϕ_i and ϕ_o in each file:

```
invariant_input  :: DataInvariant a -> File s -> File s
invariant_input i (MkFile f _ o) = (MkFile f i o)

invariant_output :: DataInvariant a -> File s -> File s
invariant_output o (MkFile f i _) = (MkFile f i o)
```

These functions are self-explaining and thus they are not described any further. Finally, functions that perform input and outputs (described in Section 2.5) are extended to verify data invariants:

```
readFileSecIO :: File s -> SecIO s' (Sec s String)
readFileSecIO (MkFile f i _) =
   MkSecIO $
   sec `fmap` (do a <- readFile f
                  b <- i a
                  if b then return a
                       else error "Data Invariant violated")

writeFileSecIO :: File s -> String -> SecIO s ()
writeFileSecIO (MkFile f _ o) s =
    MkSecIO $
    sec `fmap` (do b <- o a
                   if b then writeFile f s
                        else error "Data Invariant violated")
```

The string read from `readFileSecIO` is sent to ϕ_i which checks the quality of the data, and either returns `True` and a string is returned, or returns `False` and the execution is stopped.

To demonstrate the use of data invariants, assume a simple scenario where one file is only allowed to hold positive integers, and another file is only allowed to hold negative integers. One way of implementing this is as follows:

```
module FilesDI (positive, negative) where
-- Trusted module
import Files(file1, file2)

positive :: File H
positive = invariant_input positive
(invariant_output positive file1)
   where positive :: DataInvariant String
         positive a = return (read a > 0)
```

```
negative :: File H
negative = invariant_input negative
(invariant_output negative file2)
   where negative :: DataInvariant String
         negative a = return (read a < 0)


module Example where
-- Untrusted module

import FilesDI

copy :: SecIO H ()
copy = do a <- readFileSecIO positive
          writeFileSecIO negative (show (-(read a)))
```

The example shows how to create two simple files with data invariants. If the untrusted code attempts to write to the file `negative` without negating the value read from `positive`, an exception will be thrown, leaving the file `negative` unchanged.

3.2.2 Data Invariants in the Password Administrator

The password administrator can benefit from data invariant integrity policies in the module `Reset`, by placing constraints on the new passwords. A password must be difficult to guess, which can be formalized as the passwords need to be at least eight characters long, composed of both letters and numbers and, at least, one special character (e.g. ?, !, %). If these requirements are not met when changing passwords, the new password is rejected and no changes are made in the `shadow` file. The implementation of the data invariant for password checking can be done as follows.

```
-- Trusted code
enforce_Password :: DataInvariant String
enforce_Password str = return $ and
          [(length s >= 8 &&
           isJust (find (isAlpha) s) &&
           isJust (find (isDigit) s) &&
           isJust (find (isSpecial) s)) | (_,s) <- spwd]

where isSpecial = \x -> x > 'z' || x < '0'
        spwd = read str :: [(UID, Cypher)]
```

The `enforce_Password` data invariant expects a string as input. The function `writeFileSecIO` is implemented using the function `writeFile`,

which writes to a file as a single string with newlines. This has to be taken into consideration by the data invariant, which first converts the string to a list of pairs with user's ID and passwords. Afterwards, every password to be written to the `shadow` file is examined against the constraints. We assume that all passwords have high integrity (fulfill the constraints) before a user attempts to change her password, and thus if the new password does not meet the requirements, the invariant will fail and the program will crash with an error message. However, if the new password fulfills all requirements, the invariant succeeds and the new password can then be written to the `shadow` file. The `enforce_Password` data invariant is used for the `shadow` file as follows:

```
module Files ( reset_shadow ) where
-- Trusted module

shadow :: File H
shadow = makeFile "/etc/shadow"

reset_shadow :: File H
reset_shadow = invariant_output enforce_Password shadow
```

The resource `reset_shadow` is the file that is given to, and used by, the `Reset` module as shown:

```
module Reset where
-- Untrusted module

import SecLib
import Spwd
import SpwdData

reset :: UID ->
         File H ->  -- /etc/shadow with data invariant
         SecIO L ()
```

The function `reset` takes a UID and the `/etc/shadow` file with the data invariant. Then, it asks for a new password from the user and attempts to write it to the `shadow` file. For simplicity, if the new password does not fulfill the requirements of the data invariant, the program will crash.

3.3 Information-flow Policies

Information-flow integrity policies govern how untrustworthy data can flow inside programs. The library is extended to accommodate integrity levels.

Two new security labels are introduced: H_i and L_i, denoting high (trustworthy) and low (untrustworthy) integrity levels, respectively. Non-interference in this setting requires that low integrity data cannot flow into high integrity entities (meaning $L_i \not\sqsubseteq H_i$), but high integrity data can flow into low integrity ones ($H_i \sqsubseteq L_i$).

3.3.1 Information-flow Integrity Implementation

The integrity requirements $L_i \not\sqsubseteq H_i$ and $H_i \sqsubseteq L_i$ are dual to the ones for confidentiality ($L \sqsubseteq H$ and $H \not\sqsubseteq L$, as discussed in Section 2.2). Thanks to this duality, the library abstractions for confidentiality can be used to guarantee non-interference regarding integrity. To illustrate that, the lattice involving integrity levels is easily incorporate into the library as follows:

```
data Hi -- High Integrity
data Li -- Low Integrity

instance Less Hi Hi where
   less _ _ = ()

instance Less Hi Li where
   less _ _ = ()

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

It is then possible to present a non-interference example involving integrity security levels:

```
x :: Sec Hi Int
y :: Sec Li Int

z :: Sec Li Int
z = do x' <- up x
       y' <- y
       return (x' + y')
```

The example shows that a high integrity value (`x`) can be used in computations involving low integrity data (`z`) by applying the `up` combinator.

In contrast, the following example violates non-interference and thus it does not type check:

```
x :: Sec Hi Int
y :: Sec Li Int
```

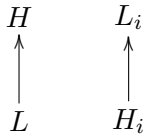


Figure 3.1: The allowed flows for confidentiality and integrity.

```
w :: Sec Hi Int
w = do x' <- x
       y' <- y
       return (x' + y')
```

In this example, a low integrity value (`y`) is used in a computation involving high integrity data (`w`). A type error will be generated since the non-interference integrity policy is violated.

To handle information-flow for confidentiality and integrity policies at the same time, a new instance for `Less` is created to accept tuples of security levels[29]:

```
instance (Less c1 c2, Less i1 i2) =>
                Less (c1,i1) (c2,i2) where
         less _ _ = ()
```

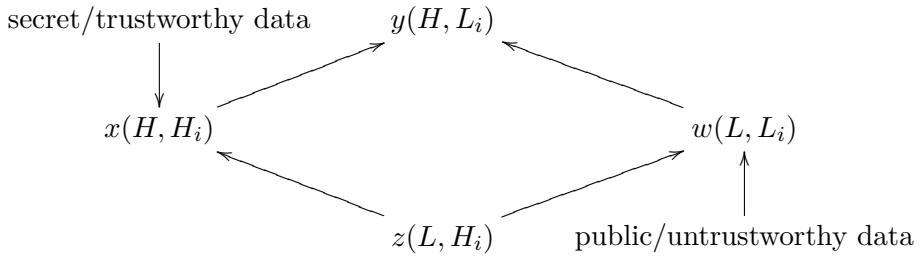
The tuple corresponds to the confidentiality and integrity levels of protected data, the first element of the tuple is related to confidentiality, while the second one is related to integrity levels. For example, it is possible to define values with different confidentiality and integrity levels:

```
x :: Sec (H, Hi) Int

y :: Sec (H, Li) Int

z :: Sec (L, Hi) Int

w :: Sec (L, Li) Int
```

The values `y` and `w` cannot affect `x` or `z`, since the non-interference integrity policy would be violated ($L_i \not\sqsubseteq H_i$). Similarly, the values `x` and `y` cannot be transformed into `z` or `w`, since the non-interference confidentiality policy would be violated ($H \not\sqsubseteq L$).

To better show these relationships, consider the two lattices in Figure 3.1, describing the non-interference policies for confidentiality and integrity,

respectively. This diagram captures the fact that data with the security level at the bottom can flow into the security level at the top, but not the other way around. The following diagram shows how the values `x`, `y`, `z` and `w` are encoded in the *product lattice* by combining confidentiality and integrity:



The above diagram shows the way security labels can be altered. New values from untrusted code enter the lattice from the point specified as `public/untrustworthy values`.

Two new instances for the class `Attacker s` (showed in Figure 2.3) are defined to address integrity:

```
public :: Sec Li a -> a
public :: Sec (L, Li) a -> a
```

Applications might use the confidentiality and integrity lattices separately, and then at some point need to compare values using different security lattices. To address this issue, new combinators are added to transform single security levels to a security level involving confidentiality and integrity. To this end, the class `Attacker s` is used to transform a value in a security lattice into a value of the product lattice, as follows.

