Ensuring Information Security by Using Haskell's Advanced Type System

Matteo Di Pirro
Department of Mathematics
University of Padua
matteo.dipirro@studenti.unipd.it

Mauro Conti
Department of Mathematics
University of Padua
conti@math.unipd.it

Riccardo Lazzeretti
Department of Computer, Control and Management Engineering
Sapienza University of Rome
lazzeretti@diag.uniroma1.it

***Abstract*—Protecting data confidentiality and integrity has become increasingly important in modern software. Sometimes, access control mechanisms come short and solutions on the application-level are needed. An approach can rely on enforcing information security using some features provided by certain programming languages. Several different solutions addressing this problem have been presented in literature, and entire new languages or libraries have been built from scratch. Some of them use type systems to let the compiler check for vulnerable code. In this way we are able to rule out those implementations which do not meet a certain security requirement. In this paper we use Haskell's type system to enforce three key properties of information security: non-interference and flexible declassification policies, strict input validation, and secure computations on untainted and trusted values.**

We present a functional lightweight library for applications with data integrity and confidentiality issues. Our contribute relies on a compile time enforcing of the aforementioned properties. Our library is wholly generalized and might be adapted for satisfying almost every security requirement.

I. Introduction

Over the last few years, software has become so complex that is almost impossible to see how it can be misused. The problem is even worst when you are forced to trust other people's code. Many secure languages have been developed from scratch for solving this kind of problems. Two well-known examples are Jif [12] by Pullicino, and Flowcaml [18] by Simonet and Rocquencourt. The former is a Java extension adding support for security labels such that the developers can specify confidentiality and integrity policies to the various variables used in their program. The latter, instead, is an extension of the Objective Caml language with a type system tracing information flows. Its purpose is basically to allow writing real programs and to automatically checking whether they obey some security policy. However, introducing a new programming language is a very heavy solution. In fact, despite of the large work on that, there has been relatively little adoption of the proposed techniques. Moreover, usually only a small part of the system (maybe only a few variables in a large program) has security requirements. This is the reason why many researchers have developed new lightweight libraries for ensuring security properties while programming. This paper aims to go further this direction.

978-1-5386-1585-0/17/$31.00

The paper presents a Haskell-based library ensuring some security properties and a real-world use case. This library might be used to allow some outsiders' untrusted code to access our private information, guaranteeing that the program will not send private data to third party.

In our listings, we use the traditional Haskell's notation for defining functions. A complete introduction to this syntax, and to Haskell's functors and monads can be found in [1] [2].

Li and Zdancewic [8] have previously explored the possibility to ensure information flow security as a library, but their approach is arrow-based [6]. Hence, programmers have to be familiar with arrows. Afterwards, Russo et al. [13] have shown that a monadic solution is also possible. They have provided a lightweight Haskell library for writing and reading files. Unfortunately, their implementation of declassification policies makes an intensive use of the IO Monad. Hence, it is not completely pure and side-effects free.

In this paper we lightly broaden the latter library and formalize two new types. As a result, the following security properties are satisfied by our work: secure information flow, augmented with declassification policies, secure computation on untainted data, and dynamic user input validation. The last two are new in the overview of the language-based security. Computation on pure data is important in very sensitive applications. Every value in such applications should come from a trusted source of input: our contribution is to ensure this requirement at compile time, with high flexibility about the actual input source. Furthermore, validation is the main reason of software vulnerabilities. With our work, the programmers will be forced to validate every untrusted value and to provide two different paths in the source code. Those paths will correspond to a good situation, in which there were no errors, and to a bad situation, in which one or more errors occurred during the validation process. Although the actual validation is performed at run-time, the need of validation comes at compile time, before executing the software.

With these three simple types, on one hand we are to prove how easily some fundamental properties may be ensured; on the other hand, we supply a concrete framework usable in almost every scenario needing a fine-granularity software control. More work and types would be required for a comprehensive assurance. The paper explains in brevity how fruitful this way could be.

The remainder of this paper is organized as follows. Section II briefly overviews related work. Section III formalizes our assumptions. Section IV describes the motivating example. Section V provides descriptions and implementation details about the three secure types: `Unsecure`, `SecureFlow` and `SecureComputation`. Our discussion is then concluded in Section VI.

