Research Update 2

Wesley Nuzzo

March 15, 2023

1 intro

I've spent most of my time this week working on trying to develop a more concrete version of the email server idea I mentioned in the last update. My idea was to have the theoretical implementation using a modified version of the language defined in the paper, and have the practical implementation in C++ (which seems like a good fit because it supports type-checking and encapsulation and important features).

There are a few challenges:

My original idea was to support a variable number of users (this makes sense in most practical, real world examples). However, this is tricky to implement (at least in the theoretical model) because it requires a way to create a new entry in the security lattice for a new user, and it requires a heterogeneous array to store data belonging to multiple different users.

I/O is tricky as well. For input, if I'm trying to create a new entry for a user, I need to immediately store that data with the correct erasure policy, or else that information either won't be available when I try to store it in the array, or will not be guaranteed to be erased when the condition is met, because the initial policy failed to protect it.

However, this means that before I input a value, I need to create the condition for its erasure. After that, I need a way to copy that policy to the array.

2 Version for a constant set of users

Suppose we have n users. We can have a security lattice such that, for each i there is a lattice policy $\texttt{user}_i$, and for any two users i and j, $\texttt{user}_i \not\sqsubseteq \texttt{user}_j$. We can also have a top-level lattice policy $\top$, which indicates that a value needs to be deleted from the system.

For each user, we'll have an entry a_i for that user, and a condition c_i for its erasure. The policies on these variables would be $\Gamma(c_i) = \texttt{user}_i$, and $\Gamma(a_i) = \texttt{user}_i \nearrow^c \top$.

2.1 updating an entry

To update the value, we want to first erase the existing value, and then set it to the new value. Let's assume we have the new value in x, and there is a condition *endtransaction*, so that $\Gamma(x) = \top \searrow^{endtransaction} (\texttt{user}_i \nearrow^{c} \top)$. Then we can do the following:

```
c_i := 1; c_i := 0;
endtransaction := 1;
a_i := declassify(x, (top -> ...) to (user_i ->c top) using endtransaction)
```

The first line ensures that the existing value of a_i gets erased, while ensuring that c_i gets reset to a False value before a_i is reassigned. The second line makes it possible to declassify x, and the final line performs the declassification.

Note that *endtransaction* is necessary because without it the value would be overwritten when we set $c_i := 1$.

2.2 erasing an entry

Erasing an entry is very simple. All that one has to do is set $c_i := 1$. Since the condition has been met, the variables using the policy must be erased.

3 Thoughts on a practical implementation

Obviously, C++ does not directly implement these kinds of security constructs. I'm a little unsure as to what approach to go for here; I could try to leverage C++'s static type checking to get a version of these guarantees, or I could try adding some kind of dynamic checks, or there may be other approaches.

One challenge for a practical implementation is that we need to be able to determine whether a variable with an erasure policy *requires erasure* at a particular point in the execution, even though the condition for that erasure may have since been reset to False.

One way to do this is somehow ensure that erasure conditions can't be reset to False after they've been set True (requiring a new erasure policy and condition if you want a new variable with a similar policy).

Another approach is to have a set up so that when the condition gets set True, the code immediately deletes or erases any variables affected by that erasure condition. That way, when you go to assign from a variable, you don't need to worry about whether it requires erasure or not.