```
addIntegrity :: Attacker s => Sec s' a -> Sec (s',s) a
addConfidentiality :: Attacker s => Sec s' a -> Sec (s,s') a
```

As an example of how to use these functions, we present the following code:

```
-- Untrusted code
value :: Sec L Int

newValue :: Sec (L,Li) Int
newValue = addIntegrity value
```

The variable `value` is transformed from `L` to `(L,Li)` since there is an instance for `Attacker Li`. However, there is also an instance for `Attacker (L,Li)`, which allows creation of security levels with the type `(L,(L,Li))`. Levels of this type make no sense considering the lattices described previously, and two new type classes are introduced to address this issue:

```
addIntegrity :: (Attacker s, Integrity s) =>
                Sec s' a -> Sec (s',s) a
addConfidentiality :: (Attacker s, Confidentiality s) =>
                      Sec s' a -> Sec (s,s') a
```

Figure 3.2: Expanding the single security levels.

```
class Confidentiality s where
   confidentiality s -> ()

instance Confidentiality H where
   confidentiality _ = ()

instance Confidentiality L where
   confidentiality _ = ()

class Integrity s where
   integrity s -> ()

instance Integrity Hi where
   integrity _ = ()

instance Integrity Li where
   integrity _ = ()
```

The signatures for `addIntegrity` and `addConfidentiality` are slightly changed to use the above type classes, as shown in Figure 3.2. Similar methods are included for `SecIO`. The reason is, if confidentiality or integrity was not considered during a computation, we allow for public levels to be added later on when needed.

3.3.2 Endorsement

The need for declassifying data was presented in Section 2.8. This allows for some information to be intentionally released, which essentially lowers the security level of data, from higher to lower. This is the mechanism which creates a controlled flow for that direction the non-interference policy restricts, as shown in Figure 3.1. The same can be applied for integrity, representing that untrustworthy data becomes trustworthy, meaning that data flows from low integrity level into a higher integrity one. In the Figure 3.1, this implies that a data with level L_i is turned into H_i. This action is called *Endorsement*.

To illustrate how endorsement works, imagine a shell application. The user inputs a command in the form of a string, which has low integrity (since the user cannot be trusted). This command can contain malicious code and affect the integrity of the computer executing it (which has high integrity). Endorsement can be used to verify that user input is non-malicious. A non-malicious string is then endorsed as high integrity data when it does not contain malicious shell-commands.

The escape hatches and combinators (described in Section 2.8) used for declassification can be used for endorsement.

3.3.3 Information-flow Integrity in the Password Administrator

The module `Backup` will benefit from using information-flow integrity policies. The users of the module are allowed to choose encryption algorithm and key size by supplying a pair of strings. The reason for this is to accommodate different needs depending on the platform and application where the password administrator is used. Since `Backup` encrypts the information stored in the `/etc/shadow` file, it is important that the encryption is strong enough, so the cypher cannot easily be broken. The notion of "strong enough" varies depending on the computational power of the platform where the module is run. For example, on a desktop PC the Serpent[30] algorithm with a key size of 256 bits is quite strong, while on a mobile phone another algorithm and smaller key size could be used instead (AES[31] with a key size of 128 for example).

Therefore, the choice of encryption algorithm and key size has to be verified by the module. This verification is done by endorsing the pair of strings provided by the user in order to select the encryption method and key size. The endorsement function takes the pair of strings and assesses that the encryption algorithm and key size are allowed as well as that the key size is usable with the chosen algorithm. If these conditions are not met, the endorsement fails and produces an error. However, if they are satisfied, the endorsement function labels the pair as having high integrity, and returns it. The returned pair has security level `(L,Hi)`, which is required by the encryption function.

Some types are introduced to clarify the code:

```
-- Trusted code
newtype Key a = MkKey { unkey :: a }
type Content = String -- Unencrypted content
type Cypher  = String -- Encrypted content
type Method = (String, String) -- Encryption method,
                               -- first string is algorithm,
                               -- second is key size
```

```
type Usernames = String -- Content read from /etc/passwd
type Passwords = String -- Content read from /etc/shadow
```

Furthermore, two new kind of escape hatches are introduced for encrypting and decrypting strings, respectively, as follows:

```
-- Trusted code
encrypt_hatch :: Less (L,Hi) (s,Hi) =>
                 Method ->  -- chosen encryption method
                 Hatch (s,Hi) (L,Hi) (Word128,Content) Cypher
decrypt_hatch :: Less (s,Hi) (H,Hi) =>
                 Method ->  -- chosen encryption method
                 Cypher ->
                 Hatch (H,Hi) (s,Hi) Word128 Content
```

The encryption hatch takes an encryption method and creates a hatch which returns the encrypted cypher text when given a pair of a key and a plain text. The decryption hatch takes an encryption method, a cypher text and creates a hatch which returns the plain text when given a key. Both hatches expect the encryption method to be valid. The trusted module defining the encryption resources for the `Backup` module is defined as follows:

```
module Encryption
   (valid_enc, encrypt, decrypt) where
-- Trusted code
valid_enc :: Less s s => Hatch (s,Li) (s,Hi) Method Method

encrypt :: Sec (L,Hi) Method ->
           Sec (L,Hi) Usernames ->
           Sec (H,Hi) Passwords ->
           SecIO (L,Hi) Cypher

decrypt :: Method -> Cypher ->
           SecIO (L,Hi) (Usernames, Sec (H,Hi) Passwords)
```

The `Encryption` module defines the `valid_enc`, `encrypt` and `decrypt` methods to be used by the `Backup` module. The escape hatch `valid_enc` endorses an encryption method from low integrity to high integrity, while disregarding confidentiality. For simplicity, the program is terminated if endorsement fails. If it succeeds, then the returned encryption method with security level `(L,Hi)` can be used for encrypting and decrypting. The method `encrypt` takes a valid encryption method (high integrity), the user names and the passwords, and uses the `encrypt_hatch` to encrypt and declassify the cypher text. The method `decrypt` takes an encryption method without validating it first. This is because attempting to decrypt a cypher text

with the wrong algorithm or key size will not produce the expected results. Attempting to restore with a different encryption method will result in a program error. Furthermore, `decrypt` also takes the cypher text, which is read from a public file, and returns a pair representing the user names and the passwords.

Lastly, the module `Backup` can be introduced using the methods described above:

```
module Backup where
-- Non-malicious code

backup :: Less p Wrt =>
          AC p (File L) ->  -- destination file to backup to
          Sec (L,Hi) Usernames ->
          Sec (H,Hi) Passwords ->
          Method ->
          SecIO (L,Hi) ()

restore :: Less p Rd =>
           AC p (File L) -> -- source file to restore from
           Method ->
           SecIO (L,Hi) (Usernames, Sec (H,Hi) Passwords)
```

The module `Backup` has two methods, `backup` and `restore`. The method `backup` takes the information stored in `/etc/passwd` and `/etc/shadow` and backs up the information to a public file (regardless of integrity level, due to it being a new file) using the specified encryption method. The method `restore` reads the data from the specified file (regardless of integrity level, due to not yet being able to assert the integrity) and decrypts it with the chosen encryption method. When restoring, the chosen encryption method has to be the same that was used for backing up.

3.4 Completing the Case Study

Now when all the underlying mechanism for the password administrator has been explained, this section will describe the main file that ties all of it together and provides a secure API for the different functionalities present in the password administrator.

