Formalization of Dynamic Information Flow Control

Aaron Eline

December 2020

1 Introduction

For this project I extended the Imp language form Software Foundations to include dynamic information flow control. The work consists of first of extending the given Imp syntax to allow for labeling values as either being *Public* or *Secret*. The the evaluation relations must be extended to correctly propagate and respect these new labels. Finally, a property of *Indistinguishably* must be defined and a proof of correctness written showing that indistinguishable inputs always step to indistinguishable outputs. All of the following is formalized and contained in the attached Coq file.

2 Extending Imp

I started by coping the Imp implementation from Software Foundations and then extending it. The core of the extension is in expanding the values of Imp, which were either booleans or natural numbers, to now be tuples. Every value now contains either a boolean or a natural number as well as a security label. Labels can either be *Public* or *Secret*. The most basic operation on labels is *merge*:

$$merge(Public, Public) = Public$$

$$merge(\_, \_) = Secret$$

2.1 Expressions

A expressions are left basically the same. Values inside an a-expression can not be explicitly labelled. Instead, they are all assumed to be *Public*. Likewise, B-expressions are mostly left the same.

2.2 Commands

Commands are the construct where the majority of changes happen. The assignment operation now contains an explicit label. This results in merging the

explicit label with whatever label evaluating the a-expression returns. The full syntax is given below:

```
    n ::= number
    v ::= ((n label) (true label) (false label))
label ::= Public | Secret
   id ::= variable-not-otherwise-mentioned
aexp ::= n | id
        | (aexp + aexp)
        | (aexp - aepx)
        | (aexp * aexp)
bexp ::= true | false
        | (aexp = aexp)
        | (aexp ≤ aexp)
        | (not bexp)
        | (bexp && bexp)
 com ::= skip
        | (id := aexp label)
        | (seq ::= com ; com)
        | (if bexp com com)
```

2.3 Stores

The store has been extended to be a map from ids to ($\mathbb{N} \times Label$) We use the definition of Maps from the Coq standard library, which allows the use of several built-in relations. One of which is $MapsTo$, defined as follows:

$$\forall (x : id)(v : key)(m : Map\ id\ key), MapsTo\ x\ v\ k$$

Will hold if and only x is bound to v in m.

3 Evaluation

3.1 Expressions

The evaluation relation for both expressions is extended in a trivial way. It simply now $merge$'s the labels resulting from sub expressions.

$$\frac{}{aevalR[n, State, (n\ Public)]}$$

$$\frac{MapsTo[id, (v\ l), State]}{aevalR[id, State, (v\ l)]}$$

$$\frac{aevalR[a1, State, (v1\ l2)] \quad aevalR[a2, State, (v2\ l2)]}{aevalR[(a1 + a2), State, ((v1 + v2)\ (merge\ l1\ l2))]}$$

$$\frac{aevalR[a1, State, (v1\ l2)] \quad aevalR[a2, State, (v2\ l2)]}{aevalR[(a1 - a2), State, ((v1 - v2)\ (merge\ l1\ l2))]}$$

$$\frac{aevalR[a1, State, (v1\ l2)] \quad aevalR[a2, State, (v2\ l2)]}{aevalR[(a1 * a2), State, ((v1 * v2)\ (merge\ l1\ l2))]}$$

Figure 1: A-Expression Evaluation

$$\frac{}{bevalR[true, State, (true\ Public)]}$$

$$\frac{}{bevalR[false, State, (false\ Public)]}$$

$$\frac{aevalR[a1, State, (n1\ l1)] \quad aevalR[a2, Statae, (n2\ l2)]}{bevalR[(a1 = a2), State, ((n1 =? n2)\ (merge\ l1\ l2))]}$$

$$\frac{aevalR[a1, State, (n1\ l1)] \quad aevalR[a2, Statae, (n2\ l2)]}{bevalR[(a1 \leq a2), State, ((n1 \leq? n2)\ (merge\ l1\ l2))]}$$

$$\frac{bevalR[(not\ b), State, (false\ l)]}{bevalR[b, State, (true\ l)]}$$

$$\frac{bevalR[(not\ b), State, (true\ l)]}{bevalR[b, State, (false\ l)]}$$

$$\frac{bevalR[b1, State, (bool1\ l1)] \quad bevalR[b2, State, (bool2\ l2)]}{bevalR[(b1\ \&\&\ b2), State, ((and\ bool1\ bool2)\ (merge\ l1\ l2))]}$$

Figure 2: B-Expression Evaluation

3.2 Commands

The relation for commands is also extended to support security labels. However, the command relation is no longer over *States*, but instead over a new type called a *Security Context*. Defined as $(State \times Label)$. The label tracks the security level of the current computation. The definition of the evaluation relations introduces a couple of convenience properties, defined below:

- $In\ x\ s$, a property from the Coq library, holds if key x is mapped in state s.
- $writing_valid\ x\ s\ l$, which holds when x is mapped in s with the same label.
- $merge_list lst$, recursively merges a list of labels.

The additions to this relation occur in the assignment and in the if statements. For assignment, we make sure that after we merge the label of the evaluated expression with the label of the context, we are writing to a variable with the same label. In if statements, the context of the branches is updated: it's label is computing by merging the label of the guard with the context label of the if-statement.

4 Indistinguishablility

Next, we define a couple of Indistinguishablility relations.

4.1 Values

For any time T, the type $(T \times Label)$ has a indistinguishablility relation. If both labels are $Secret$, then they are indistinguishable. If both labels are $Public$, then they are only indistinguishable if the members of type T are equal.

4.2 Expressions and Commands

Expressions and commands are indistinguishable except for assignments marked as Secret.

4.3 States

For states to be indistinguishable, they must map all the same keys to values that are themselves indistinguishable.

4.4 Security Contexts

Security contexts are indistinguishable if the labels are the same and if the states are indistinguishable.

5 Correctness

The main correctness theorem is as follows: Starting from any two commands (c_1, c_2) that are indistinguishable, and two contexts $(ctxt_1, ctxt_2)$ that are indistinguishable, evaluating them will results in two new contexts that are indistinguishable.

6 Challenges

The main challenges appeared in the formulation of the indistinguishability for states. It took multiple revisions, as my first drafts were not strong enough.

7 Extensions

If I had more time, there are two areas I would've liked to explore

Firstly I would've liked to add more features into the language. While loops seem an obvious first choice, but I think added function calls would also be interesting.

Secondly I think it would be interesting to try to add a security type system to the language.

Finally, a more complicated permission lattice would be an interesting final addtion.