Generating Information-Flow Control Mechanisms from Programming Language Specifications

Andrew Bedford (andrew.bedford.1@ulaval.ca)

Laval University, Canada

Motivation

Modern operating systems rely on access-control mechanisms to protect users information. However, these 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** [3] 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** [2], which states that private information may not interfere with the publicly observable behavior of a program. More formally:

> *A program p satisfies non-interference if for any $\ell \in \mathcal{L}$, and for any two memories m and m' that are ℓ-equivalent, and for any trace o such that $\langle p, m, \epsilon \rangle \downarrow o$, then there is some trace o', such that $\langle p, m', \epsilon \rangle \downarrow o'$ and $o \upharpoonright \ell$ is a prefix of $o' \upharpoonright \ell$ (or vice versa).*

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.

```
public := private
```

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

```
if (private > 0) then
  public := 0
else
  public := 1
end
```

Approach and Uniqueness

These mechanisms are usually designed and implemented completely by a human. We have created a tool called **Ott-IFC** that automates part of the process. It takes as input a programming language's specification (i.e., syntax and semantics) and produces a mechanism's specification.

Example

```
arith_expr, a ::= x | n | a1 + a2 | a1 * a2                                bool_expr, b ::= true | false | a1 < a2
commands, c ::= skip | x := a | c1 ; c2 | if b then c1 else c2 end | while b do c end | read x from ch | write x to ch
```

To prevent explicit flows, Ott-IFC identifies the semantic rules that may modify the memory m (e.g., rule assign). In each of those rules, it updates the modified variable's label with the label of the expressions that are used in the rule.

Input

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

Output

```
<a, m, o, pc, E> || <n, m, o, pc, E>
E |- a : la
-----------------------------------------------------
<x := a, m, o, pc, E> || <stop, m[x |-> n], o, pc,
  E[x |-> pc |_| la]>
```

If an output is produced, it inserts a guard condition to ensure that no leak occurs.

Input

```
m(x) = n
-----------------------------------------------------
<write x to ch, m, o> || <stop, m[ch |-> n],
  o::(ch ; n)>
```

Output

```
E |- x : lx
E |- ch : lch
lx |_| pc <= lch
m(x) = n
-----------------------------------------------------
<write x to ch, m, o, pc, E> ||  <stop, m[ch |-> n],
  o::(ch ; n), pc, E>
```

To prevent implicit flows, it identifies commands that may influence the control-flow of the application. It then updates to the program counter pc with the level of the expressions that are present in the rule.

Input

```
<b, m, o> || <true, m, o>
<c1, m, o> || <stop, m1, o1>
-----------------------------------------------------
<if b then c1 else c2 end, m, o> || <stop, m1, o1>



<b, m, o> || <false, m, o>
<c2, m, o> || <stop, m2, o2>
-----------------------------------------------------
<if b then c1 else c2 end, m, o> || <stop, m2, o2>
```

Output

```
E |- b : lb
<b, m, o, pc, E> || <true, m, o, pc, E>
<c1, m, o, pc |_| lb, E> || <stop, m1, o1, pc |_| lb, E>
E1 = updateModifVars(E, pc, b, c2)
-----------------------------------------------------
<if b then c1 else c2 end, m, o, pc, E> ||  <stop, m1, o1
  , pc, E1>

E |- b : lb
<b, m, o, pc, E> || <false, m, o, pc, E>
<c2, m, o, pc |_| lb, E> || <stop, m2, o2, pc |_| lb, E>
E2 = updateModifVars(E, pc, b, c1)
-----------------------------------------------------
<if b then c1 else c2 end, m, o, pc, E> ||  <stop, m2, o2
  , pc, E2>
```

Current Status

We have implemented a prototype of our algorithm [1] 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. states are of the form $\langle$*command*, *memory*, *outputs*$\rangle$.

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 a skeleton of proof in Coq
- 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.

References

[1] Bedford, A. Ott-ifc's github repository. https://github.com/andrew-bedford/ott-ifc, 2017.

[2] Goguen, J. A., and Meseguer, J. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982* (1982), pp. 11–20.

[3] Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21*, 1 (2003), 5–19.