Ensuring Information Security by Using Haskell Advanced Type System

Matteo Di Pirro
Department of Mathematics
University of Padua
Padua, Italy

***Abstract*—Protecting data confidentiality and working on validated values have become increasingly important in modern software. Since the first examples of code injection or buffer overflow attacks, language-based security has improved itself. We have developed a huge amount of techniques for detecting software threats. We have also developed from scratch entire new languages for security purposes.**
The core idea behind that is to ensure properties by using type theory. With advanced and complex type systems we are able to check for code vulnerabilities at compile time. Nevertheless, learning a new language, or completely migrate to that, is a complex and difficult operation. This is why, over the last few years, researchers have developed new secure libraries for existing languages.
The contribute of this paper is twofold. On one hand I lightly build an existing Haskell library up. On the other hand I present two new types for ensuring two important properties: operating on validated input values and performing computations on untainted data.

I. Introduction

Over the last few years software has become increasingly complex. It is so much complex that is almost impossible to see how it can be abused. The problem is even worst when one is forced to trust other people's code. Many secure languages have been developed from scratch for solving this kinds of problems. Two well-known examples are Jif [11] by Pullicino and Flowcaml [16] by Simonet and Rocquencourt. The former is a Java extension which adds support for security labels such that the developers can specify confidentiality and integrity policies to the various variables used in their program. The letter, instead, is an extension of the Objective Caml language with a type system tracing information flow. Its purpose is basically to allow to write real programs and to automatically check that they obey some security policy.
However, it is a very heavy-weight solution to introduce a new programming language. In fact, despite of the large work on that, there has been relatively little adoption of the proposed techniques. Moreover, often only a small part of the system (maybe only a few variables in a large program) has security requirements. This is why many researchers have developed new lightweight libraries for ensuring security properties while programming. This paper aims to go further this direction.
Here I present a Haskell based library which ensures some security properties and a real-world use case for that. This library might be used in scenarios where we want to incorporate in our programs some code written by outsiders (untrusted programmers) to access our private information. We would like to have a guarantee the program will not send our private data to an attacker. A slightly different, but related, scenario is where we ourselves write the possible unsafe code, but we want to have the help of the type checker to find possible security mistakes.
Li and Zdancewic [7] have previously explored the possibility to ensure information flow security as a library, but their approach is *arrow* based ([5]). Hence, programmers have to be familiar with arrows. Afterwards, Russo et al. [12] have shown that a monadic solution is also possible. Unfortunately, their work makes an intense use of the `IO` monad, thus their system is non-completely static.
In this paper I lightly build the last mentioned library up and formalise two new types. As a result, the following security properties are met by my work: secure information flow, augmented with *declassification policies*, secure computation on untainted data and dynamic validation of user input before their use.
The remainder of this paper is organized as follows. Section II formalises my assumptions. Section III describes the motivating example. Sections V, VI and IV provide descriptions and implementation details about the three secure types: `Unsecure`, `SecureFlow` and `SecureComputation`. Section VII sums my contribute up and emphasises its limitations. My discussion is then concluded in Section VIII.

II. Assumptions

In the rest of the paper, I assume that the programming language I work with is a controlled version of Haskell, where code is divided up into trusted code, in brief every possible user defined extension of my library, and untrusted code, written by the attacker. The latter may only use the secure modules, and it is defined as the code actually implementing the application.

A. Trusted or Untrusted

Suppose a minimal directory structure like the following:

```
/
 └─App
    └─Src
       └─Model
       └─IO
└─SecureModule
```

```
  ├── Unsecure
  ├── SecureFlow
  └── SecureComputation
```

Here, `SecureModule` represents a directory containing the secure library. `App` is, on the contrary, supposed to contain every application-related file, such as Haskell modules and configuration or data files. In particular, every Haskell module but Main.hs (generally defined as the main file) should be into `Src`. In this simple example `Src` is made up of two subdirectories, `Model` and `IO`. The former should consist of every module implementing the application behavior and managing the data, logic and rules. The latter should include every IO-related module, such as those used for interfacing with users, databases or configuration files.
Only trusted programmers are allowed to write `IO` modules. Their task is to specify declassification policies, input validation functions and to *override* every IO function in order to satisfy security requirements. Conversely, untrusted programmers may not write anything related with IO, but they can contribute to any other application-related file, as well as trusted ones.

