A Dependently Typed Library for Static Information-Flow Control in Idris

Simon Gregersen(B), Søren Eller Thomsen, and Aslan Askarov

Aarhus University, Aarhus, Denmark
{gregersen,sethomsen,askarov}@cs.au.dk

Abstract. Safely integrating third-party code in applications while protecting the confidentiality of information is a long-standing problem. Pure functional programming languages, like Haskell, make it possible to enforce lightweight information-flow control through libraries like **MAC** by Russo. This work presents DepSec, a **MAC** inspired, dependently typed library for static information-flow control in Idris. We showcase how adding dependent types increases the expressiveness of state-of-the-art static information-flow control libraries and how DepSec matches a special-purpose dependent information-flow type system on a key example. Finally, we show novel and powerful means of specifying statically enforced declassification policies using dependent types.

Keywords: Information-flow control · Dependent types · Idris

1 Introduction

Modern software applications are increasingly built using libraries and code from multiple third parties. At the same time, protecting confidentiality of information manipulated by such applications is a growing, yet long-standing problem. Third-party libraries could in general have been written by anyone and they are usually run with the same privileges as the main application. While powerful, such privileges open up for abuse.

Traditionally, access control [7] and encryption have been the main means for preventing data dissemination and leakage, however, such mechanisms fall short when third-party code needs access to sensitive information to provide its functionality. The key observation is that these mechanisms only place restrictions on the access to information but not its propagation. Once information is accessed, the accessor is free to improperly transmit or leak the information in some form, either by intention or error.

Language-based Information-Flow Control [36] is a promising technique for enforcing information security. Traditional enforcement techniques analyze how information at different security levels flows within a program ensuring that information flows only to appropriate places, suppressing illegal flows. To achieve this, most information-flow control tools require the design of new languages, compilers, or interpreters (e.g. [12,17,22,23,26,29,39]). Despite a large, growing

F. Nielson and D. Sands (Eds.): POST 2019, LNCS 11426, pp. 51–75, 2019.
https://doi.org/10.1007/978-3-030-17138-4_3

body of work on language-based information-flow security, there has been little adoption of the proposed techniques. For information-flow policies to be enforced in such systems, the whole system has to be written in new languages – an inherently expensive and time-consuming process for large software systems. Moreover, in practice, it might very well be that only small parts of an application are governed by information-flow policies.

Pure functional programming languages, like Haskell, have something to offer with respect to information security as they strictly separate side-effect free and side-effectful code. This makes it possible to enforce lightweight information-flow control through libraries [11,20,34,35,42] by constructing an embedded domain-specific security sub-language. Such libraries enforce a secure-by-construction programming model as any program written against the library interface is not capable of leaking secrets. This construction forces the programmer to write security-critical code in the sub-language but otherwise allows them to freely interact and integrate with non-security critical code written in the full language. In particular, static enforcement libraries like **MAC** [34] are appealing as no run-time checks are needed and code that exhibits illegal flows is rejected by the type checker at compile-time. Naturally, the expressiveness of Haskell's type system sets the limitation on which programs can be deemed secure and which information flow policies can be guaranteed.

Dependent type theories [24,31] are implemented in many programming languages such as Coq [13], Agda [32], Idris [8], and F* [44]. Programming languages that implement such theories allow types to dependent on values. This enables programmers to give programs a very precise type and increased confidence in its correctness.

In this paper, we show that dependent types provide a direct and natural way of expressing precise data-dependent security policies. Dependent types can be used to represent rich security policies in environments like databases and data-centric web applications where, for example, new classes of users and new kinds of data are encountered at run-time and the security level depends on the manipulated data itself [23]. Such dependencies are not expressible in less expressive systems like **MAC**. Among other things, with dependent types, we can construct functions where the security level of the output depends on an argument:

```
getPassword : (u : Username) -> Labeled u String
```

Given a user name `u`, `getPassword` retrieves the corresponding password and classifies it at the security level of `u`. As such, we can express much more precise security policies that can depend on the manipulated data.

Idris is a general-purpose functional programming language with full-spectrum dependent types, that is, there is no restrictions on which values may appear in types. The language is strongly influenced by Haskell and has, among others, inherited its strict encapsulation of side-effects. Idris essentially asks the question: "What if Haskell had full dependent types?" [9]. This work, essentially, asks:

"What if **MAC** had full dependent types?"

We address this question using Idris because of its positioning as a general-purpose language rather than a proof assistant. All ideas should be portable to equally expressive systems with full dependent types and strict monadic encapsulation of side-effects.

In summary, the contributions of this paper are as follows.

- We present DepSec, a **MAC** inspired statically enforced dependently typed information-flow control library for Idris.
- We show how adding dependent types strictly increases the expressiveness of state-of-the-art static information-flow control libraries and how DepSec matches the expressiveness of a special-purpose dependent information-flow type system on a key example.
- We show how DepSec enables and aids the construction of policy-parameterized functions that abstract over the security policy.
- We show novel and powerful means to specify statically-ensured declassification using dependent types for a wide variety of policies.
- We show progress-insensitive noninterference [1] for the core library in a sequential setting.

Outline. The rest of the paper proceeds through a presentation of the DepSec library (Sect. 2); a conference manager case study (Sect. 3) and the introduction of policy-parameterized functions (Sect. 4) both showcasing the expressiveness of DepSec; means to specify statically-ensured declassification policies (Sect. 5); soundness of the core library (Sect. 6); and related work (Sect. 7).

All code snippets presented in the following are extracts from the source code. All source code is implemented in Idris 1.3.1. and available at

https://github.com/simongregersen/DepSec.

1.1 Assumptions and Threat Model

In the rest of this paper, we require that code is divided up into trusted code, written by someone we trust, and untrusted code, written by a potential attacker. The trusted computing base (TCB) has no restrictions, but untrusted code does not have access to modules providing input/output behavior, the data constructors of the domain specific language and a few specific functions related to declassification. In Idris, this means that we specifically do not allow access to `IO` functions and `unsafePerformIO`. In DepSec, constructors and functions marked with a `TCB` comment are inaccessible to untrusted code. Throughout the paper we will emphasize when this is the case.

We require that all definitions made by untrusted code are total, that is, defined for all possible inputs and are guaranteed to terminate. This is necessary if we want to trust proofs given by untrusted code. Otherwise, it could construct an element of the empty type from which it could prove anything:

```
empty : Void
empty = empty
```

In Idris, this can be checked using the `--total` compiler flag. Furthermore, we do not consider concurrency nor any internal or termination covert channels.

2 The DepSec Library

In information-flow control, labels are used to model the sensitivity of data. Such labels usually form a security lattice [14] where the induced partial ordering $\sqsubseteq$ specifies allowed flows of information and hence the security policy. For example, $\ell_1 \sqsubseteq \ell_2$ specifies that data with label ℓ_1 is allowed to flow to entities with label ℓ_2. In DepSec, labels are represented by values that form a verified join semilattice implemented as Idris interfaces[1]. That is, we require proofs of the lattice properties when defining an instance of `JoinSemilattice`.

```
interface JoinSemilattice a where
  join : a -> a -> a
  associative :
    (x, y, z : a) -> x `join` (y `join` z) = (x `join` y) `join` z
  commutative : (x, y : a)    -> x `join` y = y `join` x
  idempotent  : (x : a)       -> x `join` x = x
```

Dependent function types (often referred to as Π types) in Idris can express such requirements. If `A` is a type and `B` is a type indexed by a value of type `A` then `(x : A) -> B` is the type of functions that map arguments `x` of type `A` to values of type `B x`.