```
module SecureAPI
  ( s_login, s_reset, s_backup, s_restore )
where
-- Trusted module
```

```
import AC ( ac )
import SecIO ( run, makeFile )
import SecLib

import Files

-- Login
import Login
import Policies

-- Reset
import Reset

-- Backup & Restore
import Encryption ( Method )
import Backup

s_login :: IO (Maybe UID)
s_login =
    do match <- declassification
       uid   <- run $ login match login_passwd login_shadow
       return (public uid)


s_reset :: IO ()
s_reset =
    do (Just uid) <- s_login
       run $ reset uid reset_shadow
       return ()


s_backup :: FilePath -> Method -> IO ()
s_backup f m =
   do f' <- run (newFileSecIO f)
    :: IO (Sec L (AC Wrt (File L)))
      run (do pass <- readFileSecIO backup_passwd
              shad <- readFileSecIO backup_shadow
              backup (public f') pass shad m)
      return ()

s_restore :: FilePath -> Method -> IO ()
s_restore f m =
    do f' <- return (ac $ makeFile f :: AC Rd (File L))
       run (do (names,pass) <- restore f' m
```

```
            writeFileSecIO restore_passwd names
            plug (do p <- value pass
                     writeFileSecIO restore_shadow p))
    return ()
```

All these functions perform the initialization required for the submodules. The function `s_login` starts the escape hatch and sends it, along with the corresponding files with the right permissions, to the `Login` module. The `Login` module then asks the user for name and password, and attempts to authenticate her. If `s_login` returns `Nothing`, it indicates that the authentication failed, otherwise the user's ID will be returned. The UID can be made public since it is readily available in `/etc/passwd`. The function `s_reset` first calls `s_login` to authenticate a user and expects a UID. It is then passed to the `Reset` module with the corresponding file for that module. Note that the file `reset_shadow` has a data invariant associated with it, which requires that the password is strong. Function `s_backup` takes a file path and an encryption method, and creates a new file specified by the file path. Then the `/etc/passwd` and `/etc/shadow` files are read, and the result passed to the `Backup` module. Similarly, function `s_restore` takes a file path and a method, passing these to the `Backup` module (the `restore` function) and writing the result into `/etc/passwd` and `/etc/shadow`.

A module diagram with import relationships for the Password Administrator is shown in Figure 3.3. An arrow from a module `X` to a module `Y` implies that module `Y` imports `X`.

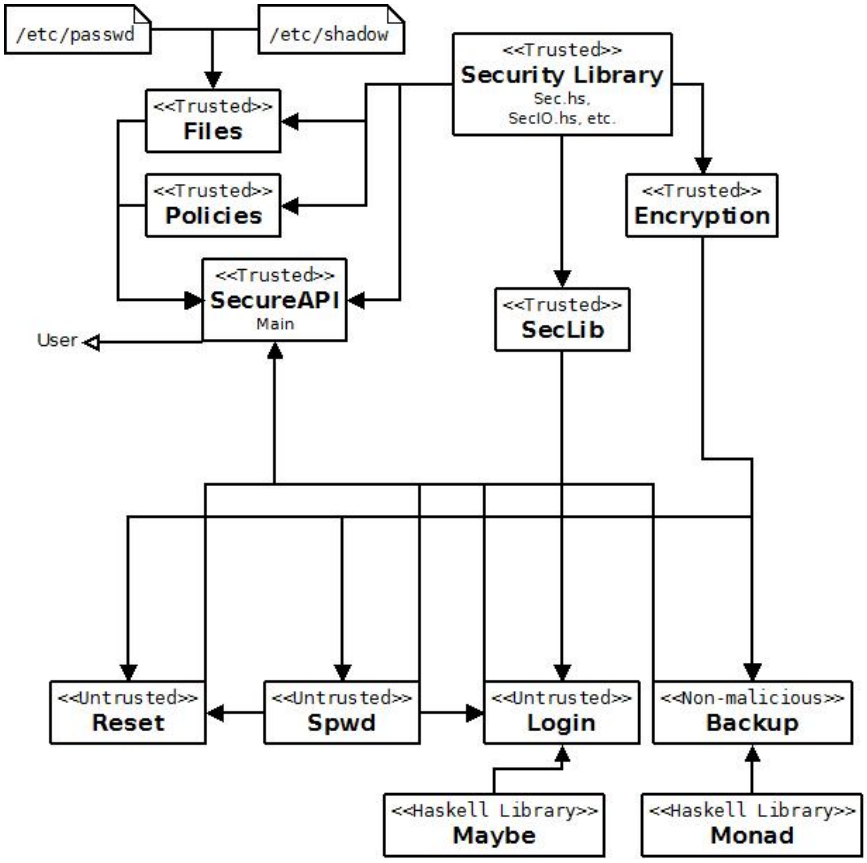


Figure 3.3: Import relationship diagram for the Password Administrator

CHAPTER 4

Discussions

In this chapter the restrictions and design choices are discussed for the integrity part of the library.

4.1 Access Control

4.1.1 Implementation

Access control is implemented using a simple wrapper data type and singleton data types representing permissions, due to the fact that it is lightweight and can be very flexible using an already implemented mechanism (the security lattice).

4.1.2 Permissions

The permissions for access control can be interpreted as sets and not lists, meaning that the following two constructs are equivalent:

$$(Read, Write) \equiv (Write, Read)$$

It is possible to express that equivalence using the security lattice by introducing the instance `Less (Read,Write) (Write,Read)`. However, considering sets of permissions, rather than lists, requires to introduce, at least, $n!$ instances of `Less`, where n is the different numbers of permissions.

Permissions could also be encoded using type classes, but singleton data types are used instead. The reason for this is that they are more elementary, and then effectively making them simpler.

Another approach of enforcing permissions is by data abstraction, which means that only the corresponding read or write functions are exported

to the untrusted module. This would however mean that a module would be separated, some times unnaturally, when using several read and write operations on different files with different permissions. We want to avoid these kind of restrictions that are forced upon the programmers.

4.2 Information-flow Integrity Policies

Currently, non-interference is enforced between high and low integrity levels, but no guarantees can be placed on the integrity of data protected at high integrity levels in potentially malicious code. Potentially malicious sources can create and alter values at high integrity levels. An example of this is shown below.

```
create = return 10 :: Sec Hi Int

alter :: Sec Hi Int -> Sec Hi Int
alter sec = (\s -> -100) `fmap` sec
```

As the example shows, untrusted sources can create values with high integrity level by using `return`, and alter them by using `fmap`. One possible way to aid against this problem could be considering information-flow integrity policies and data invariants. This work only considers information-flow integrity policies in non-malicious code.

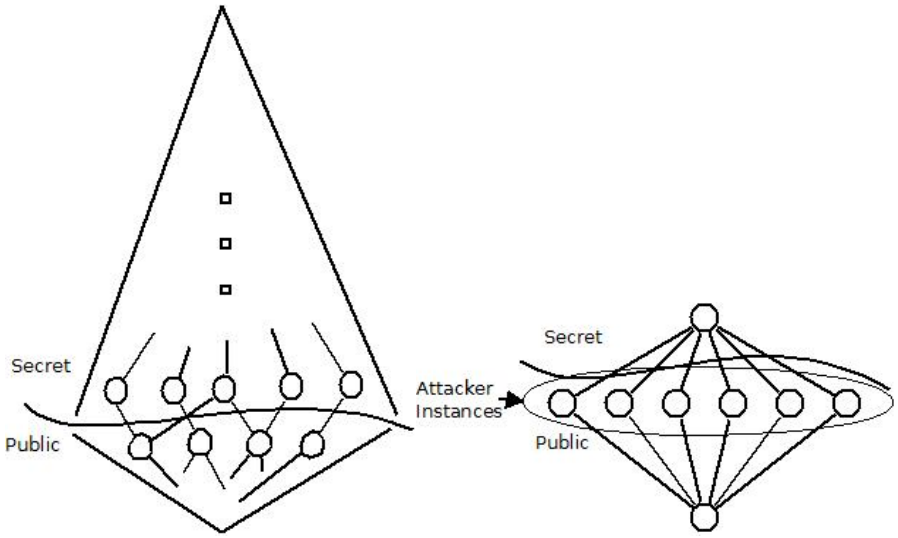
4.3 Public Information

The method `public` (introduced in Figure 2.3) underwent some change during our work. Previously, `public` belonged to the `Sec` module, and was defined as follows:

```
public :: Sec s a -> s -> a
```

Given a security monad, and a key representing the constructor for the current security level, then the monad is "opened" and the value published. This previously worked due to the fact that only observable security levels had their constructor exported to the untrusted modules. In the two-point lattice, `SecLib` in [10] exports the full constructor for `L`, but only the type for `H`. In our implementation, `public` uses the `Attacker s` class and has the type `Sec s a -> a`, since it is possible to export all the constructors for the security levels without jeopardizing security.

To illustrate the difference, consider the two examples in Figure 4.1. The Figure 4.1a shows a security lattice with several public levels (below the line) and several secret levels (above the line). There are four public levels bordering to the secret levels, and these are the ones that are required



(a) Many secret levels (b) One secret level

Figure 4.1: Different security lattice scenarios

to be fully exported or instantiated with `Attacker s`. For the public levels further down in the lattice, the `up` combinator can be used before making the information public. The implementation in [10] requires that all the secret levels do not have their constructor exported. In Figure 4.1b, there are six public levels bordering to a secret one. Similarly, the implementation in [10] requires that the constructor of the secret level not be exported, while all the other seven public levels need to be. However, our implementation requires six instances of `Attacker s`.

The approach we settled on is using the `Attacker s` class, since this version makes it a bit clearer what is actually publicly available, and it is useful when defining the functions showed in Figure 3.2 (for expanding single security levels to both confidentiality and integrity).

CHAPTER 5

Conclusions

The aim of this work was to design a library that provides integrity and confidentiality policies, which has been achieved. The integrity policies considered are access control, data invariants and information-flow integrity policies. Access control is implemented using a variation of access control list (ACL). Data invariants are enforced at program end points by inspecting the property when providing input and output of data. For non-malicious code, information-flow policies are incorporated by extending the security lattice with integrity levels, and thus offering the code to work with untrusted and trusted data, without the risk of mixing them. A password administrator that incorporates confidentiality and the mentioned integrity policies is presented in this work. Parts of the implementation was described and detailed to show how integrity policies can be incorporated to provide a richer set of security policies. To the best of our knowledge, this is currently the only library that incorporates both information-flow policies for confidentiality and integrity policies.

The integrity part of the library is designed with several aspects in mind (beside security), to increase its usefulness. The entire library is small (less than 400 lines of code) and includes only a few modules, making it lightweight, and thus increasing its portability. Moreover, the library does not introduce computational expensive functions. The parts of a program using the security of the library should have small overhead, while the parts not considering security have none at all. The library includes several security parts (e.g. confidentiality levels, integrity levels, access control, data invariants, declassification, endorsement, etc), and these can be combined or used separately. All these aspects give a good reason for using the security library.

References

[1] A. C. Myers, *JFlow: Practical mostly-static information flow control*. ACM Symp. on Principles of Programming Languages, 1999.

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

[3] S. Chong, K. Vikram, A. C. Myers, *Jif: Enforcing confidentiality and integrity in web applications*. In Proc. 16th USENIX Security, 2007.

[4] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, X. Zheng, *Secure web applications via automatic partitioning*. Proc. of ACM Symposium on Operating Systems Principles, ACM, 2007.

[5] F. Pettier, V. Simonet, *Information flow inference for ML*. Proc. ACM Symp. on Principles of Programming Languages, 2002.

[6] V. Simonet, *The Flow Caml system*. Software release, `http://cristal.inria.fr/ simonet/soft/flowcaml/`, 2003.

[7] John Barnes, *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

[8] John Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[9] Peng Li, Yun Mao, Steve Zdancewic, *Information Integrity Policies*. In Proc. of the Workshop on Formal Aspects in Security & Trust (FAST), 2003.

[10] Alejandro Russo, Koen Claessen, John Hughes, *A Library for Light-Weight Information-Flow Security in Haskell*. Chalmers University of Technology, Gothenburg, Sweden, 2008.

[11] Dorothy E. Denning, Peter J. Denning, *Certification of programs for secure information flow*. Comm. of the ACM, 1977.

[12] Philip Wadler, *Comprehending Monads*. Proc. of the 1990 ACM Conference on LISP and Functional Programming, 1990.

[13] Philip Wadler, *The Essence of Functional Programming*. Conference Records of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, 1992.

[14] Michael Barr, Charles Wells, *Category Theory for Computing Science*. Prentice Hall, 1990.

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

[16] A. Narayanan, 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, 2005.

[17] A. Sabelfeld, 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, 2004.

[18] A. Sabelfeld, D. Sands, *Dimensions and principles of declassification*. In CSFW '05: Proc. of the 18th IEEE Computer Security Foundations Workshop, pages 255-269, IEEE Computer Society, 2005.

[19] A. Askarov, A. Sabelfeld, *Localized delimited release: combining the what and where dimensions of information release*. In PLAS '07: Proc. of the 2007 workshop on Programming languages and analysis for security, pages 53-60, New York, 2007.

[20] H. Mantel, 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.

[21] A. C. Myers, B. Liskov, *A decentralized model for information flow control*. In Proc. ACM Symp, on Operating System Principles, pages 129-142, 1997.

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

[23] A. C. Myers, B. Liskov, *Protecting privacy using the decentralized label model*. ACM Transactions on Software Engineering and Methodology, pages 410-442, 2000.

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

[25] N. Broberg, 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 180-194, Springer, 2006.

[26] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, Robert Morris, *Information Flow Control for Standard OS Abstractions*. In Proc. of the 21st Symposium on Operating Systems Principles, 2007.

[27] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, David Mazières, *Making Information Flow Explicit in HiStar*. In Proc. of the 7th Symposium on Operating Systems Design and Implementation, 2006.

[28] D. Ferraiolo, R. Kuhn, *Role-based access control*. In 15th NIST-NCSC National Computer Security Conference, 1992.

[29] Steve Zdancewic, Andrew C. Myers, *Robust Declassification*. In Proc. of the 2001 IEEE Computer Security Foundations Workshop, pages 15-23, 2001.

[30] Ross Anderson, Eli Biham, Lars Knudsen, *The Case for Serpent*. Third AES Candidate Conference, Available at, `http://www.cl.cam.ac.uk/ rja14/Papers/serpentcase.pdf`, 2000.

[31] Joan Daemen, Vincent Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

List of Figures

List of Tables

APPENDIX A

Library Code

A.1 Sec.hs

```
-- | Provides security for pure computations (trusted).
module Sec where

import Lattice

-- Sec

-- | This type represents values of type a
-- | at confidentility level s.
newtype Sec s a = MkSec a


instance Functor (Sec s) where
  h `fmap` (MkSec x) = MkSec (h x)


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

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

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

```
-- | Lift information into higher positions
-- | on the security lattice.
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'

-- | Break the abstraction provide by 'Sec'.
-- | It is used only in trusted code!
reveal :: Sec s a -> a
reveal (MkSec a) = a

-- | Internal function, not exported.
-- | For type-checking purposes.
unSecType :: Sec s a -> s
unSecType _ = undefined
```

A.2 Lattice.hs

```
-- | Encodes the security lattice used
-- | in the library (trusted).
module Lattice where

-- Security types

class Confidentiality s where
        conf :: s -> ()

-- | Data type representing the security level
-- | associated to public information.
data L = L

-- | Data type representing the security level
-- | associated to secret information.
data H = H

instance Confidentiality L where
        conf _ = ()

instance Confidentiality H where
```

```
        conf _ = ()



class Integrity s where
        int :: s -> ()

data Li = Li

data Hi = Hi

instance Integrity Li where
        int _ = ()

instance Integrity Hi where
        int _ = ()


-- Less
class Less s s' where
  -- | This method determines that information
  -- | at security level s can be pushed up to
  -- | security level s' (not exported to untrusted code).
  less :: s -> s' -> ()


-- Security lattice for confidentiality
instance Less L L where
  less _ _ = ()


instance Less H H where
  less _ _ = ()

instance Less L H where
  less _ _ = ()


-- Security lattice for integrity
instance Less Hi Hi where
  less _ _ = ()

instance Less Hi Li where
  less _ _ = ()
```