III. A REAL-WORLD USE CASE

Consider a company which would like to manage its employees data and the situation of its stores. Suppose that every operation on those data must be done after a login. In such a scenario, passwords should remain secret, and a declassification policy should be applied during the login procedure. In fact, after a successful login, a user can retrieve some information, such as that the provided password was correct.
Subsequently, the logged user may manage the stores situation. Moreover, if he or she is a company leader, he or she can increase an employee's salary. The salary is strictly confidential, so nobody may know it.
The following security properties must be met in this application. First, passwords and salaries are confidential and the application may not make them known. Second, a natural number (e.g. the increment amount) coming from the user must be validated before its use. In addition, in sensitive operations, the value must be marked as pure (i.e. untainted). We would be sure that two of them, secure information flow and secure computation over data, are ensured at compile time, before the application execution. The third, input validation, must be performed at run-time, since input vary from an execution to another. Thus, the type system will only be able to check that an input value is validated before a real use. I shall go deeper on that in Section IV.
The application database is composed by three JSON (**Ja**va**S**cript **O**bject **N**otation) files, as follows: `credentials.json`, `employees.json` and `stores.json`, with the obvious meaning. The choice of JSON files is motivated by the simplicity in their manipulation. A SQL-based (or whatever) version would be possible too.

IV. THE UNSECURE MODULE

The `Unsecure` module makes sure an input value will be validated before its use. Listing 1 shows the definition.

Listing 1. Unsecure module

```
type ValidationFunctions a b =
  [a -> Maybe b]
data Unsecure a b =
  Unsecure (ValidationFunctions a b, a)

validate :: Unsecure a b -> Either a [b]

umap :: Unsecure a b -> (a -> a)
  -> Unsecure a b

upure :: a -> ValidationFunctions a b
  -> Unsecure a b
```

Here, `a` and `b` represent *every possible* type, as usual in Haskell type definitions. The former is the input type and the latter is an *error* type. Using an error type is necessary, because the validation could fail. If it fails, the error list is returned. Following the Haskell error mechanism, every error in that list is represented as a constructor of the type `b`.
`Unsecure` is defined as a pair made up of `ValidationFunctions` (*VF*), from a value of type `a` to a value of an error type `b`, and a value of type `a` (`v` in the rest of the section). `validate` simply makes sure `v` actually meets the *VF* constraints. If so, `v` is returned; otherwise it returns the error list, since `v` could fail more than one constraint. This is why `Either` is required.
`upure` and `umap` try to simulate an applicative approach. The former allows programmers to create an `Unsecure` value. The latter allows them to manipulate `v` before validation. An `Unsecure` value is supposed to be created by an IO function and manipulated by programmers. It may not be defined as a canonical functor or applicative because of its own nature. Type `a` must not be changed, because with this definition every validation function is based on it. A feasible solution would be a constrained functor type signature, but this may not be done with normal functors, as well as applicatives. Nevertheless, Sculthorpe et al. [15] had shown that this would be achievable with a little effort. There are a lot of other works on this topic, such as Co-Yoneda functor, based on Yoneda Lemma ([4], [3]). However, for the sake of simplicity, this work is limited to canonical functor, applicatives and monads.
Listing 2 shows how `Unsecure` can be used for validating strings as natural numbers.