A lattice induces a partial ordering, which gives a direct way to express flow constraints. We introduce a verified partial ordering together with an implementation of this for `JoinSemilattice`. That is, to define an instance of the `Poset` interface we require a concrete instance of an associated data type `leq` as well as proofs of necessary algebraic properties of `leq`.

```
interface Poset a where
  leq : a -> a -> Type
  reflexive     : (x : a) -> x `leq` x
  antisymmetric : (x, y : a) -> x `leq` y -> y `leq` x -> x = y
  transitive    : (x, y, z : a) -> x `leq` y -> y `leq` z -> x `leq` z

implementation JoinSemilattice a => Poset a where
  leq x y = (x `join` y = y)
  ...
```

This definition allows for generic functions to impose as few restrictions as possible on the user while being able to exploit the algebraic structure in proofs, as will become evident in Sects. 3 and 4. For the sake of the following case studies, we also have a definition of a `BoundedJoinSemilattice` requiring a least element `Bottom` of an instance of `JoinSemilattice` and a proof of the element being the unit.

[1] Interfaces in Idris are similar to type classes in Haskell.

```
data Labeled : label -> Type -> Type where
  MkLabeled : valueType -> Labeled label valueType -- TCB

data DIO : l -> Type -> Type where
  MkDIO : IO valueType -> DIO l valueType -- TCB

Monad (DIO l) where
  ...

label : Poset label => {l : label} -> a -> Labeled l a

unlabel : Poset label => {l, l' : label}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> DIO l' a

plug : Poset label => {l, l' : label}
    -> DIO l' a
    -> {auto flow : l `leq` l'}
    -> DIO l (Labeled l' a)

run : DIO l a -> IO a -- TCB

lift : IO a -> DIO l a -- TCB
```

Fig. 1. Type signature of the core DepSec API.

The Core API. Figure 1 presents the type signature of DepSec's core API. Notice that names beginning with a lower case letter that appear as a parameter or index in a type declaration will be automatically bound as an implicit argument in Idris, and the `auto` annotation on implicit arguments means that Idris will attempt to fill in the implicit argument by searching the calling context for an appropriate value.

Abstract data type `Labeled` ℓ a denotes a value of type a with sensitivity level ℓ. We say that `Labeled` ℓ a is *indexed* by ℓ and *parameterized* by a. Abstract data type `DIO` ℓ a denotes a secure computation that handles values with sensitivity level ℓ and results in a value of type a. It is internally represented as a wrapper around the regular `IO` monad that, similar to the one in Haskell, can be thought of as a state monad where the state is the entire world. Notice that both data constructors `MkLabeled` and `MkDIO` are not available to untrusted code as this would allow pattern matching and uncontrolled unwrapping of protected entities. As a consequence, we introduce functions `label` and `unlabel` for labeling and unlabeling values. Like Rajani and Garg [33], but unlike **MAC**, the type signature of `label` imposes no lattice constraints on the computation context. This does not leak information as, if $l \sqsubseteq l'$ and a computation c has type `DIO` l' (`Labeled` l V) for any type V, then there is no way for the labeled return value of c to escape the computation context with label l'.

As in **MAC**, the API contains a function `plug` that safely integrates sensitive computations into less sensitive ones. This avoids the need for nested computations and *label creep*, that is, the raising of the current label to a point where the computation can no longer perform useful tasks [34,47]. Finally, we also have functions `run` and `lift` that are only available to trusted code for unwrapping of the `DIO` ℓ monad and lifting of the `IO` monad into the `DIO` ℓ monad.

Labeled Resources. Data type `Labeled` ℓ a is used to denote a labeled Idris value with type a. This is an example of a *labeled resource* [34]. By itself, the core library does not allow untrusted code to perform any side effects but we can safely incorporate, for example, file access and mutable references as other labeled resources. Figure 2 presents type signatures for files indexed by security levels used for secure file handling while mutable references are available in the accompanying source code. Abstract data type `SecFile` ℓ denotes a secure file with sensitivity level ℓ. As for `Labeled` ℓ a, the data constructor `MkSecFile` is not available to untrusted code.

The function `readFile` takes as input a secure file `SecFile l'` and returns a computation with sensitivity level `l` that returns a labeled value with sensitivity level `l'`. Notice that the `l` $\sqsubseteq$ `l'` flow constraint is required to enforce the *no read-up* policy [7]. That is, the result of the computation returned by `readFile` only involves data with sensitivity at most `l`. The function `writeFile` takes as input a secure file `SecFile l''` and a labeled value of sensitivity level `l'`, and it returns a computation with sensitivity level `l` that returns a labeled value with sensitivity level `l''`. Notice that both the `l` $\sqsubseteq$ `l'` and `l'` $\sqsubseteq$ `l''` flow constraints are required, essentially enforcing the *no write-down* policy [7], that is, the file never receives data more sensitive than its sensitivity level.

Finally, notice that the standard library functions for reading and writing files in Idris used to implement the functions in Fig. 2 do not raise exceptions. Rather, both functions return an instance of the sum type `Either`. We stay consistent with Idris' choice for this instead of adding exception handling as done in **MAC**.

```
data SecFile : {label : Type} -> (l : label) -> Type where
  MkSecFile : (path : String) -> SecFile l -- TCB

readFile : Poset label => {l,l' : label}
        -> {auto flow : l `leq` l'}
        -> SecFile l'
        -> DIO l (Labeled l' (Either FileError String))

writeFile : Poset label => {l,l',l'' : label}
         -> {auto flow : l `leq` l'} -> {auto flow' : l' `leq` l''}
         -> SecFile l''
         -> Labeled l' String
         -> DIO l (Labeled l'' (Either FileError ()))
```

Fig. 2. Type signatures for secure file handling.

3 Case Study: Conference Manager System

This case study showcases the expressiveness of DepSec by reimplementing a conference manager system with a fine-grained data-dependent security policy introduced by Lourenço and Caires [23]. Lourenço and Caires base their development on a minimal λ-calculus with references and collections and they show how secure operations on relevant scenarios can be modelled and analysed using *dependent information flow types*. Our reimplementation demonstrates how DepSec matches the expressiveness of such a special-purpose built dependent type system on a key example.

In this scenario, a user is either a regular user, an author user, or a program committee (PC) member. The conference manager contains information about the users, their submissions, and submission reviews. This data is stored in lists of references to records, and the goal is to statically ensure, by typing, the confidentiality of the data stored in the conference manager system. As such, the security policy is:

- A registered user's information is not observable by other users.
- The content of a paper can be seen by its authors as well as its reviewers.
- Comments to the PC of a submission's review can only be seen by other members that are also reviewers of that submission.
- The only authors that are allowed to see the grade and the review of the submission are those that authored that submission.

