Encoding Secure Information Flow with Restricted Delegation and Revocation in Haskell

Doaa Hassan *
Computers and Systems Department
National Telecommunication Institute
Cairo, Egypt
doaa@nti.sci.eg

Amr Sabry
School of Informatics and Computing
Indiana University
Bloomington, Indiana
sabry@indiana.edu

Abstract

Distributed applications typically involve many components, each with unique security and privacy requirements. Such applications require fine-grained access control mechanisms that allow dynamic *delegation* and *revocation* of access rights. Embedding such domain-specific requirements in a functional language like Haskell puts all the expressiveness of the host language at the disposal of the domain user. Using a custom monad, we design and implement an embedded Haskell library that embraces the *decentralized label model*, allowing mutually-distrusting principals to express individual confidentiality and integrity policies. Our language includes first-class references, higher-order functions, declassification and endorsement of policies, and user authority in the presence of global unrestricted delegation. Then, building on previous work by the first author, we extend the language to enable fine-grained dynamic delegation and revocation of access rights. The resulting language generalizes, extends, and simplifies various libraries for expressing and reasoning about information flow.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.4.6 [*Security and Protection*]: Information flow controls; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives

Keywords Information Flow Security; Monad; Library.

1. Introduction

In the emerging paradigm of cloud computing, organizations offload their software services to providers from whom users can rent virtual machines to access the services. This framework introduces several opportunities for unauthorized access to user's resources uploaded to the cloud. As several researchers argue [1, 8, 9], one of the core issues is that of *restricted delegation and revocation* of access rights. The main idea of restricted delegation and revocation is to temporarily allow information to flow to a predefined chain of principals, while retaining the right to revoke it at any time. This model of delegation was investigated in access control policies [30], but was not extended to information flow security policies. In particular, existing security typed programming languages that support delegation (e.g., Jif [10, 22], and flowCaml [26]) do not directly support restricted delegation and revocation.

Building on previous work by the first author [16, 17], we design – and implement as a Haskell library – a language with restricted delegation and revocation facilities. The original work produced a small stand-alone first-order language, RDR, to formalize the semantics of restricted delegation and revocation. Our current work generalizes, extends, and simplifies the previous RDR design and implementation along several dimensions. First, the previous work lacked, in both design and implementation, many of the higher-order abstractions that are needed for realistic distributed programming (e.g., first-class references, higher-order functions, input/output channels, etc.). Although the secure information flow semantics of these constructs is quite subtle (see e.g. [31]), our monadic embedding in Haskell allows us to directly inherit these constructs with little overhead. In addition, although RDR is based on the *decentralized label model* (DLM) [23] that allows (distributed) principals to state and enforce individual confidentiality and integrity security policies, its security model did not support code running under the authority of distinct (and distributed) principals. Finally, RDR expressed the semantics of delegation and revocation at the level of principals instead of the more expressive level of policies. Instead, our design keeps track of the dynamic delegations and revocations using a notion of a "label chain" that is more consistent with the DLM. One of the key design decisions of RDR that we retain and exploit is that the delegation and revocation actions are specified at the fine-grained level of individual values. Typically, one would not expect each individual value to have its own access control policy as this would be prohibitively costly to implement. However, if used for shared objects of interest, specifying policies at the level of data elements enables us to partition data objects and enforce different access control policies for the same object. For example, given a personal health record, policies that allow patients, doctors, pharmacists, and insurance agents, to access different parts of the record can be enforced simultaneously.

In the next section, we review the DLM as it is central to the ideas of the paper. Sec. 3 builds on the DLM and introduces the monadic embedding of a rich language for secure information flow with first-class references and higher-order procedures. Sec. 4 extends the DLM with the notion of label chains and then extends the language with facilities for restricted delegation and revocation

*This work was performed while the author was a visiting scholar at the School of Informatics and Computing, Indiana University.

FPCDSL '13, September 22, 2013, Boston, MA, USA.

ACM 978-1-4503-2380-2/13/09. . . $15.00.
http://dx.doi.org/10.1145/2505351.2505354

and provides examples illustrating the use and expressiveness of the language. We then review related work (Sec. 5) and conclude.

2. Labels

There are several security-typed programming languages that permit delegation and revocation relationships between principals. For example, a general form of delegation is present in Jif [10, 22]. The delegation relation is derived from a principal hierarchy which is a partial order that determines which principals can *act for* other principals. This partial order may change during the execution of a program but is assumed to be mostly static. Delegation is also supported in flowCaml [26], another well-known security-typed programming language. Following the *decentralized label model* (DLM), the fundamental abstraction in many of these approaches, including ours, is that of a *label*. A label is an entity that attaches to locations, channels, and values to specify the principals that "own" them and their security policies for the entity. We review the conventional model in some detail, in order to set up the stage for our extension to dynamically support restricted forms of delegation and revocation.

2.1 Decentralized Label Model (DLM)

In a distributed environment, e.g., the cloud, we cannot assume the existence of a central authority that would manage all the computing resources. The DLM allows each resource or collection of resources to be managed by a different *principal*. This principal *owns* the resources: it specifies the security requirements via a set of policies and is responsible for enforcing them. Different principals are not assumed to trust each other but explicit delegations are possible among principals to specify who can *act for* whom. Generally speaking, a principal p who can act for another principal q has *all* of the rights and responsibilities of q. It is possible to refine the principals into various roles and correspondingly refine the "acts-for" relationship to only grant specific rights. Indeed it is our goal, to model one such relationship: specifically, we are interested in situations in which a principal p can delegate access to a resource to a chain of principals $q_1, \ldots, q_n$ subject to the following constraints: (i) initially q_1 is the active principal in the chain, (ii) the active principal can access the data and can delegate it to the next principal in the chain and *only* to that next principal, (iii) the entire chain or any subchain therein may be revoked at any time.