Listing 2. Unsecure for natural numbers

```
data NatError = NegativeNumber |
    NonNumeric

getNat :: IO (Unsecure String NatError)
getNat = do n <- getLine
            return $ upure n [
                    isNumeric ,
                    isNatural
                ]

isNumeric s = if null $ dropWhile isDigit
      s
              then Nothing
              else Just NonNumeric

isNatural n = if read n >= 0
              then Nothing
              else Just NegativeNumber

useNat = do n <- getNat
            case validate n of
              Left nat -> operation nat
              Right es -> showerrors es
```

getNat takes a string from the user and constraints that with two functions, isNumeric and isNatural. NatError represents a category of possible errors. Each validation function should have a respective error type. When a user is asked for a natural number an IO (Unsecure String NatError) is given to the programmers instead of an IO Int. Doing so, they can be sure the boxed String actually represents what they want.
Even with this minimal example the Unsecure benefits are clear: programmers must validate inputs before using them. As an immediate consequence, many vulnerabilities are called off. As another example, consider a function returning a string and validating it by checking its length. Buffer overflow attacks are based on a poor check of the string length, so by using Unsecure we are sure about the validity of that string. This is possible with a few lines of code.

V. The SecureFlow module

Software usually manipulates information with different security policies. For instance, passwords are sensitive data, while names or birth dates are not. During execution sometimes we want to be sure sensitive information doesn't flow to insecure or public output channels. On the other hand, usually we have to perform some operations with reserved data: this is often done declassifying information.
The goal with SecureFlow is to ensure information flow security and declassification policies at compile-time, so that if a code is correctly compiled there are no certain security violations.
Russo et al. [12] have developed a monadic approach, but their intensive use of the IO monad makes their solution not fully static. In this section I lightly build their work up removing any IO reference. For the sake of simplicity I don't give a discussion about how this can enforce security properties on files or any other data storage mechanism.

A. Security lattice

SecureFlow is based on a lattice structure, first introduced by Denning [2], and represented in this library as a type family [6]. Security levels are associated to data in order to establish their degree of confidentiality. The lattice ordering relation, written $\sqsubseteq$, represents allowed flows. For instance, $l_1 \sqsubseteq l_2$ indicates that information at security level l_1 may flow into entities of security level l_2. In this paper the lattice is implemented like a ticketing system. Listing 3 shows the Lattice module.

Listing 3. The Lattice module
```
type family LEQ sl sh :: Constraint
data Ticket s = Ticket
```

Basically, LEQ, as a type family, represents a partial function at the type level. Applying the function to parameters (called *type indices*) yields a type. Type families permit a program to compute what data constructors it will operate on, rather than having them fixed statically. Here, LEQ means *Less or Equal*. Trusted programmers are asked to instantiate it specifying the real security lattice. A three level lattice example is shown in Listing 4.

Listing 4. Three levels lattice
```
module ThreeLevels (Low, Medium, High,
    low, medium) where

data Low    = L
data Medium = M
data High   = H

type instance (LEQ Low Low)         = ()
type instance (LEQ Low Medium)      = ()
type instance (LEQ Low High)        = ()
type instance (LEQ Medium Medium)   = ()
type instance (LEQ Medium High)     = ()
type instance (LEQ High High)       = ()

low :: Ticket Low
low = Ticket

medium :: Ticket Medium
medium = Ticket

high :: Ticket High
high = Ticket
```

Here, Low, Medium and High are singleton types ([17], [9]). Basically, singleton types are those which have only one value. Thus, the value of a singleton type has a unique type representing the value. A type theory that allows types to be parameterised by values (like the one adopted by Haskell) can use singleton types to let types depend on singleton values. The sequence of type instance allows programmers to specify the relations among Low, Medium and High. Listing 4 shows also a ticketing system instance. Tickets are generally used as a level proof. Note that the high ticket is not exported, so that one may get access to high secure data only according to the declassification policies. The ordering relation among `Low`, `Medium` and `High` is the following: $Low \sqsubseteq Medium \sqsubseteq High$.

B. SecureFlow Implementation

Basically, `SecureFlow` is just an identity monad instance tagged with a proposition allowing access to its value. Actually, that proposition is just a ticket. Listing 5 shows the module.