To achieve this security policy, Lourenço and Caires make use of indexed security labels [22]. The security level U is partitioned into a number of security compartments such that $U(uid)$ represents the compartment of the registered user with id uid. Similarly, the security level A is indexed such that $A(uid, sid)$ stands for the compartment of data belonging to author uid and their submission sid, and PC is indexed such that $PC(uid, sid)$ stands for data belonging to the PC member with user id uid assigned to review the submission with id sid. Furthermore, levels $\top$ and $\bot$ are introduced such that, for example, $U(\bot) \sqsubseteq U(uid) \sqsubseteq U(\top)$. Now, the security lattice is defined using two equations:

$$\forall uid, sid.\ U(uid) \sqsubseteq A(uid, sid) \tag{1}$$

$$\forall uid1, uid2, sid.\ A(uid1, sid) \sqsubseteq PC(uid2, sid) \tag{2}$$

Lourenço and Caires are able to type a list of submissions with a dependent sum type that assigns the content of the paper the security level $A(uid, sid)$, where uid and sid are fields of the record. For example, if a concrete submission with identifier 2 was made by the user with identifier 1, the content of the paper gets classified at security level $A(1, 2)$. In consequence, $A(1, 2) \sqsubseteq PC(n, 2)$ for any $uid\ n$ and the content of the paper is only observable by its assigned reviewers. Similar types are given for the list of user information and the list of submission reviews, enforcing the security policy described in the above.

To express this policy in DepSec, we introduce abstract data types `Id` and `Compartment` (cf. Fig. 3) followed by an implementation of the `BoundedJoinSemilattice` interface that satisfies Eqs. (1) and (2).

```
data Id : Type where
  Top : Id
  Nat : Nat -> Id
  Bot : Id

data Compartment : Type where
  U  : Id -> Compartment
  A  : Id -> Id -> Compartment
  PC : Id -> Id -> Compartment
```

Fig. 3. Abstract data types for the conference manager sample security lattice.

```
record User where
  constructor MkUser
  uid   : Id
  name  : Labeled (U uid) String
  univ  : Labeled (U uid) String
  email : Labeled (U uid) String

record Submission where
  constructor MkSubmission
  uid   : Id
  sid   : Id
  title : Labeled (A uid sid) String
  abs   : Labeled (A uid sid) String
  paper : Labeled (A uid sid) String

record Review where
  constructor MkReview
  uid     : Id
  sid     : Id
  PC_only : Labeled (PC uid sid) String
  review  : Labeled (A Top sid) String
  grade   : Labeled (A Top sid) Integer
```

Fig. 4. Conference manager types encoded with DepSec.

Using the above, the required dependent sum types can easily be encoded with DepSec in Idris as presented in Fig. 4. With these typings in place, implementing the examples from Lourenço and Caires [23] is straightforward. For example, the function `viewAuthorPapers` takes as input a list of submissions and a user identifier `uid1` from which it returns a computation that returns a list of submissions authored by the user with identifier `uid1`. Notice that `uid` denotes the automatically generated record projection function that retrieves the field `uid` of the record, and that `(x: A ** B)` is notation for a dependent pair (often referred to as a Σ type) where `A` and `B` are types and `B` may depend on `x`.

```
viewAuthorPapers : Submissions
                -> (uid1 : Id)
                -> DIO Bottom (List (sub : Submission ** uid1 = (uid sub)))
```

The `addCommentSubmission` operation is used by the PC members to add comments to the submissions. The function takes as input a list of reviews, a user identifier of a PC member, a submission identifier, and a comment with label `A uid1 sid1`. It returns a computation that updates the `PC_only` field in the review of the paper with identifier `sid1`.

```
addCommentSubmission : Reviews -> (uid1 : Id) -> (sid1 : Id)
                    -> Labeled (A uid1 sid1) String
                    -> DIO Bottom ()
```

Notice that to implement this specific type signature, up-classification is necessary to assign the comment with type `Labeled (A uid1 sid1) String` to a field with type `Labeled (PC uid sid1) String`. This can be achieved soundly with the `relabel` primitive introduced by Vassena et al. [47] as `A uid1 sid1` $\sqsubseteq$ `PC uid sid1`. We include this primitive in the accompanying source code together with several other examples. The entire case study amounts to about 300 lines of code where half of the lines implement and verify the lattice.

4 Policy-Parameterized Functions

A consequence of using a dependently typed language, and the design of DepSec, is that functions can be defined such that they abstract over the security policy while retaining precise security levels. This makes it possible to reuse code across different applications and write other libraries on top of DepSec. We can exploit the existence of a lattice `join`, the induced poset, and their verified algebraic properties to write such functions.

```
readTwoFiles : BoundedJoinSemilattice label
            => {l, l' : label}
            -> SecFile l
            -> SecFile l'
            -> DIO Bottom (Labeled (join l l') (Either FileError String))
readTwoFiles file1 file2 {l} {l'} =
  do file1' <- readFile {flow = leq_bot_x l} file1
     file2' <- readFile {flow = leq_bot_x l'} file2
     let dio : DIO (join l l') (Either FileError String)
       = do c1 <- unlabel {flow = join_x_xy l l'} file1'
            c2 <- unlabel {flow = join_y_xy l l'} file2'
            pure $ case (c1, c2) of
                        (Right c1', Right c2') => Right $ c1' ++ c2'
                        (Left e1, _) => Left e1
                        (_, Left e2) => Left e2
     plug {flow = leq_bot_x (join l l')} dio
```

Fig. 5. Reading two files to a string labeled with the join of the labels of the files.

Figure 5 presents the function `readTwoFiles` that is parameterized by a bounded join semilattice. It takes two secure files with labels `l` and `l'` as input and returns a computation that concatenates the contents of the two files labeled with the join of `l` and `l'`. To implement this, we make use of the `unlabel` and `readFile` primitives from Figs. 1 and 2, respectively. This computation unlabels the contents of the files and returns the concatenation of the contents if no file error occurred. Notice that `pure` is the Idris function for monadic return, corresponding to the `return` function in Haskell. Finally, this computation is plugged into the surrounding computation. Notice how the usage of `readFile`

and `unlabel` introduces several proof obligations, namely $\bot \sqsubseteq$ `l`, `l'`, `l` $\sqcup$ `l'` and `l`, `l'` $\sqsubseteq$ `l` $\sqcup$ `l'`. When working on a concrete lattice these obligations are usually fulfilled by Idris' automatic proof search but, currently, such proofs need to be given manually in the general case. All obligations follow immediately from the algebraic properties of the bounded semilattice and are given in three auxiliary lemmas `leq_bot_x`, `join_x_xy`, and `join_y_xy` available in the accompanying source code (amounting to 10 lines of code).