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


-- Product lattice for confidentiality and integrity
instance (Less c1 c2, Less i1 i2) =>
             Less (c1,i1) (c2,i2) where
         less _ _ = ()



-- Access Control
data Wrt = MkWrt
data Rd  = MkRd
data Any = MkAny

instance Less Rd Rd where
   less _ _ = ()

instance Less Wrt Wrt where
   less _ _ = ()

instance Less Wrt Rd where
   less _ _ = ()

instance Less (Rd, Wrt) (Rd,Wrt) where
   less _ _ = ()

instance Less (Rd, Wrt) Rd where
   less _ _ = ()

instance Less (Rd, Wrt) Wrt where
   less _ _ = ()


instance Less Any s where
   less _ _ = ()
```

A.3 SecIO.hs

```
-- | Provide security for computations with side-effects.
module SecIO where
```

```
import Lattice
import Sec
import Attacker

import Control.Monad
import System.Directory
import Data.IORef
import Network.Socket

-- Monad

{-| Secure side-effect computations.
    These computations perform side-effects
    at security level s (or above)
    and return a value of type a.
-}
newtype SecIO s a = MkSecIO (IO (Sec s a))

instance Functor (SecIO s) where
  h `fmap` (MkSecIO io) = MkSecIO ( do sec <- io
                                       return (h `fmap` sec) )


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

-- | Lift a pure confidential value into
-- | a secure side-effect computation.
value :: Sec s a -> SecIO s a
value sa = MkSecIO (return sa)


-- | Execution of secure computations.
run :: SecIO s a -> IO (Sec s a)
run (MkSecIO m) = m
```

```
-- | Safetly downgrade the security level that restrict
-- | the side-effects performed by the computations.
plug :: Less sl sh => SecIO sh a -> SecIO sl (Sec sh a)
plug secio_sh@(MkSecIO m) =
        less sl sh `seq` secio_sl
        where
        (secio_sl) = MkSecIO (do sha <- m
                                 return (sec sha))
        sl = unSecIOType secio_sl
        sh = unSecIOType secio_sh

-- | Internal function, not exported.
-- | For type-checking purposes.
unSecIOType :: SecIO s a -> s
unSecIOType _ = undefined


-- | Extending the security level of a single security type
addIntegritySecIO :: (Attacker s, Integrity s) =>
                      SecIO s' a -> SecIO (s',s) a
addIntegritySecIO (MkSecIO a) =
        MkSecIO (addIntegrity `fmap` a)

addConfidentialitySecIO :: (Attacker s, Confidentiality s) =>
                           SecIO s' a -> SecIO (s,s') a
addConfidentialitySecIO (MkSecIO a) =
        MkSecIO (addConfidentiality `fmap` a)


{-|
    Represent secure locations.
    Data type that is internally used to easily introduce
    new side-effects into SecIO.
    Type t is the raw type needed to perform side-effects.
    For instance, t is 'FilePath' for file, 'IORef' a for
    writing and reading references, etc.
    Type s is the confidentiality level of the location.
    Type a is the kind of values written and read form t.
-}

type DataInvariant a = (a -> IO Bool)
data Loc t s a b = MkLoc t (DataInvariant a) (DataInvariant a)
```

```
invariant_input :: DataInvariant a ->
                   Loc t s a b -> Loc t s a b
invariant_input di (MkLoc t _ o) = MkLoc t di o

invariant_output :: DataInvariant a ->
                    Loc t s a b -> Loc t s a b
invariant_output di (MkLoc t i _) = MkLoc t i di


{-| Introduce unsecure read and write operations
    for side-effects of type t
    that stores and read values of type a.
-}
class UnsecureRW t a b | t -> a b where
   unsec_write :: t -> a -> IO b
   unsec_read  :: t -> IO a

{-| Introduce safe read and write operations for locations
    of type 'Loc' t s a. This class acts as a wrapper for
    the type class 'UnsecureRW'.
-}
class UnsecureRW t a b => SecureRW t s a b where
   {-| Secure read operation when the fact of just
       reading might be visible for the attacker from
       inside of the program. For example, reading from
       a channel in a network communication changes the
       buffer pointer for that channel,
       which can be exploited to leak information.
   -}
   effectful_read  :: Loc t s a b -> SecIO s a
   {-| Secure read operation when the fact of just
       reading is not visible by the attacker from
       inside of the program. For example, reading
       from references does not produce any observable effect.
   -}
   effectless_read :: Loc t s a b -> SecIO s' (Sec s a)
   -- | Secure write operation.
   effectful_write :: Loc t s a b -> a -> SecIO s b

   -- Create read and write resource
   create_resource :: t -> Loc t s a b


instance UnsecureRW t a b => SecureRW t s a b where
```

```
    effectless_read (MkLoc loc i _)   =
        MkSecIO $ (sec.sec) `fmap`
                (do a <- unsec_read loc
                    b <- i a
                    if b then return a else diError)
        where diError = error "Data invariant violated."

    effectful_read  (MkLoc loc i _)   =
        MkSecIO $ (sec `fmap`
                (do a <- unsec_read loc
                    b <- i a
                    if b then return a else diError))
        where diError = error "Data invariant violated."

    effectful_write (MkLoc loc _ o) a =
        MkSecIO $ (sec `fmap`
                (do b <- o a
                    if b then unsec_write loc a else diError))
        where diError = error "Data invariant violated."

    create_resource t =
        MkLoc t (\_ -> return True) (\_ -> return True)


-- | Locations that represent files.
type File s = Loc FilePath s String ()

instance UnsecureRW FilePath String () where
   unsec_read file      = readFile file
   unsec_write file str = writeFile file str


-- | Secure reading from a file.
readFileSecIO  :: File s -> SecIO s' (Sec s String)
readFileSecIO  = effectless_read

-- | Secure writing to a file.
writeFileSecIO :: File s -> String -> SecIO s ()
writeFileSecIO = effectful_write

makeFile :: FilePath -> File s
makeFile = create_resource

-- Creation of low files
```

```
newFileSecIO :: Attacker s => FilePath -> SecIO s (File s)
newFileSecIO f = MkSecIO $
     do b <- doesFileExist f
        case b of
                True  -> error "File alredy exists."
                False -> return $ sec (create_resource f)

-- | Derived operations for reading from files (legacy code).
s_read :: Less s' s => File s' -> SecIO s String
s_read file =
   do ss <- readFileSecIO file
      value (up ss)

-- | Derived operations for writing into files (legacy code).
s_write :: Less s s' =>
           File s' -> String -> SecIO s (Sec s' ())
s_write file s = plug (writeFileSecIO file s)


-- | Locations that represent references.
type Ref s a = Loc (IORef a) s a ()

instance UnsecureRW (IORef a) a () where
   unsec_read  ref    = readIORef ref
   unsec_write ref a  = writeIORef ref a


-- | Secure reading from a reference.
readRefSecIO  :: Ref s a -> SecIO s' (Sec s a)
readRefSecIO  = effectless_read

-- | Secure writing to a reference.
writeRefSecIO :: Ref s a -> a -> SecIO s ()
writeRefSecIO = effectful_write


{-| Secure creation of a reference.
    We assume that the attacker has no manner to observe
    the side-effect of creating a reference from inside of
    the program, for example, by inspecting the free memory.
    At the moment, there are no such functions inside
    of 'SecIO'. Nevertheless, if there is a consideration of
    include, for instance, a function that returns the
    free memory in the program, then the function type of
```

```
    'newIORefSecIO' needs to be changed.
-}
newIORefSecIO :: a -> SecIO s' (Ref s a)
newIORefSecIO a =
        MkSecIO ( (sec.create_resource) `fmap` newIORef a )


{-|  Location that represents the screen-keyword.
     Here, we can choose between
     a) The attacker is on the screen-keyword,
        which implies that when taking input
        from the screen has an effect --
        the attacker can detect the input. Therefore,
        we need to implement taking input from
        the keyword using 'effectful_read'.
     b) The attacker is not on the screen-keyword.
        In this case, we implement reading using
        'effectless_read'. We choose option a) as a model.
-}
type Screen = Loc () L String ()


instance UnsecureRW () String () where
   unsec_read  _    = getLine
   unsec_write _ a  = putStr a


-- | Secure input from the keyword.
getLineSecIO  :: Screen -> SecIO L String
getLineSecIO  = effectful_read