Listing 5. SecureFlow monad

```
data SecureFlow s a = SF a
type Hatch s a b = SecureFlow s (a -> b)

instance Functor (SecureFlow s) where
  fmap f (SF x)  = SF $ f x

instance Applicative (SF s) where
  pure x             = SF x
  (SF f) <*> (SF x) = SF $ f x

instance Monad (SecureFlow s) where
  return            = pure
  (Allowed a) >>= f = f a

open :: LEQ s s' => Ticket s' ->
    SecureFlow s a -> a

up :: LEQ s s' => SecureFlow s a ->
    SecureFlow s' a

declassifyWith :: (LEQ s k, LEQ s' s) =>
    Hatch k a b -> SecureFlow s a ->
    SecureFlow s' b
```

Proposition 1. *`SecureFlow`, as defined, is a monad.*

Proof. I am to prove the three monad laws.

1) **Left identity**: `return a >>= f = f a`

```
(return a >>= f) = (pure a >>= f) =
= ((SF a) >>= f) = f a
```

2) **Right identity**: `m >>= return = m`

```
((SF a) >>= return) =
= (return a) = (pure a) =
= (SF a) = m
```

3) **Associativity**: `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

```
(((SF a) >>= f) >>= g) =
= ((f a) >>= g) = (g (f a))

(SF a >>= (\x -> f x >>= g)) =
= (\x -> f x >>= g) a =
= ((f a) >>= g) = (g (f a))
```

`SecureFlow` satisfies the three monad laws, hence it is a monad. □

`SecureFlow` is also a functor and an applicative. The proof is largely similar to the last one, so it is omitted.
The module also exports three important functions, `up`, `open` and `declassifyWith`.
`open` is used to look at a protected value of type `SecureFlow s a`. However, in order to do that, one has to prove to have a value of type `s`. This proof is given with the ticketing system. Note that, if the value is protected with `SecureFlow s a`, the ticket type must be in the `LEQ` relation with `s`. This actually means $s \sqsubseteq s'$ must hold. Otherwise the compiler gives a type error because of the unsatisfied type constraint `LEQ s s'`.
The function `up` can be used to turn any protected value into a protected value at a higher security level. Basically it acts like a cast function from a lower to a higher level.

C. Declassification

Non-interference is a security policy which specifies the absence of information flows from secret to public data. So far, the provided library has followed strictly this policy. However, real-word applications release some information as part of their intended behavior. Non-interference does not provide any way to distinguish between such releases of information and those ones produced by malicious code, programming errors or vulnerability attacks. Consequently, relaxing the notion of non-interference is necessary, considering declassification policies as intended ways to leak information.
Sabelfeld and Sands [13] have provided a classification of those policies. Basically, each dimension represents a specific policy aspect corresponding to *what*, *where*, *when* and by *whom* data is released. Researchers have defined type systems for enforcing these aspects ([1], [19], [18]), but their encoding into this work would considerably increase its complexity. This is why here I provide a *declassification combinator*, `Hatch`, which allows trusted programmers to compose and create declassification policies from scratch. Those policies are based on the `SecureFlow` monad, hence a statical type check is done on them. In this way, declassification still lies on the non-interference idea, but untrusted programmers are allowed to work with secret data without producing leaks. Furthermore, only trusted programmers are authorised to write them.
Basically, `Hatch` is just a *SecureFlow*-encapsulated function of type `a -> b`. `declassifyWith` simply applies that function to the actual value and returns a new value with a lower security level. In order to work, the relation $s' \sqsubseteq s \sqsubseteq k$ must hold. It is such that the final security tag will be *LEQ* than the original one.
Listing 6 shows a login function.

Listing 6. Declassificated login

```
login :: String -> String ->
  Hatch High [Credential] Bool
login e p = pure
  (\cs -> elem (Credential e p) cs)
```

```
check = (declassifyWith (login e p) cs)
  :: SecureFlow Medium Bool
success = open medium check
```

Basically `Credential` represents the pair *(email address, password)*, and it is tagged with the *High* level. `login` simply takes a list of *Credentials* and checks for a match with the provided email address and password. Note that `High` in `login` type signature refers to the list security level rather than to the final one. The latter is code provided giving an explicit type to the `declassifyWith` result (in this case, *Medium*, forced by `:: SecureFlow Medium Bool`). With a `medium` ticket we shall be able to access the boolean value and check whether or not the user is actually allowed to login. That would not be possible without declassifying *Credentials* list.
Listing 7 shows how a sensitive salary could be declassified, for instance, for test purposes.