Writing functions operating on a fixed number of resources is limiting. However, the function in Fig. 5 can easily be generalized to a function working on an arbitrary data structure containing files with different labels from an arbitrary lattice. Similar to the approach taken by Buiras et al. [11] that hide the label of a labeled value using a data type definition, we hide the label of a secure file with a dependent pair

```
GenFile : Type -> Type
GenFile label = (l : label ** SecFile l)
```

that abstracts away the concrete sensitivity level of the file. Moreover, we introduce a specialized join function

```
joinOfFiles : BoundedJoinSemilattice label
           => List (GenFile label)
           -> label
```

that folds the `join` function over a list of file sensitivity labels. Now, it is possible to implement a function that takes as input a list of files, reads the files, and returns a computation that concatenates all their contents (if no file error occurred) where the return value is labeled with the join of all their sensitivity labels.

```
readFiles : BoundedJoinSemilattice a
         => (files: (List (GenFile a)))
         -> DIO Bottom (Labeled (joinOfFiles files)
                                (Either (List FileError) String))
```

When implementing this, one has to satisfy non-trivial proof obligations as, for example, that $l \sqsubseteq$ `joinOfFiles(files)` for all secure files $f \in$ `files` where the label of f is l. While provable (in 40 lines of code in our development), if equality is decidable for elements of the concrete lattice we can postpone such proof obligations to a point in time where it can be solved by reflexivity of equality. By defining a decidable lattice order

```
decLeq : JoinSemilattice a => DecEq a => (x, y : a) -> Dec (x `leq` y)
decLeq x y = decEq (x `join` y) y
```

we can get such a proof "for free" by inserting a dynamic check of whether the flow is allowed. With this, a `readFiles'` function with the exact same functionality as the original `readFiles` function can be implemented with minimum effort. In the below, `prf` is the proof that the label `l` of `file` may flow to `joinOfFiles files`.

```
readFiles' : BoundedJoinSemilattice a => DecEq a
          => (files: (List (GenFile a)))
          -> DIO Bottom (Labeled (joinOfFiles files)
                                 (Either (List FileError) String))
readFiles' files =
  ...
  case decLeq l (joinOfFiles files) of
    Yes prf => ...
    No _ => ...
```

The downside of this is the introduction of a negative case, the `No`-case, that needs handling even though it will never occur if `joinOfFiles` is implemented correctly.

In combination with `GenFile`, `decLeq` can be used to implement several other interesting examples. For instance, a function that reads all files with a sensitivity label below a certain label to a string labeled with that label. The accompanying source code showcases multiple such examples that exploit decidable equality.

5 Declassification

Realistic applications often release some secret information as part of their intended behavior; this action is known as *declassification*.

In DepSec, trusted code may declassify secret information without adhering to any security policy as trusted code has access to both the `DIO` ℓ a and `Labeled` ℓ a data constructors. However, only giving trusted code the power of declassification is limiting as we want to allow the use of third-party code as much as possible. The main challenge we address is how to grant untrusted code the right amount of power such that declassification is only possible in the intended way.

Sabelfeld and Sands [38] identify four dimensions of declassification: *what*, *who*, *where*, and *when*. In this section, we present novel and powerful means for static declassification with respect to three of the four dimensions and illustrate these with several examples. To statically enforce different declassification policies we take the approach of Sabelfeld and Myers [37] and use escape hatches, a special kind of functions. In particular, we introduce the notion of a *hatch builder*; a function that creates an escape hatch for a particular resource and which can only be used when a certain condition is met. Such an escape hatch can therefore be used freely by untrusted code.

5.1 The *what* Dimension

Declassification policies related to the *what* dimension place restrictions on exactly "what" and "how much" information is released. It is in general difficult to statically predict how data to be declassified is manipulated or changed by programs [35] but exploiting dependent types can get us one step closer.

To control what information is released, we introduce the notion of a *predicate hatch builder* only available to trusted code for producing hatches for untrusted code.

```
predicateHatchBuilder : Poset lt => {l, l' : lt} -> {D, E : Type}
                     -> (d : D)
                     -> (P : D -> E -> Type)
                     -> (d : D ** Labeled l (e : E ** P d e)
                                -> Labeled l' E) -- TCB
```

Intuitively, the hatch builder takes as input a data structure `d` of type `D` followed by a predicate `P` upon `d` and something of type `E`. It returns a dependent pair of the initial data structure and a declassification function from sensitivity level `l` to `l'`. To actually declassify a labeled value `e` of type `E` one has to provide a proof that `P d e` holds. Notice that this proof may be constructed in the context of the sensitivity level `l` that we are declassifying from.

The reason for parameterizing the predicate `P` by a data structure of type `D` is to allow declassification to be restricted to a specific context or data structure. This is used in the following example of an auction system, in which only the highest bid of a specific list of bids can be declassified.

Example. Consider a two point lattice where `L` $\sqsubseteq$ `H`, `H` $\not\sqsubseteq$ `L` and an auction system where participants place bids secretly. All bids are labeled `H` and are put into a data structure `BidLog`. In the end, we want only the winning bid to be released and hence declassified to label `L`. To achieve this, we define a declassification predicate `HighestBid`.

```
HighestBid : BidLog -> Bid -> Type
HighestBid = \log, b => (Elem (label b) log, MaxBid b log)
```

Informally, given a log `log` of labeled bids and a bid `b`, the predicate states that the bid is in the log, `Elem (label b) log`, and that it is the maximum bid, `MaxBid b log`. We apply `predicateHatchBuilder` to a log of bids and the `HighestBid` predicate to obtain a specialized escape hatch of type `BidHatch` that enforces the declassification policy defined by the predicate.

```
BidHatch : Type
BidHatch = (log : BidLog ** Labeled H (b : Bid ** HighestBid log b)
                           -> Labeled L Bid)
```

This hatch can be used freely by untrusted code when implementing the auction system. By constructing a function

```
getMaxBid : (r : BidLog) -> DIO H (b : Bid ** HighestBid r b)
```

untrusted code can plug the resulting computation into an `L` context and declassify the result value using the argument `hatch` function.

```
auction : BidHatch -> DIO L (Labeled L Bid)
auction ([] ** _) = pure $ label ("no bids", 0)
auction (r :: rs ** hatch) =
  do max <- plug (getMaxBid (r :: rs))
    let max' : Labeled L Bid = hatch max
    ...
```

To show the `HighestBid` predicate (which in our implementation comprises 40 lines of code), untrusted code will need a generalized `unlabel` function that establishes the relationship between `label` and the output of `unlabel`. The only difference is its return type: a computation that returns a value and a proof that when labeling this value we will get back the initial input. This definition poses no risk to soundness as the proof is protected by the computation sensitivity level.

```
unlabel' : Poset lt => {l,l': lt}
        -> {auto flow: l `leq` l'}
        -> (labeled: Labeled l a)
        -> DIO l' (c : a ** label c = labeled)
```

Limiting Hatch Usage. Notice how escape hatches, generally, can be used an indefinite number of times. The `Control.ST` library [10] provides facilities for creating, reading, writing, and destroying state in the type of Idris functions and, especially, allows tracking of state change in a function type. This allows us to limit the number of times a hatch can be used. Based on a concept of resources, a dependent type `STrans` tracks how resources change when a function is invoked. Specifically, a value of type `STrans m returnType in_res out_res` represents a sequence of actions that manipulate state where `m` is an underlying computation context in which the actions will be executed, `returnType` is the return type of the sequence, `in_res` is the required list of resources available before executing the sequence, and `out_res` is the list of resources available after executing the sequence.