-- | Secure output to the screen.
putStrSecIO :: Screen -> String -> SecIO L ()
putStrSecIO = effectful_write

makeScreen :: Screen
makeScreen = create_resource ()


{-------------------------------------
 --- Network operations
-------------------------------------}

type SecSocket s   = Loc (Socket,Int) s String Int
```

```
{- Data types needed to create a connection -}
{- Which connection is high or low is defined in the
   trusted code!
-}
data SecSockAddr s = MkSockAddr SockAddr


inet_addrSecIO :: String -> SecIO s HostAddress
inet_addrSecIO s = MkSecIO $ (return `fmap` (inet_addr s))


socketSecIO :: Family -> SocketType -> ProtocolNumber ->
               SecIO s' (SecSocket s)
socketSecIO f s p =
        MkSecIO ( (sec.(\s -> create_resource (s,0)))
                  `fmap` socket f s p)


-- it might be possible to return SecIO s' (), if it is the
-- case that sIsBoundSecIO is the only observer if a socket
-- is bound or not.
bindSocketSecIO :: SecSocket s -> SecSockAddr s -> SecIO s ()
bindSocketSecIO (MkLoc (s,_) i o) (MkSockAddr address) =
                MkSecIO (sec `fmap` bindSocket s address)


sIsBoundSecIO :: SecSocket s -> SecIO s' (Sec s Bool)
sIsBoundSecIO (MkLoc (s,_) i o) =
        MkSecIO (sec `fmap` (sec `fmap` sIsBound s))


acceptSecIO :: SecSocket s ->
               SecIO s (SecSocket s, SecSockAddr s)
acceptSecIO (MkLoc (s,_) i o) = MkSecIO (sec `fmap` m)
        where m = do (sock, addr) <- accept s
                     return (MkLoc (sock,0) i o,
                             MkSockAddr addr)


instance UnsecureRW (Socket,Int) String Int where
  unsec_read  (sock,n)     = recv sock n
  unsec_write (sock,_) str = send sock str
```

```
recvSecIO :: SecSocket s -> Int -> SecIO s String
recvSecIO (MkLoc (s,_) i o) n =
        effectful_read (MkLoc (s,n) i o)


sendSecIO :: SecSocket s -> String -> SecIO s Int
sendSecIO = effectful_write
```

A.4 Attacker.hs

```
module Attacker where

import Sec
import Lattice

-- | The class Attacker s represents an attackers
-- | observational power
class Attacker s where
        public :: Sec s a -> a

instance Attacker L where
        public (MkSec a) = a

instance Attacker Li where
        public (MkSec a) = a

instance Attacker (L,Li) where
        public (MkSec a) = a

-- | Adds a public integrity level
addIntegrity :: (Attacker s, Integrity s) =>
                Sec s' a -> Sec (s',s) a
addIntegrity (MkSec a) = return a

-- | Adds a public confidentiality level
addConfidentiality :: (Attacker s, Confidentiality s) =>
                      Sec s' a -> Sec (s,s') a
addConfidentiality (MkSec a) = return a
```

A.5 SecLib.hs

```
-- | The security library.
-- | This is the * only * module of the library to be
-- | imported by untrusted code.
module SecLib
  (
  -- Sec
    Sec
  , up
  -- Attacker
  , addIntegrity
  , addConfidentiality
  , addIntegritySecIO
  , addConfidentialitySecIO
  , public
  -- SecIO
  , SecIO
  , value
  , plug
  -- Files
  , File
  , ACSecIO.readFileSecIO
  , ACSecIO.writeFileSecIO
  , ACSecIO.newFileSecIO
  -- , SecIO.readFileSecIO
  -- , SecIO.writeFileSecIO
  -- , SecIO.newFileSecIO
  -- References
  , Ref
  , ACSecIO.readRefSecIO
  , ACSecIO.writeRefSecIO
  , ACSecIO.newIORefSecIO
  -- , SecIO.readRefSecIO
  -- , SecIO.writeRefSecIO
  -- , SecIO.newIORefSecIO
  -- Network
  , SecSocket
  , SecSockAddr
  , inet_addrSecIO
    -- AC is not needed here, since any
    -- reasonable connection involves sending and
    -- receiving data.
  , bindSocketSecIO
```

```
  , sIsBoundSecIO
  , acceptSecIO
  , socketSecIO
  , recvSecIO
  , sendSecIO
  -- Screen/Keyword
  , Screen
  , getLineSecIO
  , putStrSecIO
  , makeScreen
  -- Legacy code
  , s_read
  , s_write
  -- Declassification
  , Hatch
  , Open
  , Close
  , Authority
  , certify
  -- Access Control
  , AC
  -- Lattice
  , L (..)
  , H (..)
  , Rd  (..)
  , Wrt (..)
  , Any (..)
    -- Method less must not be exported!
  , Less ()
  , Li ()
  , Hi ()
  )
where

import Lattice
import Sec
import SecIO
import Declassification
import AC
import ACSecIO
import Attacker
```

A.6 Declassification.hs

```
-- | Privide declassification combinators (trusted).
module Declassification
(
    Hatch
  , Open
  , Close
  , Authority
  , hatch
  , ntimes
  , flock
  , dlm
  , certify
)

where

import Sec ( reveal )
import SecIO
import Data.IORef
import Lattice ( Less )
import Sec ( Sec )


-- | Type for escape hatches.
type Hatch s s' a b = Sec s a -> SecIO s' b

-- | Used by 'flock'.
-- | 'Open' represents computations that open flow locks.
type Open  = IO ()
-- | Used by 'flock'.
-- | 'Close' represents computations that close flow locks.
type Close = IO ()

-- | Used by 'dlm'.
data Authority s = Authority Open Close


-- | Creates an escape hatch.
hatch :: Less s' s => (a -> b) -> Hatch s s' a b
hatch f = \sa -> return ( ( f (reveal sa) ) )

{- | Limite the number of times that an escape hatch
```

```
     can be applied by a single run of the program.
-}
ntimes :: Int -> Hatch s s' a b -> IO (Hatch s s' a b)
ntimes n f =
     do ref <- newIORef n
        return $ \sa ->
                    MkSecIO $
                     do k <- readIORef ref
                        let MkSecIO m = f sa
                        if k <= 0
                           then error "ntimes fails"
                           else do writeIORef ref (k-1)
                                   m


{- | This function associates a flow lock to an escape hatch.
     Then, the escape hatch can be successfully applied when
     the flow lock is open. In contrast, the escape hatch
     cannot by applied after closing the flock lock.
-}
flock :: Hatch s s' a b -> IO (Hatch s s' a b, Open, Close)
flock f = do ref <- newIORef False
             return (\sa -> MkSecIO $
                              do b <- readIORef ref
                                 let MkSecIO m = f sa
                                 if b then m
                                      else error "flock fails!"
                     , writeIORef ref True
                     , writeIORef ref False
                    )


{- | This function allows to an escape hatch to be
     applied only when the running code can be
     certified with some authority.
-}
dlm :: Hatch s s' a b -> IO (Hatch s s' a b, Authority s )
dlm f = do (whof, open, close) <- flock f
           return (whof, Authority open close)

-- | Certifies that a piece of code have certain authority.
certify :: s -> Authority s -> IO a -> IO a
certify s (Authority open close) io =
  s `seq` ( do open ; a <- io ; close ; return a )
```

A.7 AC.hs

```
module AC where

import Sec
import Lattice

data AC p o = MkAC o

-- Function to wrap access control around an object
ac :: o -> AC p o
ac = MkAC


plugAC :: Less p p' => p' -> (o -> b) -> AC p o -> b
plugAC p' prim ac_p@(MkAC o) = less p p' `seq` prim o
                           where p = unACType ac_p

unACType :: AC p o -> p
unACType = undefined
```

A.8 ACSecIO.hs

```
module ACSecIO where

import AC
import Sec
import SecIO
import Lattice
import Attacker
import Network.Socket

-- Reading primitives
plugRd :: Less p Rd => (o -> b) -> AC p o -> b
plugRd = plugAC MkRd

readFileSecIO :: Less p Rd =>
```

```
                 AC p (File s) -> SecIO s' (Sec s String)
readFileSecIO = plugRd SecIO.readFileSecIO

readRefSecIO :: Less p Rd =
                AC p (Ref s a) -> SecIO s' (Sec s a)
readRefSecIO  = plugRd SecIO.readRefSecIO