Listing 7. Declassificated salary

```
-- Employee model would be like the
-- following:
-- { firstName  :: SecureFlow Low String
-- , lastName   :: SecureFlow Low String
-- , birthdate  :: SecureFlow Low String
-- , salary     :: SecureFlow High Int
-- , email      :: SecureFlow Medium String
-- , leader     :: SecureFlow Low Bool
-- }

showSalary :: Hatch High Int Int
showSalary = pure id
```

This model specifies a *High* level for `salary`, but thanks to `showSalary` a declassification to another security level l, where $l \sqsubseteq High$, would be possible.
Two other important things are, finally, pointed out by this example. First, `SecureFlow` may be applied both for an entire module (like *Credential* before), or with a more fine granularity, as in *Employee*. Second, declassification functions are extremely sensitive and only trusted programmers ought to write them. However, remembering my assumptions, only trusted programmers are supposed to "instantiate" the secure library, so that once they have defined declassification policies no one will be able to modify them. Otherwise there would be no way to ensure information flow security with this kind of approach.

VI. The SecureComputation module

The idea behind `SecureComputation` is exactly the same as `SecureFlow`. The difference relies on its aim. The former provides a way of working with pure data.
Taint analysis ([14], [8]) is a well-known technique for dynamic detecting software vulnerabilities. However, this dynamic dimension is not fully sound when applied to modern software. It is so much complex that a full testing process is practically infeasible. Moreover, like Dijkstra said, "program testing can be used to show the presence of bugs, but never to show their absence". That means we cannot just trust dynamic analysis. We need a statical check.
My version of this statical check is `SecureComputation` shown in a shortened form in Listing 8.

Listing 8. SecureComputation module

```
data SC m a = SC a

type family MustBePure m :: Constraint
data P = P
data T = T
type instance (MustBePure P) = ()

open :: MustBePure m => SC m a -> a
open (SC a) = a

smap :: (MustBePure m, MustBePure m') =>
    (a -> b) -> SC m a -> SC m' b
smap f (SC a) = SC $ f a

spure :: a -> SC m a
spure = SC

sapp :: (MustBePure m, MustBePure m') =>
    SC m' (a -> b) -> SC m a -> SC m' b
sapp (SC f) sc = smap f sc

sreturn :: a -> SecureComputation m a
sreturn = spure

sbind :: (MustBePure m, MustBePure m') =>
     SC m a -> (a -> SC m' b) -> SC m' b
sbind (SC a) f = f a
```

As one can note, `SecureComputation` is based on the same type family method as `SecureFlow`. Basically, `P` and `T` are singleton types meaning *Pure* and *Tainted*. Naturally, only `P` is defined as an instance of the `MustBePure` constraint.
An *SecureComputation* encapsulated value may be opened if and only if the `SecureComputation` holder is pure. Furthermore, computations on encapsulated values are allowed if and only if those values are pure (in the meaning that their containers are).
Again, `SecureComputation` is not a monad because of its type constraints. Making it a monad would be possible ([15]), as stated in Section IV, but it is out of this paper scope. For the sake of simplicity I redefine functor, applicative and monad functions with another name (actually just adding *s* as prefix) so that it might be used *like* a monad. The meaning of those functions (`smap`, `spure`, `sapp` and `sbind`) is the usual one except for type constraints. Validity of a potential real instantiation would be also provable without considering type constraints. An example is given by the following proposition.

Proposition 2. *`SecureComputation`, without type constraints, is a monad.*

Proof. I am to prove the three monad laws. During the proof, `>>=` and `return` are supposed to be, respectively, `sbind` and `sreturn`. I use the original names for the sake of consistency.