To represent state transitions more directly, `ST` is a type level function that computes an appropriate `STrans` type given a underlying computation context, a result type, and a list of *actions*, which describe transitions on resources. Actions can take multiple forms but the one we will make use of is of the form `lbl ::: ty_in :-> ty_out` that expresses that the resource `lbl` begins in state `ty_in` and ends in state `ty_out`. By instantiating `ST` with `DIO l` as the underlying computation context:

```
DIO' : l -> (ty : Type) -> List (Action ty) -> Type
DIO' l = ST (DIO l)
```

and use it together with a resource `Attempts`, we can create a function `limit` that applies its first argument `f` to its second argument `arg` with `Attempts (S n)` as its initial required state and `Attempts n` as the output state.

```
limit : (f : a -> b) -> (arg : a)
     -> DIO' l b [attempts ::: Attempts (S n) :-> Attempts n]
```

That is, we encode that the function consumes "an attempt." With the `limit` function it is possible to create functions where users are forced, by typing, to specify how many times it is used.

As an example, consider a variant of an example by Russo et al. [35] where we construct a specialized hatch `passwordHatch` that declassifies the boolean comparison of a secret number with an arbitrary number.

```
passwordHatch : (labeled : Labeled H Int)
             -> (guess : Int)
             -> DIO' l Bool [attempts ::: Attempts (S n) :-> Attempts n]
passwordHatch (MkLabeled v) = limit (\g => g == v)
```

To use this hatch, untrusted code is forced to specify how many times it is used.

```
pwCheck : Labeled H Int
       -> DIO' L () [attempts ::: Attempts (3 + n) :-> Attempts n]
pwCheck pw =
  do x1 <- passwordHatch pw 1
     x2 <- passwordHatch pw 2
     x3 <- passwordHatch pw 3
     x4 <- passwordHatch pw 4 -- type error!
     ...
```

5.2 The *who* and *when* Dimensions

To handle declassification policies related to *who* may declassify information and *when* declassification may happen we introduce the notion of a *token hatch builder* only available to trusted code for producing hatches for untrusted code to use.

```
tokenHatchBuilder : Poset labelType => {l, l' : labelType} -> {E, S : Type}
                 -> (Q : S -> Type)
                 -> (s : S ** Q s) -> Labeled l E -> Labeled l' E -- TCB
```

The hatch builder takes as input a predicate `Q` on something of type `S` and returns a declassification function from sensitivity level `l` to `l'` given that the user can prove the existence of some `s` such that `Q s` holds. As such, by limiting when and how untrusted can obtain a value that satisfy predicate `Q`, we can construct several interesting declassification policies.

The rest of this section discusses how predicate hatches can be used for time-based and authority-based control of declassification; the use of the latter is demonstrated on a case study.

Time-Based Hatches. To illustrate the idea of token hatches for the *when* dimension of declassification, consider the following example. Let `Time` be an abstract data type with a data constructor only available to trusted code and `tick : DIO l Time` a function that returns the current system time wrapped in the `Time` data type such that this is the only way for untrusted code to construct anything of type `Time`. Notice that this does not expose an unrestricted timer API as untrusted code can not inspect the actual value.

Now, we instantiate the token hatch builder with a predicate that demands the existence of a `Time` token that is greater than some specific value.

```
TimeHatch : Time -> Type
TimeHatch t = (t' ** t <= t' = True) -> Labeled H Nat -> Labeled L Nat
```

As such, `TimeHatch t` can only be used after a specific point in time `t` has passed as only then untrusted code will be able to satisfy the predicate.

```
timer : Labeled H Nat -> TimeHatch t -> DIO L ()
timer secret {t} timeHatch =
  do time <- tick
     case decEq (t <= time) True of
       Yes prf =>
         let declassified : Labeled L Nat = timeHatch (time ** prf) secret
         ...
       No _ => ...
```

Authority-Based Hatches. The *Decentralized Labeling Model* (DLM) [27] marks data with a set of principals who owns the information. While executing a program, the program is given *authority*, that is, it is authorized to act on behalf of some set of principals. Declassification simply makes a copy of the released data and marks it with the same set of principals but excludes the authorities.

Similarly to Russo et al. [35], we adapt this idea such that it works on a security lattice of `Principal`s, assign authorities with security levels from the lattice, and let authorities declassify information at that security level.

To model this, we define the abstract data type `Authority` with a data constructor available only to trusted code so that having an instance of `Authority s` corresponds to having the authority of the principal `s`. Notice how assignment of authorities to pieces of code consequently is a part of the trusted code. Now, we instantiate the token hatch builder with a predicate that demands the authority of `s` to declassify information at that level.

```
authHatch : { l, l' : Principal }
         -> (s ** (l = s, Authority s))
         -> Labeled l a -> Labeled l' a
authHatch {l} = tokenHatchBuilder (\s => (l = s, Authority s))
```

That is, `authHatch` makes it possible to declassify information at level `l` to `l'` given an instance of the `Authority l` data type.

Example. Consider the scenario of an online dating service that has the distinguishing feature of allowing its users to specify the visibility of their profiles at a fine-grained level. To achieve this, the service allows users to provide a *discovery agent* that controls their visibility. Consider a user, Bob, whose implementation of the discovery agent takes as input his own profile and the profile of another user, say Alice. The agent returns a possibly side-effectful computation that returns an option type indicating whether Bob wants to be discovered by Alice. If that is the case, a profile is returned by the computation with the information about Bob that he wants Alice to be able to see. When Alice searches for candidate matches, her profile is run against the discovery agents of all candidates and the result is added to her browsing queue.

To implement this dating service, we define the record type `ProfileInfo A` that contains personal information related to principal `A`.

```
record ProfileInfo (A : Principal) where
  constructor MkProfileInfo
  name      : Labeled A String
  gender    : Labeled A String
  birthdate : Labeled A String
  ...
```

The interesting part of the dating service is the implementation of discovery agents. Figure 6 presents a sample discovery agent that matches all profiles with the opposite gender and only releases information about the name and gender. The discovery agent demands the authority of `A` and takes as input two profiles `a : ProfileInfo A` and `b : ProfileInfo B`. The resulting computation security level is `B` so to incorporate information from `a` into the result, declassification is needed. This is achieved by providing `authHatch` with the authority proof of `A`. The discovery agent `sampleDiscoverer` in Fig. 6 unlabels `B`'s gender, declassifies and unlabels `A`'s gender and name, and compares the two genders. If the genders match, a profile with type `ProfileInfo B` only containing the name and gender of `A` is returned. Otherwise, `Nothing` is returned indicating that `A` does not want to be discovered. Notice that `Refl` is the constructor for the built-in equality type in Idris and it is used to construct the proof of equality between principals required by the hatch.

```
sampleDiscoverer : {A, B : Principal}
                -> Authority A
                -> (a : ProfileInfo A)
                -> (b : ProfileInfo B)
                -> DIO B (Maybe (ProfileInfo B))
sampleDiscoverer {A} {B} auth a b =
  do bGender <- unlabel $ gender b
     aGender <- unlabel $ authHatch (A ** (Refl, auth)) (gender a)
     aName <- unlabel $ authHatch (A ** (Refl, auth)) (name a)
     case decEq bGender aGender of
       Yes _ => pure Nothing
       No _  => pure (Just (MkProfileInfo aName aGender "" "" ""))
```

Fig. 6. A discovery agent that matches with all profiles of the opposite gender and only releases the name and gender.