To take full advantage of the DLM, each code fragment is assumed to run under the authority of a distinct principal. During execution, only those policies owned by principals who can act for the current user are "activated." Other policies are *ignored*. This is a somewhat counter-intuitive aspect as it would allow code running under the authority of a principal like *bob* to read the private data of another principal like *amy* even if *amy* specifically attaches a private policy to her data. The informal justification is that the DLM infrastructure does not allow the code running under *bob*'s authority to do anything "harmful" with this data. Formally, a security violation only occurs when the data leaves the entire system via an output channel. Principal *bob* could certainly write the data to a channel owned by *amy* but this would not constitute a security violation. If, however, *bob* attempted to write the data to a public channel, he would need to *declassify* the data by removing *amy*'s private policy which, fortunately, he cannot do.

2.2 Confidentiality Policies

In our presentation, a label ℓ consists of exactly two policies: a *confidentiality policy R* and an *integrity policy W*. A *confidentiality policy* or *reader policy* is defined as follows:

$$(\textit{Reader policies}) \quad R \quad ::= \quad \mathsf{rp}\{o : p\} \mid R \sqcup R \mid R \sqcap R$$

A basic policy $\mathsf{rp}\{o : p\}$ states that principal o is the owner of the data in question (and hence an implicit reader) and that principal p is explicitly given the permission to also read the data. Policies with multiple owners and/or readers can be expressed via $\sqcup$ and $\sqcap$. Intuitively, $\sqcup$ takes the *intersection* of the readers and $\sqcap$ takes the *union* of the readers but, as formalized below, the semantics is much more subtle and depends on the authority under which the current code is executing and on existing delegation relationships. We now turn our attention to each of these points.

Unrestricted delegation. The DLM framework is parametrized by a partial order relationship among principals that indicates "unrestricted" delegation relationships among principals. In the common case, a principal p delegates *all* of its authority, unconditionally, to another principal q. The delegation relationship is not assumed to be static but is coarse-grained and assumed to change infrequently, e.g., when a new doctor is employed at a hospital. In our presentation, we assume a reflexive transitive relation $\succcurlyeq$ among principals that summarizes which principals can act for which other principals due to these unrestricted delegations. It is often convenient to assume the existence of two special principals: $\bot$ and $\top$. The principal $\bot$ allows any principal to act for it, and the principal $\top$ can act for any principal. In actual applications, i.e., when the set of principals is known, the principal $\top$ models a trusted principal, e.g., the OS kernel, and the principal $\bot$ can be thought of as a placeholder for an actual principal with minimal rights.

Semantics of confidentiality policies. We can now formalize the precise set of readers associated with a confidentiality policy. As briefly mentioned above, the set of readers depends on the authority under which the current code is executing. In other words, policy owners are only able to enforce their policies if they can act for the user under whose authority the current code is executing. Otherwise, if the policy owners cannot act for the current user, their policies are *ignored*. Formally, given an application involving principals drawn from a set P, and given that the current code is running under the authority of a particular principal $u \in P$, the set of readers allowed by a confidentiality policy is defined as follows:

$$\begin{array}{rcl} \mathit{readers}_u^P(\mathsf{rp}\{o : p\}) & = & \\ \quad \begin{cases} \{q \mid q \in P,\ q \succcurlyeq o \text{ or } q \succcurlyeq p\} & o \succcurlyeq u \\ P & \text{otherwise} \end{cases} & & \end{array}$$

$$\mathit{readers}_u^P(R_1 \sqcup R_2) \quad = \quad \mathit{readers}_u^P(R_1) \cap \mathit{readers}_u^P(R_2)$$

$$\mathit{readers}_u^P(R_1 \sqcap R_2) \quad = \quad \mathit{readers}_u^P(R_1) \cup \mathit{readers}_u^P(R_2)$$

The most subtle clause is the first. If the code is running under the authority of principal u then we first decide if the owner of the policy o can act for u. If not, everyone is allowed to be a reader which would be the case if no policy existed. If, however, o can act for u, then the policy is enforced: only the owner o, the explicit reader p, and those who can act for them would be allowed to access the data. Notice that if $u \succcurlyeq u'$, then for every policy R, we have that $\mathit{readers}_{u'}^P(R) \subseteq \mathit{readers}_u^P(R)$. To see why, consider the case in which the policy is $\mathsf{rp}\{o : p\}$ and o acts for u' but not for u. (The reverse is impossible because the relation $\succcurlyeq$ is transitive, i.e., if $o \succcurlyeq u$ then also $o \succcurlyeq u'$.) In this case, $\mathit{readers}_u^P(\mathsf{rp}\{o : p\} = P$, the set of all principals, but the corresponding set from the perspective of u' only contains the principals that can act for o and p.

The definition suggests a natural lattice-ordering for confidentiality policies:

- The partial ordering $\sqsubseteq_u^R$ is defined via the superset relation on the sets of readers. Note that the partial order relation is parametrized by the current user u, i.e., that each user "sees" a different partial order among policies.

- The bottom element of the lattice $\bot_R$ is the policy $\mathsf{rp}\{\bot : \bot\}$ where $\bot$ is the principal who allows every other principal to act on its behalf. The set of readers for this policy is the entire set of principals and hence this policy is the *least restrictive* confidentiality policy.
- The top element of the lattice $\top_R$ is the policy $\mathsf{rp}\{\top : \top\}$ where $\top$ is the principal who can act for every other principal. The set of readers for this policy is the empty set and hence this policy is the *most restrictive* confidentiality policy.
- The join and meet of the lattice elements are given by $\sqcup$ and $\sqcap$ respectively.