1) **Left identity**: `return a >>= f = f a`

```
(return a >>= f) = (spure a >>= f) =
= ((SC a) >>= f) = f a
```

2) **Right identity**: `m >>= return = m`

```
((SC a) >>= return) =
= (return a) = (spure a) =
= (SC a) = m
```

3) **Associativity**: `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

```
(((SF a) >>= f) >>= g) =
= ((f a) >>= g) = (g (f a))

(SF a >>= (\x -> f x >>= g)) =
= (\x -> f x >>= g) a =
= ((f a) >>= g) = (g (f a))
```

`SecureComputation` satisfies the three monad laws, hence it is a monad. □

`SecureComputation` is useful as a type for user-provided values. Listing 9 shows how an input could be encapsulated and marked as tainted. Haskell, in fact, does not provide a function returning a `Num` (where `Num` is an abstract type class concretised by every type class representing a number, such as `Int` or `Float`). Thus one has to use `getLine`, which returns a `String`. In Section IV I showed a way of validating this `String`. Here, contrariwise, no validation is performed on the user provided value; it is just marked as tainted (or not pure), so that it cannot be used as a parameter when a pure computation is required.

Listing 9. Tainted natural number

```
getUnpureNat :: IO (SecureComputation T
    String)
getUnpureNat = do n <- getLine
                  return $ spure n
```

Note that programmers can mark as tainted every value they want. They are also able to make differences among different input sources and values. That points `SecureComputation` flexibility out. For instance, recall the running example, and in particular the operations on stores. Every stored product has a price and a number representing its stocks. Increments or decrements on those numbers are allowed only with pure values. Otherwise a type error must be detected. A suitable general function should have the following type signature:

```
SC P a -> String -> (a -> Store -> Store)
-> SC P [Store] -> SC P [Store].
```

The parameters have meanings as follows:

1) modification value (*v*);
2) product name (*p*);
3) the real modification function (*f*) (for instance, `modifyPrice`);
4) a list of stores (*s*).

It returns a list of stores where *p* has been modified according to *f* based on *v*. Note that *v* and *s* must be pure while it doesn't matter for *p*. A type error is detected every time an unpure (or tainted) value is provided supposing it to be a pure one. If the code compiles and `SecureComputation` is cleverly used, there are not operations on pure data based on tainted one.

VII. Limitations

Every security technique has some drawbacks. In particular, with software security, the main drawback is an usability loss. Secure languages and type systems are chiefly used in critical situations, where a secure software is absolutely mandatory. Programmers in those scenarios must write secure and verified code. On the other hand, the vast majority of current software does not make the security the principal concern. Furthermore, programmers sometimes write leaky code as a consequence of an usability lack of the programming language itself.

This library, as defined, suffers from such usability problems. First of all, trusted programmers must override and define new input functions to include `Unsecure` and `SecureComputation` in the application code. This will be annoying if every new function requires an override. On the other hand, a fine granularity input check without those modules would imply a huge amount of code, with possible repetitions, mistakes and consistency lacks. Anyway, those modules are supposed to be used in small portions of the entire program, and in particular only for sensitive values or scenarios. Otherwise a fully secure programming language would be more suitable. Furthermore, with the current version, `Unsecure` and `SecureComputation` are not functor, applicative and monad instances.

Despite its complete independence from the final use, the `Hatch` combinator leaves some freedom to programmers. As a matter of fact, it is defined in terms of the encapsulated-value security tag and programmers have to specify explicitly the final level. That entails a possible attack. Consider the following `showSalary` function, first shown in Listing 7:

```
showSalary :: Hatch High Int Int
showSalary = pure id
```

High is referred to the salary level *before* the declassification, but `Hatch` specifies nothing about the level *after* the declassification. That is a big leak because, with the three levels lattice, one may open the declassified value with either a `low` or a `medium` ticket. Everything is left to the explicit type annotation which, based on my assumptions, might be written by an untrusted programmer! A feasible solution is a more strict, and complicated, `Hatch` version, shown in Listing 10 and named `Hatch'`.