6 Soundness

Recent works [46,47] present a mechanically-verified model of **MAC** and show progress-insensitive noninterference (PINI) for a sequential calculus. We use this work as a starting point and discuss necessary modification in the following. Notice that this work does not consider any declassification mechanisms and neither do we; we leave this as future work.

The proof relies on the *two-steps erasure* technique, an extension of the *term erasure* [21] technique that ensures that the same public output is produced if

secrets are erased before or after program execution. The technique relies on a type-driven erasure function ε_{ℓ_A} on terms and configurations where ℓ_A denotes the attacker security level. A configuration consists of an ℓ-indexed compartmentalized store Σ and a term t. A configuration $\langle \Sigma, t \rangle$ is erased by erasing t and by erasing Σ pointwise, i.e. $\varepsilon_{\ell_A}(\Sigma) = \lambda \ell . \varepsilon_{\ell_A}(\Sigma(\ell))$. On terms, the function essentially rewrites data and computations above ℓ_A to a special $\bullet$ value. The full definition of the erasure function is available in the full version of this paper [15]. From this definition, the definition of low-equivalence of configurations follows.

Definition 1. *Let c_1 and c_2 be configurations. c_1 and c_2 are said to be ℓ_A-equivalent, written $c_1 \approx_{\ell_A} c_2$, if and only if $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2)$.*

After defining the erasure function, the noninterference theorem follows from showing a *single-step simulation* relationship between the erasure function and a small-step reduction relation: erasing sensitive data from a configuration and then taking a step is the same as first taking a step and then erasing sensitive data. This is the content of the following proposition.

Proposition 1. *If $c_1 \approx_{\ell_A} c_2$, $c_1 \rightarrow c_1'$, and $c_2 \rightarrow c_2'$ then $c_1' \approx_{\ell_A} c_2'$.*

The main theorem follows by repeated applications of Proposition 1.

Theorem 1 (PINI). *If $c_1 \approx_{\ell_A} c_2$, $c_1 \Downarrow c_1'$, and $c_2 \Downarrow c_2'$ then $c_1' \approx_{\ell_A} c_2'$.*

Both the statement and the proof of noninterference for DepSec are mostly similar to the ones for **MAC** and available in the full version of this paper [15]. Nevertheless, one has to be aware of a few subtleties.

First, one has to realize that even though dependent types in a language like Idris may depend on data, the data itself is not a part of a value of a dependent type. Recall the type `Vect n Nat` of vectors of length `n` with components of type `Nat` and consider the following program.

```
length : Vect n a -> Nat
length {n = n} xs = n
```

This example may lead one to believe that it is possible to extract data from a dependent type. This is *not* the case. Both `n` and `a` are implicit arguments to the `length` function that the compiler is able to infer. The actual type is

```
length : {n : Nat} -> {a : Type} -> Vect n a -> Nat
```

As a high-level dependently typed functional programming language, Idris is elaborated to a low-level core language based on dependent type theory [9]. In the elaboration process, such implicit arguments are made explicit when functions are defined and inferred when functions are invoked. This means that in the underlying core language, only explicit arguments are given. Our modeling given in the full version of this paper reflects this fact soundly.

Second, to model the extended expressiveness of DepSec, we extend both the semantics and the type system with compile-time pure-term reduction and

higher-order dependent types. These definitions are standard (defined for Idris by Brady [9]) and available in the full version of our paper. Moreover, as types now become first-class terms, the definition of ε_{ℓ_A} has to be extended to cover the new kinds of terms. As before, primitive types are unaffected by the erasure function, but dependent and indexed types, such as the type `DIO`, have to be erased homomorphically, e.g., $\varepsilon_{\ell_A}\ (\texttt{DIO}\ \ell\ \tau : \texttt{Type}) \triangleq \texttt{DIO}\ \varepsilon_{\ell_A}(\ell)\ \varepsilon_{\ell_A}(\tau)$. The intuition of why this is sensible comes from the observation that indexed dependent types considered as terms may contain values that will have to be erased. This is purely a technicality of the proof. If defined otherwise, the erasure function would not commute with capture-avoiding substitution on terms, $\varepsilon_{\ell_A}(t[v/x]) = \varepsilon_{\ell_A}(t)[\varepsilon_{\ell_A}(v)/x]$, which is vital for the remaining proof.

7 Related Work

Security Libraries. The pioneering and formative work by Li and Zdancewic [20] shows how *arrows* [18], a generalization of monads, can provide information-flow control without runtime checks as a library in Haskell. Tsai et al. [45] further extend this work to handle side-effects, concurrency, and heterogeneous labels. Russo et al. [35] eliminate the need for arrows and implement the security library **SecLib** in Haskell based solely on monads. Rather than labeled values, this work introduces a monad which statically label side-effect free values. Furthermore, it presents combinators to dynamically specify and enforce declassification policies that bear a resemblance to the policies that DepSec are able to enforce statically.

The security library **LIO** [41,42] dynamically enforces information-flow control in both sequential and concurrent settings. Stefan et al. [40] extend the security guarantees of this work to also cover exceptions. Similar to this work, Stefan et al. [42] present a simple API for implementing secure conference reviewing systems in **LIO** with support for data-dependent security policies.

Inspired by the design of **SecLib** and **LIO**, Russo [34] introduces the security library **MAC**. The library statically enforces information-flow control in the presence of advanced features like exceptions, concurrency, and mutable data structures by exploiting Haskell's type system to impose flow constraints. Vassena and Russo [46], Vassena et al. [47] show progress-insensitive noninterference for **MAC** in a sequential setting and progress-sensitive noninterference in a concurrent setting, both using the two-steps erasure technique.

The flow constraints enforcing confidentiality of read and write operations in DepSec are identical to those of **MAC**. This means that the examples from **MAC** that do not involve concurrency can be ported directly to DepSec. To the best of our knowledge, data-dependent security policies like the one presented in Sect. 3 cannot be expressed and enforced in **MAC**, unlike **LIO** that allows such policies to be enforced dynamically. DepSec allows for such security policies to be enforced statically. Moreover, Russo [34] does not consider declassification. To address the static limitations of **MAC**, **HLIO** [11] takes a hybrid approach by exploiting advanced features in Haskell's type-system like singleton types and constraint polymorphism. Buiras et al. [11] are able to statically enforce

information-flow control while allowing selected security checks to be deferred until run-time.

Dependent Types for Security. Several works have considered the use of dependent types to capture the nature of data-dependent security policies. Zheng and Myers [51,52] proposed the first dependent security type system for dealing with dynamic changes to runtime security labels in the context of Jif [29], a full-fledged IFC-aware compiler for Java programs, where similar to our work, operations on labels are modeled at the level of types. Zhang et al. [50] use dependent types in a similar fashion for the design of a hardware description language for timing-sensitive information-flow security.

A number of functional languages have been developed with dependent type systems and used to encode value-dependent information flow properties, e.g. Fine [43]. These approaches require the adoption of entirely new languages and compilers where DepSec is embedded in an already existing language. Morgenstern and Licata [25] encode an authorization and IFC-aware programming language in Agda. However, their encoding does not consider side-effects. Nanevski et al. [30] use dependent types to verify information flow and access control policies in an interactive manner.