Thus, one can visualize the lattice with the least restrictive policy at the bottom, the most restrictive policy at the top, and view the join operation as moving up the lattice and the meet operation as moving down the lattice.

2.3 Integrity Policies

Confidentiality policies express which principals are permitted to read a given data element. Thus the more restrictive a confidentiality policy is, the fewer readers it allows. Integrity policy express which principals "trust" the data. Thus, dual to the ordering on confidentiality policies, an integrity policy that states that everyone must trust the data is the most restrictive, and an integrity policy that requires no one to trust the data is the least restrictive. Formally, the semantics of integrity policies is similar to the semantics in the previous section with some of the key relationships reversed.

Integrity policies, also commonly called *writer policies*, are defined as follows:

$$(\textit{Writer policies}) \quad W \quad ::= \quad \mathsf{wp}\{o : p\} \mid W \sqcup W \mid W \sqcap W$$

A basic policy $\mathsf{wp}\{o : p\}$ states that principal o is the owner of the data in question (and hence an implicit writer) and that principal p is explicitly given the permission to also write the data. As before, policies with multiple owners and/or readers can be expressed via $\sqcup$ and $\sqcap$, but in this case $\sqcup$ takes the *union* of the writers and $\sqcap$ takes the *intersection* of the writers. Formally, we have:

$$writers_u^P(\mathsf{wp}\{o : p\}) \quad = \quad \begin{cases} \{q \mid q \in P,\ q \succcurlyeq o \text{ or } q \succcurlyeq p\} & o \succcurlyeq u \\ \emptyset & \text{otherwise} \end{cases}$$

$$writers_u^P(W_1 \sqcup W_2) \quad = \quad writers_u^P(W_1) \cup writers_u^P(W_2)$$

$$writers_u^P(W_1 \sqcap W_2) \quad = \quad writers_u^P(W_1) \cap writers_u^P(W_2)$$

Again, the most subtle clause is the first. If the code is running under the authority of principal u then we first decide if the owner of the policy o can act for u. If not, we act as if the policy did not exist and do not require anyone to trust the data. If, however, o can act for u, then the policy is enforced and, as before, only the owner o, the explicit writer p, and those who can act for them would be allowed to write the data. The lattice properties in this case are as follows:

- The partial ordering $\sqsubseteq_u^W$ is defined via the subset relation on the sets of writers.
- The bottom element of the lattice $\bot_W$ is the policy $\mathsf{wp}\{\top : \top\}$ where $\top$ is the principal who can act for every other principal. The set of writers for this policy is the empty set and hence this policy is the *least restrictive* integrity policy which requires no one to trust the data.
- The top element of the lattice $\top_W$ is the policy $\mathsf{wp}\{\bot : \bot\}$ where $\bot$ is the principal who allows every other principal to act on its behalf. The set of writers for this policy is the set of all principals and hence this policy is the *most restrictive* integrity policy which requires everyone to trust the data.
- The join and meet of the lattice elements are given by $\sqcup$ and $\sqcap$ respectively.

Thus, given these conventions, one can also visualize this lattice with the least restrictive policy at the bottom, the most restrictive policy at the top, and view the join operation as moving up the lattice and the meet operation as moving down the lattice.

2.4 Labels

As mentioned earlier, conventional labels in the DLM consist of a pair of a confidentiality policy and an integrity policy. We denote such labels ℓ using pairs (R, W). These labels form a lattice with bottom $\bot_L$ and top $\top_L$ which is the product lattice of the confidentiality and integrity lattices described above. In more detail, the label $\ell_1 \sqcup \ell_2$ is more restrictive than either ℓ_1 or ℓ_2. Computationally, if a value v_1 has label ℓ_1 and a value v_2 has label ℓ_2 and we calculate $v_1 \oplus v_2$ for some binary operation $\oplus$, the resulting label is $\ell_1 \sqcup \ell_2$. The reason is that access to the resulting value can be granted only if both the policies for the individual components v_1 and v_2 are enforced. Conversely, the label $\ell_1 \sqcap \ell_2$ is more permissive than either ℓ_1 or ℓ_2. Computationally, consider a value v labeled ℓ_1 by one principal and labeled ℓ_2 by another principal. The net effect is that only the least restrictive policies can be enforced. (This could be the situation if a principal restricts access to a value, but his boss, who can act for him, declassifies the value. The net effect should be that the value is declassified.)

Example. Consider a simple situation inspired by the common tax preparer example [23]. There are three principals *bob*, *taxUser*, and *taxPreparer* where it is assumed that *taxUser* can act for *bob* but that *taxUser* and *taxPreparer* do not trust each other. Now consider the following three policies:

- Private *taxUser* policy: $p_1 = \mathsf{rp}\{\textit{taxUser} : \textit{taxUser}\}$;
- Private *taxPreparer* policy: $p_2 = \mathsf{rp}\{\textit{taxPreparer} : \textit{taxPreparer}\}$;
- Policy for data calculated using *bob*'s private data and *taxPreparer* private algorithms: $p = p_1 \sqcup p_2$.

The policy p is viewed differently by each of the three principals. From the perspective of both *bob* and *taxUser*, policy p only allows *taxUser* as a reader. However, from the perspective of *taxPreparer*, the same policy only allows *taxPreparer* as a reader.

3. A Monad for Secure Information-Flow

We begin our presentation of the Haskell embedding by first considering a language without restricted delegation and revocation. Our approach, like other similar efforts (e.g. [25, 27]), is to dynamically encode the information flow using Haskell terms. The main reason is to accommodate the dynamic policies resulting from runtime restricted delegation and revocation actions. A natural approach is to define a new abstract type of "protected values" whose access control and integrity is enforced by a monadic sublanguage.