II. Related Work

Zdancewic [20] explains the challenges related to the use of language based information security techniques in real-world software. According to him, such technologies suffer of usability problems. In particular, specifying complex security properties directly in the source code is often a difficult error-prone operation. Our library makes it simple to specify different security requirements with different granularities, as shown in Section V-B. A small subset of programmers is asked to redefine some I/O functions and to specify every needed declassification policy. This has to be done only once, before writing the rest of the application. There is no need for further annotations.

Li and Zdancewic [8] provide a Haskell implementation based on arrows combinators [6] for ensuring information flow security as a library. Russo et al. [13] show similar result implemented only using monads, a functional construct simpler than arrows. The latter approach also provide a way for dealing with files without forcing programmers to use verbose types. They prove the non-interference guarantee given by their formalization, on which `SecureFlow` is partially based on. Their approach to declassification policies differentiates among four dimensions (*who*, *what*, *where* and by *whom*, [14]). Unfortunately they make an intensive use of the IO Monad, thus their solution is not completely side-effect free. Our proposal uses only `SecureFlow` and allows programmers to impose constraints on the levels involved in the declassification.

Scholte et al. [15] show how most software attacks are due to absence of input validation. Those vulnerabilities are detectable using IDE plugins, as shown by Baset and Denning [3], but their adoption is poor. Even if they help increasing the security of code, it is cumbersome for developers to evaluate different tools on their own. Furthermore, each tool may use a different approach leading to different results and false positives and negatives. Other approaches, such as [5] for Android applications by Scholte et al., aim to detect those vulnerabilities, but simple detection is often not enough. As shown in Section V-A, our library provides an easy way to impose restrictions and validations constraints on existing types. In case of error it gives information about what was wrong. In this way programmers are forced to validate input values, if they want to use them.

III. Assumptions

In the rest of the paper, we assume that the programming language we work with is a controlled version of Haskell, where code is divided up into trusted code, without any restriction, and untrusted code, written by the attacker. The latter may not define any I/O module, but only use them. If an untrusted programmer wrote sensitive code there would be no possibility to ensure security using programming languages.

Our library allows potentially bad code to interact with the users and with sensitive data, but not to define how these interactions will occur.

We suppose that every information is safely stored in an external database. It does not matter what kind of database is actually used. The library we present allows trusted programmers to wrap the I/O functions needed for interacting with the database in more secure functions and to compose them according to the functional paradigm.

IV. 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 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, so that, after a successful login, some information is disclosed, such as the password correctness.

Subsequently, the logged user may manage the stores situation. Moreover, if he or she is a company leader then he or she may increase an employee's salary. The salary is strictly confidential, so nobody should know it.

The following security properties must be satisfied in this application. Firstly, passwords and salaries are confidential and the application may not make them known. Secondly, 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).

In such a scenario the different security requirements are specified at different granularities. The password constraint acts at a database level, while the increment one is finer and is relative to a specific functionality. It is important that software is developed such that misbehaviors are prevented. Nonetheless, is not worth to write the entire application using a complex language with complex security features. A better solution would be to add security only when needed, in order to keep the code as simple as possible.

V. The library

In this section we present our library. It is composed by three types. The first, `Unsecure`, is used to validate values coming from untrusted sources of input. It helps programmers to impose constraints on the input values. The second, `SecureFlow`, is used to enforce non-interference and define declassification policies. Lastly, `SecureComputation`, is used to force values to be trusted before their use in sensitive operations.

A. Unsecure

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

Listing 1: The Unsecure module

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

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

Here, `a` and `b` represent *every possible* type, as usual in Haskell's 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, a list representing every error occurred is returned. Each error 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 errors list, since `v` could fail more than one constraint. This is why `Either` is required.

An `Unsecure` value is supposed to be created by an I/O 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 that. Nevertheless, making it a concrete functor instance is possible, as shown by Sculthorpe et al. [17]. We define two functions, `upure` and `umap`, to simulate an applicative approach. Their type signature is shown in Listing 2. The former allows programmers to create an `Unsecure` value. The latter allows them to manipulate `v` before validation.