Lourenço and Caires [23] introduce the notion of *dependent information-flow types* and propose a *fine-grained* type system; every value and function have an associated security level. Their approach is different to the *coarse-grained* approach taken in our work where only some computations and values have associated security labels. Rajani and Garg [33] show that both approaches are equally expressive for static IFC techniques and Vassena et al. [48] show the same for dynamic IFC techniques.

Principles for Information Flow. Bastys et al. [6] put forward a set of informal principles for information flow security definitions and enforcement mechanisms: *attacker-driven security, trust-aware enforcement, separation of policy annotations and code, language-independence, justified abstraction, and permissiveness.*

DepSec follows the principle of trust-aware enforcement, as we make clear the boundary between the trusted and untrusted components in the program. Additionally, the design of our declassification mechanism follows the principle of separation of policy annotations and code. The use of dependent types increases the permissiveness of our enforcement as we discuss throughout the paper. While our approach is not fully language-independent, we posit that the approach may be ported to other programming languages with general-purpose dependent types.

Declassification Enforcement. Our hatch builders are reminiscent of downgrading policies of Li and Zdancewic [19]. For example, similar to them, DepSec's declassification policies naturally express the idea of *delimited release* [36] that provides explicit characterization of the declassifying computation. Here, DepSec's policies can express a broad range of policies that can be expressed

through predicates, an improvement over simple expression-based enforcement mechanisms for delimited release [4,5,36].

An interesting point in the design of declassification policies is *robust declassification* [49] that demands that untrusted components must not affect information release. *Qualified robustness* [2,28] generalizes this notion by giving untrusted code a limited ability to affect information release through the introduction of an explicit endorsement operation. Our approach is orthogonal to both notions of robustness as the intent is to let the untrusted components declassify information but only under very controlled circumstances while adhering to the security policy.

8 Conclusion and Future Work

In this paper, we have presented DepSec – a library for statically enforced information-flow control in Idris. Through several case studies, we have showcased how the DepSec primitives increase the expressiveness of state-of-the-art information-flow control libraries and how DepSec matches the expressiveness of a special-purpose dependent information-flow type system on a key example. Moreover, the library allows programmers to implement policy-parameterized functions that abstract over the security policy while retaining precise security levels.

By taking ideas from the literature and by exploiting dependent types, we have shown powerful means of specifying statically enforced declassification policies related to *what*, *who*, and *when* information is released. Specifically, we have introduced the notion of predicate hatch builders and token hatch builders that rely on the fulfillment of predicates and possession of tokens for declassification. We have also shown how the `ST` monad [10] can be used to limit hatch usage statically.

Finally, we have discussed the necessary means to show progress-insensitive noninterference in a sequential setting for a dependently typed information-flow control library like DepSec.

Future Work. There are several avenues for further work. Integrity is vital in many security policies and is not considered in **MAC** nor DepSec. It will be interesting to take integrity and the presence of concurrency into the dependently typed setting and consider internal and termination covert channels as well. It also remains to prove our declassification mechanisms sound. Here, attacker-centric epistemic security conditions [3,16] that intuitively express many declassification policies may be a good starting point.

Acknowledgements. Thanks are due to Mathias Vorreiter Pedersen, Bas Spitters, Alejandro Russo, and Marco Vassena for their valuable insights and the anonymous reviewers for their comments on this paper. This work is partially supported by DFF project 6108-00363 from The Danish Council for Independent Research for the Natural Sciences (FNU), Aarhus University Research Foundation, and the Concordium Blockchain Research Center, Aarhus University, Denmark.

References

1. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 333–348. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88313-5_22
2. Askarov, A., Myers, A.C.: Attacker control and impact for confidentiality and integrity. Log. Methods Comput. Sci. **7**(3) (2011). https://doi.org/10.2168/LMCS-7(3:17)2011
3. Askarov, A., Sabelfeld, A.: Gradual release: unifying declassification, encryption and key release policies. In: 2007 IEEE Symposium on Security and Privacy (S&P 2007), Oakland, California, USA, 20–23 May 2007, pp. 207–221. IEEE Computer Society (2007). https://doi.org/10.1109/SP.2007.22
4. Askarov, A., Sabelfeld, A.: Localized delimited release: combining the what and where dimensions of information release. In: Hicks, M.W. (ed.) Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, San Diego, California, USA, 14 June 2007, pp. 53–60. ACM (2007). https://doi.org/10.1145/1255329.1255339
5. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, 8–10 July 2009, pp. 43–59. IEEE Computer Society (2009). https://doi.org/10.1109/CSF.2009.22
6. Bastys, I., Piessens, F., Sabelfeld, A.: Prudent design principles for information flow control. In: Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, pp. 17–23. ACM (2018)
7. Bell, D.E., La Padula, L.J.: Secure computer system: unified exposition and multics interpretation. Technical report. MITRE Corp., Bedford, MA (1976)
8. Brady, E.: IDRIS—systems programming meets full dependent types. In: Jhala, R., Swierstra, W. (eds.) Proceedings of the 5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011, Austin, TX, USA, 29 January 2011, pp. 43–54. ACM (2011). https://doi.org/10.1145/1929529.1929536
9. Brady, E.: Idris, a general-purpose dependently typed programming language: design and implementation. J. Funct. Program. **23**(5), 552–593 (2013). https://doi.org/10.1017/S095679681300018X
10. Brady, E.: State machines all the way down, January 2016. http://idris-lang.org/drafts/sms.pdf
11. Buiras, P., Vytiniotis, D., Russo, A.: HLIO: mixing static and dynamic typing for information-flow control in Haskell. In: Fisher, K., Reppy, J.H. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, 1–3 September 2015, pp. 289–301. ACM (2015). https://doi.org/10.1145/2784731.2784758
12. Chapman, R., Hilton, A.: Enforcing security and safety models with an information flow analysis tool. In: McCormick, J.W., Sward, R.E. (eds.) Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies 2004, Atlanta, GA, USA, 14 November 2004, pp. 39–46. ACM (2004). https://doi.org/10.1145/1032297.1032305
13. Coquand, T., Huet, G.P.: The calculus of constructions. Inf. Comput. **76**(2/3), 95–120 (1988). https://doi.org/10.1016/0890-5401(88)90005-3

14. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977). https://doi.org/10.1145/359636.359712
15. Gregersen, S., Thomsen, S.E., Askarov, A.: A dependently typed library for static information-flow control in Idris (2019). arXiv:1902.06590
16. Halpern, J.Y., O'Neill, K.R.: Secrecy in multiagent systems. ACM Trans. Inf. Syst. Secur. **12**(1), 5:1–5:47 (2008). https://doi.org/10.1145/1410234.1410239
17. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: tracking information flow in Javascript and its APIs. In: Cho, Y., Shin, S.Y., Kim, S., Hung, C., Hong, J. (eds.) Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea, 24–28 March 2014, pp. 1663–1671. ACM (2014). https://doi.org/10.1145/2554850.2554909
18. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**(1–3), 67–111 (2000). https://doi.org/10.1016/S0167-6423(99)00023-4
19. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, 12–14 January 2005, pp. 158–170. ACM (2005). https://doi.org/10.1145/1040305.1040319
20. Li, P., Zdancewic, S.: Encoding information flow in Haskell. In: 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006), Venice, Italy, 5–7 July 2006, p. 16. IEEE Computer Society (2006). https://doi.org/10.1109/CSFW.2006.13
21. Li, P., Zdancewic, S.: Arrows for secure information flow. Theor. Comput. Sci. **411**(19), 1974–1994 (2010). https://doi.org/10.1016/j.tcs.2010.01.025
22. Liu, J., George, M.D., Vikram, K., Qi, X., Waye, L., Myers, A.C.: Fabric: a platform for secure distributed computation and storage. In: Matthews, J.N., Anderson, T.E. (eds.) Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, 11–14 October 2009, pp. 321–334. ACM (2009). https://doi.org/10.1145/1629575.1629606
23. Lourenço, L., Caires, L.: Dependent information flow types. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, 15–17 January 2015, pp. 317–328. ACM (2015). https://doi.org/10.1145/2676726.2676994
24. Martin-Löf, P., Sambin, G.: Intuitionistic Type Theory, vol. 9. Bibliopolis, Naples (1984)
25. Morgenstern, J., Licata, D.R.: Security-typed programming within dependently typed programming. In: Hudak, P., Weirich, S. (eds.) Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, 27–29 September 2010, pp. 169–180. ACM (2010). https://doi.org/10.1145/1863543.1863569
26. Myers, A.C.: JFlow: practical mostly-static information flow control. In: Appel, A.W., Aiken, A. (eds.) Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1999, San Antonio, TX, USA, 20–22 January 1999, pp. 228–241. ACM (1999). https://doi.org/10.1145/292540.292561
27. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. **9**(4), 410–442 (2000). https://doi.org/10.1145/363516.363526

28. Myers, A.C., Sabelfeld, A., Zdancewic, S.: Enforcing robust declassification. In: 17th IEEE Computer Security Foundations Workshop (CSFW-17 2004), Pacific Grove, CA, USA, 28–30 June 2004, pp. 172–186. IEEE Computer Society (2004). https://doi.org/10.1109/CSFW.2004.9
29. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif 3.0: Java information flow, July 2006
30. Nanevski, A., Banerjee, A., Garg, D.: Verification of information flow and access control policies with dependent types. In: 32nd IEEE Symposium on Security and Privacy, S&P 2011, Berkeley, California, USA, 22–25 May 2011, pp. 165–179. IEEE Computer Society (2011). https://doi.org/10.1109/SP.2011.12
31. Nordström, B., Petersson, K., Smith, J.M.: Programming in Martin-Löf's Type Theory: An Introduction. Clarendon Press, New York (1990)
32. Norell, U.: Towards a Practical Programming Language Based on Dependent Type Theory, vol. 32. Citeseer (2007)
33. Rajani, V., Garg, D.: Types for information flow control: labeling granularity and semantic models. In: 31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, 9–12 July 2018, pp. 233–246. IEEE Computer Society (2018). https://doi.org/10.1109/CSF.2018.00024
34. Russo, A.: Functional pearl: two can keep a secret, if one of them uses Haskell. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, 1–3 September 2015, pp. 280–288 (2015). https://doi.org/10.1145/2784731.2784756
35. Russo, A., Claessen, K., Hughes, J.: A library for light-weight information-flow security in Haskell. In: Gill, A. (ed.) Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008, pp. 13–24. ACM (2008). https://doi.org/10.1145/1411286.1411289
36. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003). https://doi.org/10.1109/JSAC.2002.806121
37. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 174–191. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-37621-7_9
38. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: 18th IEEE Computer Security Foundations Workshop (CSFW-18 2005), Aix-en-Provence, France, 20–22 June 2005, pp. 255–269. IEEE Computer Society (2005). https://doi.org/10.1109/CSFW.2005.15
39. Simonet, V.: Flow Caml in a nutshell. In: Hutton, G. (ed.) Proceedings of the first APPSEM-II Workshop, Nottingham, United Kingdom, March 2003
40. Stefan, D., Mazières, D., Mitchell, J.C., Russo, A.: Flexible dynamic information flow control in the presence of exceptions. J. Funct. Program. **27**, e5 (2017). https://doi.org/10.1017/S0956796816000241
41. Stefan, D., Russo, A., Buiras, P., Levy, A., Mitchell, J.C., Mazières, D.: Addressing covert termination and timing channels in concurrent information flow systems. In: Thiemann, P., Findler, R.B. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP 2012, Copenhagen, Denmark, 9–15 September 2012, pp. 201–214. ACM (2012). https://doi.org/10.1145/2364527.2364557
42. Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in Haskell. In: Claessen, K. (ed.) Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011, pp. 95–106. ACM (2011). https://doi.org/10.1145/2034675.2034688

43. Swamy, N., Chen, J., Chugh, R.: Enforcing stateful authorization and information flow policies in Fine. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 529–549. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_28
44. Swamy, N., Chen, J., Fournet, C., Strub, P., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, Tokyo, Japan, 19–21 September 2011, pp. 266–278. ACM (2011). https://doi.org/10.1145/2034773.2034811
45. Tsai, T., Russo, A., Hughes, J.: A library for secure multi-threaded information flow in Haskell. In: 20th IEEE Computer Security Foundations Symposium, CSF 2007, Venice, Italy, 6–8 July 2007, pp. 187–202. IEEE Computer Society (2007). https://doi.org/10.1109/CSF.2007.6
46. Vassena, M., Russo, A.: On formalizing information-flow control libraries. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, 24 October 2016, pp. 15–28 (2016). https://doi.org/10.1145/2993600.2993608
47. Vassena, M., Russo, A., Buiras, P., Waye, L.: MAC a verified static information-flow control library. J. Log. Algebr. Methods Program. **95**, 148–180 (2018). http://www.sciencedirect.com/science/article/pii/S235222081730069X
48. Vassena, M., Russo, A., Garg, D., Rajani, V., Stefan, D.: From fine- to coarse-grained dynamic information flow control and back. PACMPL **3**(POPL), 76:1–76:31 (2019). https://doi.org/10.1145/2694344.2694372
49. Zdancewic, S., Myers, A.C.: Robust declassification. In: 14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), Cape Breton, Nova Scotia, Canada, 11–13 June 2001, pp. 15–23. IEEE Computer Society (2001). https://doi.org/10.1109/CSFW.2001.930133
50. Zhang, D., Wang, Y., Suh, G.E., Myers, A.C.: A hardware design language for timing-sensitive information-flow security. In: Özturk, Ö., Ebcioglu, K., Dwarkadas, S. (eds.) Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, 14–18 March 2015, pp. 503–516. ACM (2015). https://doi.org/10.1145/2694344.2694372
51. Zheng, L., Myers, A.C.: Dynamic security labels and noninterference (extended abstract). In: Dimitrakos, T., Martinelli, F. (eds.) Formal Aspects in Security and Trust. IFIP, vol. 173, pp. 27–40. Springer, Boston (2005). https://doi.org/10.1007/0-387-24098-5_3
52. Zheng, L., Myers, A.C.: Dynamic security labels and static information flow control. Int. J. Inf. Secur. **6**(2–3), 67–84 (2007). https://doi.org/10.1007/s10207-007-0019-9