The interface of our monad RDR is included below. A protected value is internally represented as a plain value and a label but that representation is hidden from the user. The main point of the RDR monad is to control access to protected values. As we explain below, it provides a way to create protected values using the method tagM; once a value has been tagged, there is no way to manipulate it except via the operations in the RDR monad. The only method that could be used to produce the underlying value is runM, which explicitly declassifies the value before releasing it.

```
newtype P a -- protected values are abstract
```

$$\frac{\rho \vdash (e, \emptyset) \mapsto (V, \sigma) \quad \rho \vdash (\mathsf{declassifyM}\ V\ \bot_L, \sigma) \mapsto (\langle v, \bot_L\rangle, \sigma')}{\mathsf{runM}\ \rho\ e \mapsto v} \qquad \frac{}{(\succcurlyeq, u, pc) \vdash (\mathsf{returnM}\ \langle v, \ell\rangle, \sigma) \mapsto (\langle v, \ell \sqcup pc\rangle, \sigma)}$$

$$\frac{\rho \vdash (e, \sigma) \mapsto (V, \sigma') \qquad \rho \vdash (c[V/a], \sigma') \mapsto (V', \sigma'')}{\rho \vdash (\mathbf{do}\ a \leftarrow e; c, \sigma) \mapsto (V', \sigma'')} \qquad \frac{}{\rho \vdash (\mathsf{tagM}\ \ell\ v, \sigma) \mapsto (\langle v, \ell\rangle, \sigma)}$$

$$\frac{}{\rho \vdash (\langle v_1, \ell_1\rangle \oplus \langle v_2, \ell_2\rangle, \sigma) \mapsto (\langle \oplus(v_1, v_2), \ell_1 \sqcup \ell_2\rangle, \sigma)}$$

$$\frac{x \notin dom(\sigma) \qquad pc \sqcup \ell_x \sqsubseteq_u \ell_v}{(\succcurlyeq, u, pc) \vdash (\mathsf{newM}\ \ell_x\ \langle v, \ell_v\rangle, \sigma) \mapsto (\langle x, pc\rangle, \sigma[x \leftarrow \langle v, \ell_x\rangle])}$$

$$\frac{\sigma(x) = \langle v, \ell_v\rangle}{\rho \vdash (\mathsf{readM}\ \langle x, \ell_x\rangle, \sigma) \mapsto (\langle v, \ell_v \sqcup \ell_x\rangle, \sigma)} \qquad \frac{\sigma(x) = \langle v', \ell_s\rangle \qquad pc \sqcup \ell_x \sqsubseteq_u \ell_v \qquad pc \sqcup \ell_v \sqsubseteq_u \ell_s}{(\succcurlyeq, u, pc) \vdash (\mathsf{writeM}\ \langle x, \ell_x\rangle\ \langle v, \ell_v\rangle, \sigma) \mapsto \sigma[x \leftarrow \langle v, \ell_s\rangle]}$$

$$\frac{i = v?\ 1 : 2 \qquad (\succcurlyeq, u, pc \sqcup \ell_v) \vdash (e_i, \sigma) \mapsto (V, \sigma')}{(\succcurlyeq, u, pc) \vdash (\mathsf{ifM}\ \langle v, \ell_v\rangle\ e_1\ e_2, \sigma) \mapsto (V, \sigma')} \qquad \frac{u \succcurlyeq u' \qquad (\succcurlyeq, u', pc) \vdash (e, \sigma) \mapsto (V, \sigma')}{(\succcurlyeq, u, pc) \vdash (\mathsf{localUserM}\ u'\ e, \sigma) \mapsto (V, \sigma')}$$

$$\frac{\begin{array}{c} f = \mathsf{proc}\{pc', u'\}(a : \ell_a) : \ell_r.e \qquad pc \sqcup \ell_f \sqsubseteq_u pc' \qquad u \succcurlyeq u' \qquad pc \sqcup \ell_v \sqsubseteq_u \ell_a \\ (\succcurlyeq, u', pc') \vdash (e[\langle v, \ell_a\rangle/a], \sigma) \mapsto (\langle w, \ell_w\rangle, \sigma') \qquad pc' \sqcup \ell_w \sqsubseteq_{u'} \ell_r \end{array}}{(\succcurlyeq, u, pc) \vdash (\mathsf{callM}\ \langle f, \ell_f\rangle\ \langle v, \ell_v\rangle, \sigma) \mapsto (\langle w, \ell_r\rangle, \sigma')}$$

$$\frac{R_v \sqsubseteq_u^R R \sqcup \mathsf{rp}\{u : \top\} \qquad W \sqcap \mathsf{wp}\{u : \bot\} \sqsubseteq_u^W W_v}{(\succcurlyeq, u, pc) \vdash (\mathsf{declassifyM}\ \langle v, (R_v, W_v)\rangle\ (R, W), \sigma) \mapsto (\langle v, (R, W)\rangle, \sigma)}$$

Figure 1. Semantics of standard constructs.

```
type Env = (actsFor, Principal , Label)

type R a = IORef (P a)
data Proc m a b = Proc {
       pcL  :: Label,
       auth :: Principal ,
       argL :: Label,
       resL :: Label,
       code :: P a → m (P b)
}

class  (Monad m, MonadIO m, MonadReader Env m) ⇒
  M m where
  runM         :: Env → m (P a) → IO a
  returnM      :: P a → m (P a)
  localUserM   :: Principal  → m (P a) → m (P a)
  tagM         :: Label → a → m (P a)
  binopM       :: (a → b → c) → P a → P b → m (P c)
  ifM          :: P Bool → m (P a) → m (P a) → m (P a)
  callM        :: P (Proc m a b) → P a → m (P b)
  newM         :: Label → P a → m (P (R a))
  readM        :: P (R a) → m (P a)
  writeM       :: P (R a) → P a → m (P ())
  declassifyM  :: P a → Label → m (P a)

newtype RDR a = RDR { runRDR :: ReaderT Env IO a }
instance  Monad RDR
instance  MonadIO RDR
instance  MonadReader Env RDR
instance  M RDR
```

