Generating Information-Flow Control Mechanisms from Programming Language Specifications

Andrew Bedford (github.com/andrew-bedford)

Laval University, Canada

Motivation

Modern operating systems rely on access-control mechanisms to protect users information. However, access control mechanisms are insufficient as they cannot regulate the propagation of information once it has been released.
To address this issue, a new research trend called **language-based information-flow security** has emerged. The idea is to use techniques from programming languages, such as program analysis and type checking, to enforce information-flow policies. Mechanisms that enforce such policies are called **information-flow control mechanisms**.

Problem

Developing sound information-flow control mechanisms can be a laborious and error-prone task due to the numerous ways through which information may flow in a program.

Background

Most information-flow control mechanisms seek to enforce a policy called **non-interference**, which states that private information may not interfere with the publicly observable behavior of a program. To enforce non-interference, two types of information flows must be taken into account:

1. **Explicit flows** occur when private information flows directly into public information.
2. **Implicit flows** occur when private information influences public information through the control-flow of the application.

Ott-IFC

We have created a tool called **Ott-IFC** that takes as input a programming language's specification (i.e., syntax and semantics, written in **Ott**) and produces a mechanism's specification. The development process of a mechanism using Ott and Ott-IFC looks like this:

1. Write a specification of the language on which we want to enforce non-interference in Ott.
2. Use Ott-IFC to generate the mechanism.
3. Use Ott to export the mechanism to LaTeX/Coq/Isabelle/HOL and complete the implementation.

Example

a ::= x | n | a1 + a2 | a1 * a2 b ::= true | false | a1 < a2 c ::= skip | x := a | c1 ; c2 | if b then c1 else c2 end | while b do c end

Input

```
<a, m> || <n, m>
-----------------------------------
<x := a, m> || <skip, m[x |-> n]>
```

Output

```
  <a, m> || <n, m>
  E |- a : ta
  E |- x : tx
  ta <= tx
  -----------------------------------
  <x:=a, m, E, pc> || <skip, m[x |-> n
], E[x |-> ta], pc>
```

Input

```
<b , m > || < true , m >
<c1 , m > || < skip , m1 >
--------------------------------------------

<if b then c1 else c2 end, m> || <skip ,
 m1>

<b , m> || < false , m>
<c2 , m> || < skip , m2>
--------------------------------------------

<if b then c1 else c2 end, m> || <skip ,
 m2>
```

Output

```
E |- b : l_b
<cmd1, m, o, pc |_| l_b, E> || <stop,
 m1, o1, pc |_| l_b, E>
<b, m, o, pc, E> || <true, m, o, pc, E
 >
-----------------------------------------------------


<if b then cmd1 else cmd2 end, m, o,
 pc, E> ||  <stop, m1, o1, pc, E>

E |- b : l_b
<b, m, o, pc, E> || <false, m, o, pc,
 E>
<cmd2, m, o, pc |_| l_b, E> || <stop,
 m2, o2, pc |_| l_b, E>
-----------------------------------------------------


<if b then cmd1 else cmd2 end, m, o,
 pc, E> ||  <stop, m2, o2, pc, E>
```

Current Status

We have implemented a prototype of our algorithm and validated that it works on two imperative languages. It currently supports languages whose specification:

1. is composed of expressions, which may only read the memory, and commands, which may read or write the memory
2. program configurations are of the form $\langle command, memory \rangle$.

We have also begun to draft a soundness proof, that is, a proof showing that the generated mechanisms enforce non-interference.

Future Work

- Add support for a greater variety of languages
- Parametrize Ott-IFC so that it can generate multiple types of mechanisms
- Automatically generate
- Use Ott-IFC's rewriting rules to verify existing mechanisms

Acknowledgements

We would like to thank Josée Desharnais, Nadia Tawbi for their support and the anonymous reviewers for their comments.