Listing 10. A more strict Hatch version

```
type Hatch' s l a b = SecureFlow l (
    SecureFlow s (a -> b))
```

```
makeHatch :: (a -> b) -> Hatch' s l a b
makeHatch f = pure $ pure f

declassifyWith' :: (LEQ s k, LEQ s' s,
    LEQ l s') => Hatch' k l a b ->
    SecureFlow s a -> SecureFlow s' b
```

In this new version, `Hatch′` is defined as a double `SecureFlow` encapsulation of the declassification function. The new type parameter, `l` is required to specify a *lower bound* for the final security tag. As a matter of fact, note the new type constraint in `declassifyWith′`. By the virtue of this new constraint, the relation $l \sqsubseteq s'$ must hold. Therefore I am forcing the final tag to be *LEQ* than the one declared in the type signature. The new `showSalary` function will be as follows:

```
showSalary :: Hatch′ High Medium Int Int
showSalary = makeHatch id
```

Where `makeHatch` is just a shortcut to hide the double `SecureFlow` encapsulation.

With this modification, programmers still have to explicitly declare the final tag as a type annotation, but that type is constrained by the declassification policy type signature, so that it must be at least (*LEQ*) the declared one.

Finally, in order to force the declared and the actual final tags to be the same, another type constraint ought to be added to `declassifyWith′`: `LEQ s′ l`. This means the relations $l \sqsubseteq s'$ and $s' \sqsubseteq l$ must hold, which requires l to be equal to s'.

VIII. Conclusions

Taking ideas from the literature, in this paper I have presented a simple library for information security in Haskell. I have formalised three new types satisfying three important principles: mandatory input validation, non-inference and computation on pure data. Two of them ensure the corresponding property in a static way so that it is satisfied if and only if the source code compiles. The first, contrariwise, may only be checked at run-time.

Besides, the library provides a simple way for formalising declassification policies. Considering it is wholly generalised, it might be adapted for satisfying almost every security requirement.

The actual Haskell implementation is partially based on monads, a widespread concept in functional programming. Although only one out of three types is a monad instance, the idea behind the other two is exactly the same. It would be possible to make them concrete monad instances, but that would require an effort out of the scope of this paper.

The library implementation and every example shown in this paper are publicly available in [10].

References

[1] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive declassification policies and modular static enforcement," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 339–353.

[2] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[3] Edward Kmett, "Data.functor.coyoneda," https://hackage.haskell.org/package/kan-extensions-5.0.1/docs/Data-Functor-Coyoneda.html, 2016, online; accessed 15 January 2017.

[4] D. Elkins, "Calculating monads with category theory," *The Monad. Reader Issue 13*, p. 73, 2009.

[5] J. Hughes, "Generalising monads to arrows," *Science of computer programming*, vol. 37, no. 1, pp. 67–111, 2000.

[6] O. Kiselyov, S. P. Jones, and C.-c. Shan, "Fun with type functions," in *Reflections on the Work of CAR Hoare*. Springer, 2010, pp. 301–331.

[7] P. Li and S. Zdancewic, "Encoding information flow in haskell," in *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. IEEE, 2006, pp. 12–pp.

[8] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[9] B. C. Pierce, *Advanced topics in types and programming languages*. MIT press, 2005.

[10] M. D. Pirro, "Haskell secure types," https://github.com/mdipirro/haskell-secure-types/tree/master/app, 2017.

[11] K. Pullicino, "Jif: Language-based information-flow security in java," *arXiv preprint arXiv:1412.8639*, 2014.

[12] A. Russo, K. Claessen, and J. Hughes, "A library for light-weight information-flow security in haskell," in *ACM Sigplan Notices*, vol. 44, no. 2. ACM, 2008, pp. 13–24.

[13] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. IEEE, 2005, pp. 255–269.

[14] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 2010, pp. 317–331.

[15] N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill, "The constrained-monad problem," in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2013, pp. 287–298.

[16] V. Simonet and I. Rocquencourt, "Flow caml in a nutshell," in *Proceedings of the first APPSEM-II workshop*. Nottingham, United Kingdom, 2003, pp. 152–165.

[17] C. A. Stone, "Singleton kinds and singleton types," DTIC Document, Tech. Rep., 2000.

[18] S. Zdancewic, "A type system for robust declassification," *Electronic Notes in Theoretical Computer Science*, vol. 83, pp. 263–277, 2003.

[19] S. Zdancewic and A. C. Myers, "Robust declassification." in *csfw*, vol. 1. Citeseer, 2001, pp. 15–23.