The RDR monad is built by layering a Reader monad over the IO monad. The underlying IO monad provides first-class references, channels, and input-output operations. The Reader monad gives access to three parameters: the relation $\succcurlyeq$ which gives the global delegation relationships among principals, the principal under whose authority the code is executing, and the control label which summarizes information that could be extracted from the knowledge that execution reached this point. As mentioned above, the main entity of interest is that of a "protected value"; except for constants and the final result of the program, each value manipulated within the monad is protected; in particular, each reference points to a protected value, and each procedure maps protected value to protected values. In addition, the interface allows the user to annotate values, references, and procedures with "static" labels that act as upper bounds on the actual labels that are allowed to flow into these entities. These "anchor" points are necessary for static verification which takes into account all executions of the program, and not just the particular run in question.

The formal semantics is in Fig. 1. In the semantics, we use the following notations and conventions. A protected value is written $\langle v, \ell_v\rangle$ and is ranged over by capital letters V. Judgments are of the form $\rho \vdash (e, \sigma) \mapsto (V, \sigma)$ for expressions that produce values and $\rho \vdash (e, \sigma) \mapsto \sigma$ for expressions that produce no values (i.e., commands). In both cases, an expression e is evaluated in an environment ρ and initial store σ to produce a result and an updated store. The store σ represents the memory as a map from locations to protected values.

The evaluation of $(\mathsf{runM}\ \rho\ e)$ starts the evaluation of e in the environment ρ and the empty store. For the resulting value to be released, it must be declassifiable to the least restrictive label. The final store is ignored. Binary operations, abstracted using $\oplus$, combine the labels of the two participating values using $\sqcup$: the resulting value is at least as restrictive as each of the individual values. Locations are first-class and are themselves protected. When creating a new location, the calculation of the location itself is trivial and is labeled with just the control label; the contents of the location are labeled with the given static label which acts as an upper bound on

the security level of the contents. The initialization is safe if the label of the initial value is less restrictive than the upper bound label for the contents. Reading the contents of a location returns a label that is a combination of the label associated with the location itself and the label of the contents. Updating a location requires two checks: a check that the label of the new value is guarded by the static label of the contents, and another check that guarantees that the location itself is guarded by the label of its contents [31]. In all these checks the label in question is joined with the pc to ensure that no classified information flows via control dependencies to a less restrictive label. The rules for ifM and callM show the dependency on the control flow being taken into account. The rule for localUserM shows that if the current user can act for a certain principal, then that principal can be given authority to run the code. This mechanism is also used for procedure calls. The other checks in callM ensure that the argument's value can flow into the parameter's label and conversely that the return value can flow into the result's label. The final declassification rule allows the current user to remove its own policy from the current label, and nothing else. Intuitively, user u can declassify a value by removing its own label $\mathsf{rp}\{u : \top\}$ from the value's label.

It is helpful while reading the semantic rules to keep in mind the following informal soundness argument based on the "attacker's knowledge model" for dynamic information flow policies [6]. Imagine an attacker A with a security label ℓ_A who can observe any entity (e.g., value, contents of memory location, etc.) whose security label is $\sqsubseteq_u \ \ell_A$. Furthermore the attacker can determine the control flow, i.e., which expression is currently executing, if $pc \ \sqsubseteq_u \ \ell_A$. Intuitively, the desired correctness condition is that whatever information the attacker could observe in the right-hand side of an evaluation rule could have been observed in the left-hand side, i.e., no rule can ever provide an attacker with additional information. In other words, starting from a program with an empty memory, i.e., with the attacker not knowing anything, then if the program terminates the attacker is guaranteed not to have learned anything.

Specifically, regarding the semantic rules, it is easy to see that any rule in which the labels in the right-hand side increase with respect to the lattice order cannot leak any information. The rule for localUserM does not leak because as we show in the previous section the condition $u \not\succcurlyeq u'$ guarantees that any attacker who could access information under u' would have been allowed to access the same information under u. For declassification, assume that the attacker is allowed to read the value after declassification. If the attacker A is a principal for whom u does not act, then $readers^P_A(\mathsf{rp}\{u \ : \ \top\})$ is the set of all readers and hence the attacker would have been able to access the information before declassification.

Example. We continue the tax preparer example from last section. As we established, *taxPreparer* can access some data v calculated from *Bob*'s private tax records as well as proprietary algorithms owned by *taxPreparer*. Once the calculation is complete, *taxPreparer* can use declassifyM to remove its policy and release the resulting tax form to *bob*. In more detail, assuming that v is labeled with (join taxUserPolicy taxPreparerPolicy), it is safe to evaluate (declassifyM v taxUserPolicy) to downgrade the combined policy to that of *taxUser* removing the policy of *taxPreparer*. It then becomes possible for *bob* to also use declassifyM to remove its own policy and release the tax form to a public channel.

4. Restricted Delegation and Revocation

The previous language is quite expressive but does not directly allow for restricted delegation and revocation of access rights. We illustrate the basic problem and solution using an example below.

