Research Report

Wesley Nuzzo

May 23, 2023

1 Abstract

This report discusses a proposal for implementing Declassification and Erasure policies in Haskell. These extend ordinary IFC policies by allowing policies to change to a lower level (declassification) or requiring them to change to a higher level (erasure) if a particular condition is met.

The approach is to modify the LIO library to add the following: (1) a new type of label, tentatively called DEPLabel, which is similar to the existing DCLabel except that it also is capable of representing declassification and erasure policies, (2) a new type, Cond, which is similar to an LIORef of a Bool except that once set, it can never be unset, and it can be used as the condition for DEPLabel, and (3) a function update() which can be used to update a policy based on the current state of the conditions according to the LIO monad.

This report includes updated formal semantics for LIO based on this approach, a discussion of the practical implementation, and case studies.

2 link to repositories

The source code associated with this report can be found at https://github.com/UraniumCronorum/lio-de, which is a fork of https://github.com/PLSysSec/lio/ commit 622a3e7.

3 Theory

This section is an update to the formal specification presented in the LIO paper.

4 Update of figure 2 (syntax)

Bool: b
Label: l
Address: a
Policy:

$$p ::= l \mid p \nearrow^a p \mid p \searrow^a p$$

Term:

$$v ::= \texttt{true} \mid \texttt{false} \mid () \mid p \mid a \mid x \mid \lambda x.e \mid (e, e) \mid \texttt{fix}\ e \mid \texttt{Lb}\ v\ e \mid (e)^{\texttt{LIO}} \mid \bullet \mid \texttt{Cnd}\ v\ b$$

Expression:

$$e ::= \ldots \mid \texttt{newCond}\ e\ e \mid \texttt{readCond}\ e \mid \texttt{setCondTrue}\ e \mid \texttt{labelOfCond}\ e$$

5 Two different paradigms

There are two different possible paradigms for implementing this concept.

The first paradigm is the one presented in Chong's thesis: no memory location can change it's policy, but a declassification policy can be flow to it's secondary policy if it's condition is met, an erasure policy can flow to the join of its primary and secondary policy at all times, and data must be deleted if it requires erasure (i.e., it has an erasure policy whose condition is met or its primary policy requires erasure).

The second paradigm is a bit simpler: the data doesn't change, but the policy does change "under" the values as the conditions update.

6 Semantics

6.1 paradigm 2 (updating policy)

I'm still working on this one.

Label: $$\frac{\Sigma.lbl \sqsubseteq l \sqsubseteq \Sigma.clr}{\langle \Sigma, E[\texttt{label}\ p\ e] \rangle \longrightarrow \langle \Sigma, E[\texttt{return}\ (\texttt{Lb}\ p\ e)] \rangle}$$

Unlabel: $$\frac{p' = \Sigma.lbl \sqcup update(p, \Sigma) \quad p' \sqsubseteq \Sigma.clr \quad \Sigma' = \Sigma[lbl \mapsto p']}{\langle \Sigma, E[\texttt{unlabel}\ (\texttt{Lb}\ p\ e)] \rangle \longrightarrow \langle \Sigma', E[\texttt{return}\ e] \rangle}$$

—

$$\text{SetCondTrue}\quad \frac{\Sigma.\phi(a) = \texttt{Cond}\ p\ e \qquad \Sigma' = \Sigma.\phi[a \mapsto \texttt{Cond}\ p\ \texttt{true}] \qquad \Sigma.lbl \sqsubseteq p \sqsubseteq \Sigma.clr \quad \Sigma'' = \Sigma[lbl \mapsto update(\Sigma'.lbl, \Sigma')]}{\langle \Sigma, E[\texttt{setCondTrue}\ a] \rangle \longrightarrow \langle \Sigma'', E[\texttt{return}\ ()] \rangle}$$

6.1.1 helper function: *update*

$$update(l, \Sigma) = l$$

When c is true:

$$update(p \searrow^c q, \Sigma) = p \sqcap (pol(c) \sqcup q)$$

$$update(p \nearrow^c q, \Sigma) = p \sqcup q$$

When c is false:

$$update(p \searrow^c q, \Sigma) = p \searrow^c q$$

$$update(p \nearrow^c q, \Sigma) = q \sqcap (pol(c) \sqcup p \nearrow^c q)$$

7 Implementation

This section discusses the practical implementation of these ideas

8 LIO's DCLabel

We're using the DCLabel type already defined in the LIO Library as a starting point.

A DCLabel (Disjunction Category Label) has a secrecy component and an integrity component, and would be specified like:

```
dc = secrecy %% integrity
```

Where `secrecy` is the secrecy component and `integrity` is the integrity component. Both components are CNFs, which express policies in Conjunctive Normal Form, e.g.:

```
secrecy = (a1 \/ a2 \/ ...) /\ (b1 \/ b2 \/ ... ) /\ ...
```

A label `s1 %% i1` $\sqsubseteq$ `s2 %% i2` if and only if when interpreted as logical predicates, `s2` implies `s1` and `i1` implies `i2`.