-- recvSecIO :: Less p Rd =>
--              AC p (SecSocket s) -> Int -> SecIO s String
-- recvSecIO     = plugRd SecIO.recvSecIO


-- Writing primitives
plugWrt :: Less p Wrt => (o -> b) -> AC p o -> b
plugWrt = plugAC MkWrt

writeFileSecIO :: Less p Wrt =>
                  AC p (File s) -> String -> SecIO s ()
writeFileSecIO = plugWrt SecIO.writeFileSecIO

writeRefSecIO  :: Less p Wrt =>
                  AC p (Ref s a) -> a -> SecIO s ()
writeRefSecIO  = plugWrt SecIO.writeRefSecIO

-- sendSecIO :: Less p Wrt =>
--              AC p (SecSocket s) -> String -> SecIO s Int
-- sendSecIO      = plugWrt SecIO.sendSecIO

-- Creation
newIORefSecIO :: a -> SecIO ss (AC p (Ref s a))
newIORefSecIO a = do r <- SecIO.newIORefSecIO a
                     return (ac r)

-- socketSecIO :: Family -> SocketType ->
--                ProtocolNumber ->
--                SecIO s' (AC p (SecSocket s))
-- socketSecIO  f s pro = do r <- SecIO.socketSecIO f s pro
--                           return (ac r)


newFileSecIO :: Attacker s =>
                FilePath -> SecIO s (AC p (File s))
newFileSecIO f = do ff <- SecIO.newFileSecIO f
                    return (ac ff)
```

APPENDIX B

Code for the Password Administrator

B.1 SecureAPI.hs

```
module SecureAPI
        ( s_login, s_reset, s_backup, s_restore )
where

import AC ( ac )
import SecIO ( run, makeFile )
import SecLib
import Files

-- Login
import Login
import Policies
import SpwdData ( UID )

-- Reset
import Reset

-- Backup & Restore
import Backup
import Encryption ( Method )

s_login :: IO (Maybe UID)
s_login = do match <- declassification
             uid   <- run $ login match login_passwd
```

```
                                        login_shadow
            return (public uid)


s_reset :: IO ()
s_reset = do (Just uid) <- s_login
             run $ reset uid reset_shadow
             return ()


s_backup :: FilePath -> Method -> IO ()
s_backup f m = do f' <- run (newFileSecIO f)
                            :: IO (Sec L (AC Wrt (File L)))
                  run (do pass <- readFileSecIO backup_passwd
                          shad <- readFileSecIO backup_shadow
                          backup (public f') pass shad m)
                  return ()

s_restore :: FilePath -> Method -> IO ()
s_restore f m =
     do f' <- return (ac $ makeFile f :: AC Rd (File L))
        run (do (names,pass) <- restore f' m
                writeFileSecIO restore_passwd names
                plug (do p <- value pass
                         writeFileSecIO restore_shadow p))
                         return ()
```

B.2 Files.hs

```
module Files where

import SecLib
import SecIO ( makeFile, DataInvariant, invariant_output )
import AC ( ac )

import SpwdData

import List ( find )
import Maybe ( isJust )
import Data.Char ( isAlpha, isDigit )

login_passwd :: AC Rd (File L)
login_passwd = ac (makeFile "./passwd")
```

```
login_shadow :: AC Rd (File H)
login_shadow = ac (makeFile "./shadow")

reset_shadow :: AC Wrt (File H)
reset_shadow = ac (invariant_output enforce_Passwd
                   (makeFile "./shadow"))

backup_passwd :: AC Rd (File (L,Hi))
backup_passwd = ac $ makeFile "./passwd"

backup_shadow :: AC Rd (File (H,Hi))
backup_shadow = ac $ makeFile "./shadow"

restore_passwd :: AC Wrt (File (L,Hi))
restore_passwd = ac $ makeFile "./passwd"

restore_shadow :: AC Wrt (File (H,Hi))
restore_shadow = ac $ makeFile "./shadow"


enforce_Passwd :: DataInvariant String
enforce_Passwd str = return $ and
        [(length s >= 8 &&
           isJust (find (isAlpha) s) &&
           isJust (find (isDigit) s) &&
           isJust (find (isSpecial) s)) | (_,s) <- spwd]
        where
        isSpecial = \x -> x > 'z' || x < '0'
        spwd = read str :: [(UID, Cypher)]
```

B.3 SpwdData.hs

```
module SpwdData
 (
    -- Datatypes
      UID
    , Cypher
    , Name
    , Spwd (..)
 )
where
```

```
type UID    = Int
type Cypher = String
type Name   = String

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

B.4 Spwd.hs

```
module Spwd
 (
    -- API
    getSpwdName
    , putSpwd
    , getUID
 )
where

import SecLib
import SpwdData


getSpwdName :: (Less p Rd, Less p' Rd) =>
                AC p (File L) ->
                AC p' (File H) ->
                Name ->
                SecIO L (Maybe (Sec H Spwd))
getSpwdName pwd ac nam =
     do se  <- plug (parseSpwd ac)
        pw <- parsePwd pwd
        case lookup nam pw of
          Nothing  -> return Nothing
          Just n   -> return $
             Just ((\e -> do e' <- e
                             case lookup n e' of
                               Nothing -> error "impossible"
                               Just c  -> return (
                                   Spwd { uid = n ,
                                          cypher = c})) se)


putSpwd :: (Less p Rd, Less p Wrt) =>
            AC p (File H) ->
            Spwd -> SecIO H ()
```

```
putSpwd ac spwd =
     do se  <- parseSpwd ac
        let (id, c) = (uid spwd, cypher spwd)
            (ts,ds) = span (\(u,_) -> u /= id ) se
            updated = if null ds
                      -- new record
                      then ts++[(id,c)]
                      -- update record
                      else ts++[(id,c)]++(tail ds)
        writeFileSecIO ac (show updated)

getUID :: Less p Rd =>
          AC p (File L) -> Name -> SecIO L (Maybe UID)
getUID f nam = do pw <- parsePwd f
                  case lookup nam pw of
                        Nothing -> return Nothing
                        Just n  -> return (Just n)


parseSpwd :: Less p Rd =>
             AC p (File H) -> SecIO H ([(UID, Cypher)])
parseSpwd f = do s  <- readFileSecIO f
                 ss <- value s
                 return (read ss :: [(UID, Cypher)])

parsePwd :: Less p Rd =>
            AC p (File L) -> SecIO L ([(Name, UID)])
parsePwd f = do s  <- readFileSecIO f
                ss <- value s
                return (read ss :: [(Name, UID)])
```

B.5 Policies.hs

```
module Policies ( declassification ) where

import SecLib
import Declassification

import SpwdData
import Spwd

declassification :: IO (Hatch H L (Spwd, Cypher) Bool)
declassification = ntimes 3
```

```
                    (hatch (\(spwd,c) -> cypher spwd == c))
```

B.6 Login.hs

```
module Login ( login ) where

import SecLib

import SpwdData
import Spwd
import Maybe

login :: Less p Rd =>
         (Hatch H L (Spwd, String) Bool) ->
         AC p (File L) ->
         AC p (File H) ->
         SecIO L (Maybe UID)
login match pwd spwd =
     do let ?match = match
        let ?screen = makeScreen
        putStrSecIO ?screen "Welcome!\n"
        putStrSecIO ?screen "login: "
        u <- getLineSecIO ?screen
        src <- getSpwdName pwd spwd u
        case src of
          Nothing -> do putStrSecIO ?screen "Invalid user!\n"
                        return Nothing
          Just p  -> do u' <- getUID pwd u
                        auth 3 (fromJust u') p


auth:: (?screen::Screen,
        ?match::Hatch H L (Spwd, Cypher) Bool) =>
       Int -> UID -> Sec H Spwd -> SecIO L (Maybe UID)
auth 0 _ spwd = return Nothing

auth n u spwd =
  do putStrSecIO ?screen "Password: "
     pwd <- getLineSecIO ?screen
     check spwd pwd n u


check :: ( ?match :: Hatch H L (Spwd, Cypher) Bool,
```

```
          ?screen :: Screen ) =>
         Sec H Spwd ->
         String -> Int -> UID -> SecIO L (Maybe UID)
check spwd pwd n u =
     do acc <- ?match ((\s -> (s,pwd)) `fmap` spwd)
        if acc then do putStrSecIO ?screen "Valid login!\n"
                       return (Just u)
               else do putStrSecIO ?screen "Invalid login!\n"
                       auth (n-1) u spwd
```