4.1 Restricted Delegation as Declassification

Consider a situation in which principal *patient* wishes to release access to his medical history according to the following policy: the principal *doctorA* can examine the information and forward it to *doctorB* who can in turn forward it to *doctorC* if needed. No other accesses to the medical history should be granted. An unrestricted delegation, using the $\succcurlyeq$ relation, from the *patient* to *doctorA* would be too powerful as it would give *doctorA* too much power. In particular, it would allow *doctorA* to further delegate the information to other unconstrained principals. To express the desired requirements within the confines of the conventional DLM, it is possible to define an ad hoc principal hierarchy to be used specifically for access to the medical history. This approach is however awkward. Following the original RDR design, we instead propose to generalize the conventional labels to chains of labels as follows.

The idea is to think of restricted delegation as a special form of declassification. Concretely, when declassifying a value, the user provides *one* target label. We can relax this condition and allow declassification with a "chain" of target labels. The informal semantics of such a construct, delegateM $\langle v, \ell\rangle \ [\ell_1, \ldots, \ell_k]$ is as follows. First, one attempts to declassify the given value to ℓ_1 and, if that is allowed, the value becomes effectively $\langle v, \ell_1\rangle$. However, to make sure that the owner of ℓ_1 does not get unrestricted powers, we represent this information by keeping the entire delegation chain and simply moving an index up the chain to point to the current "active" principal. This idea is explained in more detail below and formalized in the semantics of the extended language.

4.2 Labels for Restricted Delegation Chains

First let's define the following labels:

- $\ell_{patient} = (\mathsf{rp}\{\top : patient\}, \bot_W)$
- $\ell_{doctorA} = (\mathsf{rp}\{\top : doctorA\}, \bot_W)$
- $\ell_{doctorB} = (\mathsf{rp}\{\top : doctorB\}, \bot_W)$
- $\ell_{doctorC} = (\mathsf{rp}\{\top : doctorC\}, \bot_W)$

The integrity component of each label is irrelevant and is fixed to be the least restrictive policy. The confidentiality component of each label defines a policy enforced by the "superuser" that grants access to the named principal. Thus, in the absence of any unrestricted delegation and no matter what current authority is running the code, the set of readers for the first label is exactly $\{patient\}$ and similarly for the remaining labels. (In practice, it is sufficient for the owner to be a principal who can act for the current authority.)

The policy that the patient wishes to enforce can be expressed using a new kind of label that consists of *chains* of conventional labels. In our running example, the desired restricted delegation by the *patient* produces the generalized label:

$$([\ell_{patient}, \ell_{doctorA}, \ell_{doctorB}, \ell_{doctorC}], 1)$$

This chain consists of a sequence of plain labels and an index pointing at the current active label in the sequence. As formalized in the next section, further delegations are only allowed if they are implied by the chain and, if so, they result in the index moving forward.

Generally, we allow more than one restricted delegation per data object which may result in several such chains, and we also allow revocation which removes some chains. As an example of these more general labels, consider:

$$\{([\ell_1, \ell_2, \ell_3, \ell_4], 2), ([\ell_5, \ell_6, \ell_7], 1)\}$$

The active label in the first chain is ℓ_3 and the active label in the second chain is ℓ_6. The fact that both these labels are active

$$\frac{\mathsf{rp}\{u:\top\} \sqsubseteq R_v \quad \mathsf{wp}\{u:\bot\} \sqsubseteq W_v \quad \forall i.R_v \sqsubseteq_u^R R_i \sqcup \mathsf{rp}\{u:\top\} \qquad \forall i.W_i \sqcap \mathsf{wp}\{u:\bot\} \sqsubseteq_u^W W_v}{(\succcurlyeq, u, pc) \vdash (\mathsf{delegateM}\ \langle v, (R_v, W_v), dl_v\rangle\ [(R_1, W_1), (R_2, W_2), \ldots], \sigma) \mapsto (\langle v, (R_v, W_v), dl_v \cup \{[(R_1, W_1), (R_2, W_2), \ldots], 0\}\rangle, \sigma)}$$

$$\frac{\mathsf{rp}\{u:\top\} \sqsubseteq R_i \quad \mathsf{wp}\{u:\bot\} \sqsubseteq W_i}{(\succcurlyeq, u, pc) \vdash (\mathsf{redelegateM}\ \langle v, \ell_v, \{..., ([\ldots, (R_i, W_i), \ell_{i+1}, \ldots], i), \ldots\}\rangle, \sigma) \mapsto (\langle v, \ell_v, \{..., ([\ldots, (R_i, W_i), \ell_{i+1}, \ldots], i+1), \ldots\}\rangle, \sigma)}$$

$$\frac{\mathsf{rp}\{u:\top\} \sqsubseteq R_v \quad \mathsf{wp}\{u:\bot\} \sqsubseteq W_v}{(\succcurlyeq, u, pc) \vdash (\mathsf{revokeM}\ \langle v, (R_v, W_v), dl_v\rangle\ [l_1, l_2, ..., l_n], \sigma) \mapsto (\langle v, (R_v, W_v), dl_v \setminus ([l_1, l_2, ..., l_n], i)\rangle, \sigma)}$$

Figure 2. Semantics of delegation and revocation.

means that the most permissive policies expressed by both these labels are enforced, i.e., that the effective current label is $\ell_3 \sqcap \ell_6$. Under the right conditions, the entire label may evolve by further delegation to either $\{([\ell_1, \ell_2, \ell_3, \ell_4], 3), ([\ell_5, \ell_6, \ell_7], 1)\}$ or to $\{([\ell_1, \ell_2, \ell_3, \ell_4], 2), ([\ell_5, \ell_6, \ell_7], 2)\}$ depending on which principal re-delegates the value to the next one in its chain. Revocation of access rights simply removes the corresponding chain from the label.

