Implementation Report

Wesley Nuzzo

May 23, 2023

1 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.

1.1 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:

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

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

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

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

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

2 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.

3 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 in the previous report.