B.7 Reset.hs

```
module Reset ( reset ) where

import SecLib

import SpwdData
import Spwd
import Maybe

reset :: (Less p Rd, Less p Wrt) => UID ->
         AC p (File H) -> SecIO L ()
reset uid spwd = do let ?screen = makeScreen
                    putStrSecIO ?screen "New password: "
                    p <- getLineSecIO ?screen
                    plug $ putSpwd spwd (Spwd uid p)
                    return ()
```

B.7.1 Encrypted passwords in the Password Administrator

For simplicity, we do not consider encrypted passwords within the `shadow` file. If the passwords were encrypted when writing to the `shadow` file, the invariant would not work as expected. To address this issue, the data type of invariants can be altered to `type DataInvariant a = a -> IO a`, and slightly altering the implementations of the functions `readFileSecIO` and `writeFileSecIO`. This new notation allows greater flexibility when defining data invariants. This can now be used to accept the new password before it is encrypted, examine the constraints, and then encrypt it before returning the string.

B.8 Encryption.hs

```
module Encryption
```

```
        ( Key ()
        , Content, Cypher, Method, Usernames, Passwords
        , valid_enc
        , Encryption.encrypt
        , Encryption.decrypt
        ) where

import Codec.Encryption.AES as AES
import Codec.Encryption.Blowfish as Blowfish

import Codec.Encryption.Padding
import Codec.Encryption.Modes
import Codec.Utils
import Data.Char
import Data.LargeWord
import Random

import Sec
import SecIO
import SecLib
import Declassification

newtype Key a = MkKey { unkey :: a }
type Content = String -- Unencrypted content
type Cypher  = String -- Encrypted contetn
type Method = (String,String) -- Encryption settings
-- first string is algorithm, second is wordsize
type Usernames = String -- Content read from /etc/passwd
type Passwords = String -- Content read from /etc/shadow

-- The Key
key128 :: Key Word128
key128 = MkKey 0x06a9214036b8a15b512e03d534120006

secureKey :: Sec (H,Hi) (Key Word128)
secureKey = return key128



-- Function to produce a seed using /dev/random
-- (or /dev/urandom for non-blocking)
makeSeed128 :: IO (Word128)
makeSeed128 =
     do s <- readFile "/dev/urandom"
```

```
        return $ fromIntegral
               $ fromTwosComp
               $ map (fromIntegral . ord) (take 128 s)


secureSeed :: SecIO (L,Hi) Word128
secureSeed = MkSecIO (return `fmap` makeSeed128)



-- Endorsing the users choice of
-- encryption method and key size
valid_enc :: Hatch (s,Li) (s,Hi) Method Method
valid_enc = hatch(\(e,s) ->
                case (e,s) of
                    ("AES","128") -> (e,s)
                    ("Blowfish",s)    -> case s of
                                           "64"  -> (e,s)
                                           "128" -> (e,s)
                    _ -> error "Endorsement failed")


-- Takes an encryption method, a key and string,
-- and encrypts the string
encrypt_hatch :: Less (L,Hi) (s,Hi) =>
                 Method ->
                 Hatch (s,Hi) (L,Hi) (Word128,Content) Cypher
encrypt_hatch (alg,word) sec =
      do iv <- secureSeed
         hatch (\(key,text) ->
           case alg of
             "AES" -> show (iv:(encryptString
                           (AES.encrypt) iv key text))
             "Blowfish" ->
               case word of
                -- Seed and key is 64bit
                "64"  -> let (seed,key') = to64(iv,key) in
                             show (seed:
                             (encryptString (Blowfish.encrypt)
                              seed key' text))
                -- Text block and seed is 64bit
                -- Key is left at size 128
                "128" -> let (seed,_) = to64(iv,key) in
                         show (seed:
```

```
                        (encryptString (Blowfish.encrypt)
                         seed key text))
                where
                to64 :: (Show a, Show b) =>
                        (a,b) -> (Word64,Word64)
                to64 (a,b) = (convert a, convert b)
        )  sec


-- Encrypts the given content
-- using the encryption method supplied
encrypt :: Sec (L,Hi) Method ->
           Sec (L,Hi) Usernames ->
           Sec (H,Hi) Passwords -> SecIO (L,Hi) Cypher
encrypt e names passwds =
     do method <- value e
        skey   <- return secureKey
        let arg = do key <- skey
                     k   <- return $ unkey key
                     c   <- text
                     return (k, c)
        encrypt_hatch method arg

        where text = do n <- up names
                        p <- passwds
                        return (show (n,p))


-- Takes an encryption method, a cypher and a key,
-- and decrypts
decrypt_hatch :: Less (s,Hi) (H,Hi) =>
                 Method ->
                 Cypher ->
                 Hatch (H,Hi) (s,Hi) Word128 Content
decrypt_hatch (alg,word) cypher key' = hatch (\key ->
   do (iv:cypher) <- return $
                     map fromIntegral (read cypher::[Integer])
      case alg of
        "AES" -> decryptString (AES.decrypt) iv key cypher
        "Blowfish" ->
          case word of
            -- Converts the seed, key and cypher
            -- to 64bits ((iv:cypher) bound to 128bits by AES)
            "64"  -> let (seed',key',cypher') =
```

```
                        to64(iv,key,cypher) in
                   decryptString (Blowfish.decrypt)
                      seed' key' cypher'
            -- Converts the seed and cypher to 64 bits
            "128" -> let (seed',_,cypher') =
                        to64(iv,key,cypher) in
                   decryptString (Blowfish.decrypt)
                      seed' key cypher'
            where
            to64 :: (Show a, Show b, Show c) =>
                    (a,b,[c]) -> (Word64, Word64, [Word64])
            to64 (s,k,c)=(convert s, convert k, map convert c)
   ) key'

-- Decrypting a cypher with the given encryption method
decrypt :: Method ->
           Cypher ->
           SecIO (L,Hi) (Usernames, Sec (H,Hi) Passwords)
decrypt e c =
     do key    <- secureKey
        text   <- decrypt_hatch e c (unkey key)
        (names,passwds) <- return (read text::(String,String))
        return (names, return passwds)


-- Encrypts with the given encryption method,
-- initialization vector, key and string
encryptString encryptF iv key s =
        cbc encryptF iv key $ pkcs5 $ strToWord s
-- Decrypts with the given encryption method,
-- initialization vector, key and cypher text
decryptString decryptF iv key c =
        wordToStr $ unPkcs5 $ unCbc decryptF iv key c

-- String to Word
strToWord = map (fromIntegral . ord)
-- Word to String
wordToStr = map (chr . fromIntegral)

-- Used to convert a higher Word to a lower one
convert w = fromIntegral (read (show w) :: Integer)
```

B.9 Backup.hs

```
module Backup ( backup, restore ) where

import Encryption

import SecLib

import Monad

-- Backups the content, with the specified encryption method
-- to the specified file
backup :: Less p Wrt =>
          AC p (File L) ->
          Sec (L,Hi) Usernames ->
          Sec (H,Hi) Passwords ->
          Method -> SecIO (L,Hi) ()
backup f names passwds enc =
     do e <- plug $ endorse (return enc)
        c <- encrypt e names passwds
        let write =
            (addIntegritySecIO $ writeFileSecIO f c)
              :: SecIO (L,Li) ()
        plug write
        return ()



-- Restore the content, with the specified encryption method
-- from the specified file
restore :: Less p Rd =>
           AC p (File L) ->
           Method ->
           SecIO (L,Hi) (Usernames, Sec (H,Hi) Passwords)
restore f enc =
     do let read =
            (addIntegritySecIO $ readFileSecIO f)
               :: SecIO (L,Li) (Sec L Cypher)
        cypher <- plug read
        decrypt enc (public (public cypher))
```