Listing 2: Definition of upure and umap

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

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

Listing 3 shows how `Unsecure` can be used for validating strings as natural numbers.

Listing 3: 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 and imposes two constraints represented by `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`. In this way they can be sure the boxed `String` actually represents what they want.

B. SecureFlow

Software usually manipulates information with different security policies. For instance, passwords are sensitive data, while names are not. During the execution sometimes we want to be sure that sensitive information does not flow to insecure or public output channels. On the other hand, we often have to perform some operations on reserved data: this is usually done by declassifying information.

The goal of `SecureFlow` is to ensure information flow security and declassification policies at compile time, so that if a source code is correctly compiled there are no certain security violations.

1) Security lattice: `SecureFlow` is based on a lattice structure, first introduced by Denning [4], and represented in this library as a type family [7]. Security levels are associated to data in order to establish their degree of confidentiality. The lattice ordering relation, written $\sqsubseteq$, represents the 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 proof system. Listing 4 shows the `Lattice` module.

Listing 4: The Lattice module

```
type family LEQ sl sh :: Constraint
data Proof s = Proof
```

LEQ means *Less or Equal*. Trusted programmers are asked to implement concrete instances specifying the actual security lattice. A three level lattice example is shown in Listing 5.

Listing 5: 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 :: Proof Low
```

```
low = Proof

medium :: Proof Medium
medium = Proof

high :: Proof High
high = Proof
```

Every security level is defined as a new singleton type [19] [10]. The sequence of `type instance` allows programmers to specify the relations among `Low`, `Medium` and `High`. Listing 5 shows also a proof system instance. Note that `high` is not exported. Hence, 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: $Low \sqsubseteq Medium \sqsubseteq High$.

2) SecureFlow Implementation: Listing 6 shows the `SecureFlow` module.

Listing 6: The SecureFlow monad

```
newtype SecureFlow s a = SF a

instance Monad (SecureFlow s) where
  return        = SF x
  (SF a) >>= f = f a

open :: LEQ s s' => Proof s' -> SecureFlow s a
    -> a
open _ (SF a) = a

up :: LEQ s s' => SecureFlow s a -> SecureFlow
    s' a
up (SF a) = SF a
```

The module also exports two important functions, `up` and `open`.

`open` is used to look at a protected value of type `SecureFlow s a`. To do that, a level proof must be provided. Note that, if the value is protected with `SecureFlow s a`, the proof 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.

3) Declassification: Non-interference is a security policy specifying the absence of information flows from secret to public channels. So far, `SecureFlow` 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 attacks. Consequently, relaxing the notion of non-interference is necessary, considering declassification policies as intended ways to leak information.

According to the Haskell's functional paradigm, we provide a declassification combinator, `Hatch`, which allows trusted programmers to compose and create declassification policies from scratch. To make its definition completely pure and side-effects free, we define `Hatch` in terms of `SecureFlow`. A naive approach could be as shown in Listing 7:

Listing 7: Naive declassification

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

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

Where `declassifyWith` simply applies the function to the actual value and returns a new value with a lower security level.

We may then write a function showing a salary in the following way:

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

Here, `High` is referred to the level *before* the declassification, but `Hatch` specifies nothing about the level *after* the declassification. This is a big leak. The result's type will be as specified by the user. Based on our assumptions, that user might be an untrusted programmer! This is the reason why we propose a stricter, even if more complex, definition, as shown in Listing 8.

Listing 8: A stricter 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
```

`Hatch` is defined as a double `SecureFlow` encapsulation of the declassification function. The `l` type parameter 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 we are forcing the final tag to be *LEQ* than the one declared in the type signature. Furthermore, 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'. `makeHatch` is just a shortcut to hide the double `SecureFlow` encapsulation.

Listing 9 shows a login function.

Listing 9: Declassificated login

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

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

`Credential` represents the pair *(email address, password)* and 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. Programmers have to explicitly declare the final tag as a type annotation (`Medium` in this case), but the type is constrained by the declassification policy's type signature, so that it must be at least (*LEQ*) the declared one (`High`). With a `medium` proof 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 the list.

`SecureFlow` may be used for an entire module (such as with `Credential`), or with a more fine granularity, as shown in listing 10.