4.3 Semantics of Delegation and Revocation

We extend the interface of our library with three new constructs:

```
delegateM   :: P a → [Label] → m (P a)
redelegateM :: P a → m (P a)
revokeM     :: P a → [Label] → m (P a)
```

Previously, a protected value was of the form $\langle v, \ell\rangle$ where v is a plain value and ℓ is a label. As motivated above, the generalization of the language with dynamic delegation and revocation necessitates an additional kind of label dl (for dynamic label) consisting of chains of conventional labels, i.e., protected values are now of the form $\langle v, \ell, dl\rangle$. The additional chains included in each protected value track the variation in information flow policies due to delegation and revocation. The semantics in Fig. 1 is generalized to propagate these additional dynamic labels. The interesting rules for the new constructs are shown in Fig. 2.

The semantics of delegation is expressed using two rules in Fig. 2. The rule for delegateM considers the case when an owner initiates a delegation chain; the second rule redelegateM considers the case when a delegatee continues delegation to the next label. The evaluation rule of delegateM updates the dynamic label of the delegated value by adding a new chain with an initial delegation index equal to 0 that makes the first label active. To authorize this delegation, we require that the current user could have declassified the current value to each of the labels in the chain individually. A delegatee can then, without further checks, increment the delegation index, as shown in the rule for redelegateM. Revocation removes the chain in question irrespective of the current value of delegation index. In all cases, the static label is unchanged to retain the information about the original owner who initiated the delegation. Additional checks are performed in each rule to ensure that only the static owner can initiate a delegation and revocation, and that only the user matching the current owner in the dynamic chain is allowed to perform the redelegation.

We illustrate some of the main ideas using two examples.

Example 1. This example presents a scenario in a medical health care center, where the patient allows (by delegation) a certain physician *phys1* to examine his medical history, provide an opinion, and delegate the updated history further to *phys2*. If either physician decides to delegate the updated history to a physician not authorized by the patient (*phys3*), the delegation should not be allowed and an error exception is raised. The basic ingredients of this example can be expressed as follows:

```
-- private label for principal p
privL p = L (RP p TopP) (WP p BottomP)

e = runM ([],TopP,leastRestrictiveL ) c
  where c :: RDR (P Int)
        c = do p1 ← localUserM patient $
                    do hist ← tagM (privL patient ) 1
                       delegateM hist
                         [privL phys1, privL phys2]
               localUserM phys1 $
                 do z ← tagM leastRestrictiveL 0
                    obs1 ← newM (privL phys1) z
                    writeM obs1 p1
                    v ← readM obs1
                    delegateM v [privL phys3] -- ERROR!
```

Example 2. This example models an online shopping transaction, in which the customer delegates the right to access her credit card information to the seller which delegates this right further to the bank in order to authorize the transaction. Once the transaction has been finished, the customer revokes the delegation to the seller who should not be allowed to access the customer's credit card information after this revocation or delegate it to the bank again:

```
e = runM ([],TopP,leastRestrictiveL ) c
  where c :: RDR (P Int)
        c = do cc0 ← localUserM customer $
                     do cc ← tagM (privL customer) 1
                        delegateM cc
                          [privL seller , privL bank]
               cc1 ← localUserM seller $
                        redelegateM cc0
               cc2 ← localUserM customer $
                        revokeM cc1
                          [privL seller , privL bank]
               localUserM seller $
                 redelegateM cc0 -- ERROR!
```

5. Related Work

The literature includes several approaches for addressing delegation and/or revocation in the context of language-based security. For instance, a general form of delegation is present in Jif [10, 22] where a principal p can "act for" another principal q in such a way that any action that would authorized for principal q would also be authorized for principal p. Delegation is also possible in flow-Caml [26] by redirecting information flow from one principal to another. For instance, a declaration '**flow !bob** $<$ **!alice**' allows **bob** to transfer information to **alice**.

The first class of extensions to the above ideas is to allow delegation relationships to change dynamically. One approach is to introduce *runtime principals* [29] and augment the language with a public key infrastructure: principals are represented by public keys, authorized actions are presented as digitally signed certificates, and the delegation relation between principals is determined from certificate chains. Another approach is to directly allow the principal

hierarchy to change at runtime [4, 18]. In such an approach, the permission context, i.e., the $\succcurlyeq$ relation, would be a first-class entity that can be manipulated and extended during evaluation. Several other approaches [3] tie changes to the permission context to the dynamic chain of procedure calls: typically the caller is given temporary authority for the dynamic extent of a method invocation that manipulates sensitive information.

Other approaches (e.g. [28]) allow delegation and revocation defined in roles based security policies, where security labels in the language are defined in terms of roles. The policies are updated according to delegation or revocation of privileges given by this delegation. If the policy updates violate policy assumption, then all effects on memory due to updating the policy are rolled back. Anther approach [2] represents authorization and delegation policies using trust management and role-based labels in the program. The delegation is allowed by enabling a flow from one variable annotated with a certain label consisting of a certain role to another one annotated with a different label, where delegation is restricted to role members. Other approaches [5, 7] address restricting the delegation by modeling the "acts for" relation between runtime principals and allowing the flow between those principals based on satisfying a given security policy condition.

Information flow libraries. Another strand of research investigates information flow libraries for general purpose programming languages. For instance, the issue of enforcing information flow policies in the context of higher-order languages with mutable state was addressed by Crary et al. [12], where information flow was tracked statically through associating a security level with elements of the mutable store and using a family of monadic types to keep track of the effects of computations. Another approach [15] implemented an abstract multi-threading secure kernel that enforces information flow policy at the operating system level, where they represent threads using the state monad, parallel composition by monad transformers, and the communication between threads by the resumption monad.