Since we're only really interested in the secrecy component, we can assume that the integrity component is always `true`. This allows us to ignore it in our analysis.

8.1 joins, meets and so on

Comparing this to the normal notation for meets and joins: if s_1, s_2 and i are CNFs, then:

$$(s_1 \texttt{ \%\% } i) \sqcup (s_2 \texttt{ \%\% } i) = (s_1 \wedge s_2 \texttt{ \%\% } i)$$

$$(s_1 \texttt{ \%\% } i) \sqcap (s_2 \texttt{ \%\% } i) = (s_1 \vee s_2 \texttt{ \%\% } i)$$

$$(s_1 \vee s_2 \texttt{ \%\% } i) \sqsubseteq (s_k \texttt{ \%\% } i) \sqsubseteq (s_1 \wedge s_2 \texttt{ \%\% } i), k \in \{1, 2\}$$

$$\top = (\texttt{cFalse \%\% } i)$$

$$\bot = (\texttt{cTrue \%\% } i)$$

9 Normal Form for Declassification and Erasure Policies

The idea is to extend the existing normal form to account for declassification and erasure policies. Because it's a normal form, if two policies are the same, they should have the same normal form, even if they're initially expressed differently.

Originally my idea was to do this:

$$p \nearrow^c q = (p \nearrow^c \top) \sqcap (p \sqcup q)$$

$$p \searrow^c q = p \sqcap (\top \searrow^c q)$$

However, it's not clear how to represent $\top \searrow^c \bot$ or $\bot \nearrow^c \top$ using this strategy.

So instead, I propose this:

$$p \nearrow^c q = (p \sqcup q) \sqcap (p \sqcup (\bot \nearrow^c \top))$$

$$p \searrow^c q = p \sqcap (q \sqcup (\top \searrow^c \bot))$$

Which enables us to add two terms, `E` c and `D` c, representing $\bot \nearrow^c \top$ and $\top \searrow^c \bot$ respectively. Every other declassification and erasure policy can be constructed from these two terms together with Principals using conjunctions and disjunctions.

10 Proposed Changes to LIO

First of all, it is probably necessary to implement a `Cond` type which functions similarly to an `LIORef` of `Bool`, but which doesn't allow conditions to be unset once set and which allows comparing conditions based on reference equivalence.

Given the proper implementation of `Cond`, we can implement the following type:

```
data AtomicPolicy = L Principal | D Cond | E Cond

-- constructors

latticeFromPrincipal :: Principal -> AtomicPolicy
latticeFromPrincipal pr = L pr

latticePolicy :: String -> AtomicPolicy
latticePolicy str = latticeFromPrincipal $ principal str

-- instance
instance Eq AtomicPolicy where
        L x == L y  =  x == y
        D c == D d  =  c == d
        E c == E d  =  c == d
        x == y  =  false
```

(Code may have some errors; not sure)

If we then replace `Principal` with `AtomicCond` in the regular DCLabel code, we should have the basic support we need for label relationships.

We can also implement the following helper functions:

```
(/>) :: (ToCNF a, ToCNF b) -> a -> b -> Cond -> CNF
```

```
(\>) :: (ToCNF a, ToCNF b) -> a -> b -> Cond -> CNF

p (/>) q c = (p /\ q) \/ (p /\ (E c))
p (\>) q c = p \/ (q /\ (D c))
```

Which would make it easier to create general declassification and erasure policies.

After this, we would need to implement the `update` function, which updates a label based on the current status of its conditions. We can define this helper function:

```
updateAtomic :: LIO AtomicPolicy -> LIO AtomicPolicy
updateAtomic (LIO (L l)) = LIO (L l)
updateAtomic (LIO (D c)) = if (readCond c) then cTrue else (D c)
updateAtomic (LIO (E c)) = if (readCond c) then cFalse else (E c)
```

Which then can be mapped to the atoms of the CNF to update the entire policy.

After this, its just a matter of calling `update` in the situations where the policy needs to be updated, as discussed earlier.

11 Case Studies

This section discusses three case studies for practical applications of these ideas.

12 Ideas for application to real-world model

I had three main ideas here.

In all cases I'm envisioning an online webserver where data is collected, and users have the ability to submit requests for their data to be deleted through this server.

Of these, my main focus is on the mailing list idea.

12.1 idea 1: mailing list

Users sign up to the mailing list by entering a login and an email address. The server can then send emails to all users on the mailing list using the addresses it has.

Users can use their login to update their email address or delete it from the server entirely. Policy should enforce erasure of the original email address when it gets updated, and erasure of any email address that gets deleted, along with account info.

12.2 idea 2: fitness tracker

A simple fitness app that records things like number of steps taken per day, and so on. Data is grouped by date and time, and users would have the ability to see their own data or delete it for a particular category or time period.

Could maybe add the option to declassify data to send it to, e.g. a doctor or a fitness coach.

12.3 idea 3: ad service

A server that stores data on a users demographics and interests, and can serve ads based on that information.

Users would be able to query the database to see what the database categorizes them as, and to request the deletion of some or all of that data.