Listing 10: Finer SecureFlow application

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

C. SecureComputation

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 [16] [9] is a well-known technique for dynamically detect software vulnerabilities. However, this dynamic dimension is not fully sound when applied to modern software. That is so much complex that a full testing process is practically infeasible. Thus, we cannot just trust dynamic analysis. We need a statical check.

Our version of this statical check is `SecureComputation`, shown in a shortened form in Listing 11.

Listing 11: The SecureComputation module

```
newtype SC m a = SC a

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
```

`SecureComputation` is based on the same type family method as `SecureFlow`. We define two levels, pure (`P`) and tainted (`T`). Only `P` is defined as a `MustBePure`'s instance.

A *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).

`SecureComputation` is not a monad because of its type constraints. Making it a monad would be possible [17], as stated in Section V-A, but it is out of this paper's scope. For the sake of simplicity we redefine functor, applicative and monad functions with another name (actually just adding *s* as prefix) so that they might be used *like* a monad. The meaning of those functions (`smap`, `spure`, `sapp` and `sbind`) is the usual one except for the type constraints.

`SecureComputation` is useful as a type for user-provided values. Listing 12 shows how a value can be encapsulated and marked as tainted. Haskell, in fact, does not provide a function returning a `Num` (where `Num` is an abstract type class concretized by every type class representing a number, such as `Int` or `Float`). Thus, `getLine` has to be used, and it returns a `String`. In Section V-A we 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 12: Tainted natural number

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

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 does not matter for *p*. A type error is detected every time an un-pure (or tainted) value is provided supposing it to be a pure one.

If the code compiles and `SecureComputation` is cleverly used, there are no operations on pure data based on tainted values.

VI. Conclusions

In this paper we have presented a simple library for information security in Haskell. We have formalized three new types satisfying three important principles: mandatory input validation, non-interference, and computation on pure data. Two of them ensure the corresponding properties in a static way so that they are satisfied if and only if the source code compiles. The first, contrariwise, may only be checked at runtime.

Besides, the library provides a simple way for formalizing declassification policies. Considering it is wholly generalized, 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 similar. 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 [11].

As we said, this simple library is not enough for a comprehensive assurance. We reserve the addition of more types, and more features, as future work. For instance, we could provide a native approach for the four declassification policies' dimensions [14]. Furthermore, we could impose constraints on how the programmers access the data. Using a proof sistem similar to the `SecureFlow`'s one, we might allow or deny access to various informations sources, without relying on access control mechanisms.

Acknowledgment

Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980). This work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), the EU-India REACH Project (agreement ICI+/2014/342-896), and by the projects "Physical-Layer Security for Wireless Communication", and "Content Centric Networking: Security and Privacy Issues" funded by the University of Padua.

This work is partially supported by the grant n. 2017-166478 (3696) from Cisco University Research Program Fund and Silicon Valley Community Foundation. This work is also partially funded by the project CNR-MOST/Taiwan 2016-17 "Verifiable Data Structure Streaming".

References

[1] "Applicative functor," https://wiki.haskell.org/Applicative_functor, 2017.

[2] "Monad," https://wiki.haskell.org/Monad, 2017.

[3] A. Z. Baset and T. Denning, "Ide plugins for detecting input-validation vulnerabilities," 2017.

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

[5] Z. Fang, Q. Liu, Y. Zhang, K. Wang, Z. Wang, and Q. Wu, "A static technique for detecting input validation vulnerabilities in android apps," *Science China Information Sciences*, vol. 60, no. 5, p. 052111, 2017.

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

[7] 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.

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

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

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

[11] M. D. Pirro, "Ensuring information security by using haskell's advanced type system," https://github.com/mdipirro/haskell-secure-types-library, 2017.

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

[13] 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.

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

[15] T. Scholte, D. Balzarotti, and E. Kirda, "Have things changed now? an empirical study on input validation vulnerabilities in web applications," *Computers & Security*, vol. 31, no. 3, pp. 344–356, 2012.

[16] 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.

[17] 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.

[18] 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.

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

[20] S. Zdancewic, "Challenges for information-flow security," in *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID04)*, 2004.