Li et al. [20, 21] encode information flow as a library by embedding a security sub-language in Haskell using the notion of arrows [24], where the information flow is enforced at run-time using static analysis techniques of computations constructed in the embedded sub-language. This idea was then extended to address reference manipulation and multi-threading [11].

The need for arrows was later eliminated by Russo et al. [25] who implemented a monadic library to provide information-flow security for Haskell programs. Their library uses the monadic types to track information flow in pure and side-effectful computations. This idea was further improved in by implementing the information flow policy enforcement in a monad transformer separated from the underlying monadic API which remains unaware and unmodified [13], where the policy is specified by lifting of the underlying monad operations into the transformed monad. Another approach [27] presented a labeled IO monad as a library for enforcing dynamic information flow control in Haskell, where the library tracks a current label and permits restricted access to IO functionality, while ensuring that the current label exceeds the labels of all observed data and restricts what can be modified. This library also provides a notion of clearance to bound the current program label and provide a form of discretionary access control. Other approaches [19] address the issue of using Haskell monads to implement a library that provides non-interference through secure-multi execution, where IO operations are given different interpretations depending on the security level linked to a given execution.

In comparison to our approach, all of the previously mentioned approaches that present the enforcement of information flow control at a kernel level or as a library in Haskell either at compile-time or at run-time did not address the issue of encoding information flow policies that vary dynamically due to delegation or revocation of access rights, which is the main objective of our paper.

6. Conclusions and Future Work

We have presented a Haskell library that builds on the DLM to enforce secure information flow, with restricted delegation and revocation of access rights, in the presence of first-class references, higher-order procedures, and multiple mutually distrusting principals. Our immediate goal is to enable interaction between this language and the recently developed *Cloud Haskell* [14]. Such interaction would allow us to augment distributed Haskell programs written for a "cloud" environment with fine-grained policies that are enforced by the Haskell infrastructure.

Acknowledgments

Insightful comments and suggestions by anonymous reviewers are greatly appreciated. This material is based upon work supported by the National Science Foundation under Grant No. 1217454 and by the Egyptian Cultural & Educational Bureau.

References

[1] M. Ahsant, J. Basney, O. Mulmo, A. Lee, and L. Johnsson. Toward an on-demand restricted delegation mechanism for grids. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, GRID'06, pages 152–159, Washington, DC, USA, 2006. IEEE Computer Society.

[2] S. Bandhakavi, W. Winsborough, and M. Winslett. A trust management approach for flexible policy management in security-typed languages. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF'08, pages 33–47, Washington, DC, USA, 2008. IEEE Computer Society.

[3] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, Mar. 2005.

[4] G. Boudol. Formal aspects in security and trust. chapter Secure Information Flow as a Safety Property, pages 20–34. Springer-Verlag, Berlin, Heidelberg, 2009.

[5] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In P. Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 180–196. Springer Berlin Heidelberg, 2006.

[6] N. Broberg and D. Sands. Flow-sensitive semantics for dynamic information flow policies. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 101–112, New York, NY, USA, 2009. ACM.

[7] N. Broberg and D. Sands. Paralocks: role-based information flow control and beyond. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL'10, pages 431–444, New York, NY, USA, 2010. ACM.

[8] W. Cheng, D. R. K. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[9] W. W.-Y. Cheng. *Information Flow for Secure Distributed Applications*. Ph.D., MIT, Cambridge, MA, USA, Aug. 2009. Also as Technical Report MIT-CSAIL-TR-2009-040.

[10] S. Chong and A. C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 198–209, New York, NY, USA, 2004. ACM. ISBN 1-58113-961-6. . URL `http://doi.acm.org/10.1145/1030083.1030110`.

[11] T. chung Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Computer Security Foundations Symposium, 2007. CSF'07. 20th IEEE*, pages 187–202, 2007.

[12] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *J. Funct. Program.*, 15(2):249–291, Mar. 2005.

[13] D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *Proceedings of the 7th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI'11, pages 59–72, 2011.

[14] J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell'11, pages 118–129, New York, NY, USA, 2011. ACM.

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

[16] D. Hassan and M. R. Mousavi. RDR: A language for restricted delegation and revocation. Technical report, Eindhoven University of Technology, 2011.

[17] D. Hassan, M. R. Mousavi, and M. A. Reniers. Restricted delegation and revocation in language-based security: (position paper). In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS'10, pages 5:1–5:7, 2010.

[18] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Proc. of Foundations of Computer Security Workshop*, 2005.

[19] M. Jaskelioff and A. Russo. Secure multi-execution in haskell. In *Proceedings of the 8th international conference on Perspectives of System Informatics*, PSI'11, pages 170–178, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *Proceedings of the 19th IEEE workshop on Computer Security Foundations*, CSFW'06, 2006.

[21] P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, Apr. 2010.

[22] A. C. Myers. Jflow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL'99, pages 228–241, New York, NY, USA, 1999. ACM.

[23] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, Oct. 2000.

[24] R. Paterson. A new notation for arrows. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP'01, pages 229–240, 2001.

[25] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell'08, pages 13–24, 2008.

[26] V. Simonet. The Flow Caml System: Documentation and user's manual. Technical Report RT-0282, INRIA, July 2003.

[27] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell'11, pages 95–106, 2011.

[28] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *Proceedings of the 19th IEEE workshop on Computer Security Foundations*, CSFW'06, pages 202–216, Washington, DC, USA, 2006. IEEE Computer Society.

[29] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. *ACM Trans. Program. Lang. Syst.*, 30(1), Nov. 2007.

[30] J. Wainer, A. Kumar, and P. Barthelmess. DW-RBAC: A formal security model of delegation and revocation in workflow systems. *Inf. Syst.*, 32(3):365–384, May 2007.

[31] S. A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.