Ott-IFC: Generating Information-Flow Control Mechanisms

Andrew Bedford
Laval University
Quebec, Canada
andrew.bedford.1@ulaval.ca

Abstract

Developing information-flow control mechanisms can be a difficult and time-consuming task, particularly when dealing with complex programming languages.

To help with this task, we have developed a tool called *Ott-IFC* that can automatically generate information-flow control mechanisms from programming language specifications (i.e., syntax and semantics).

1 Introduction

Modern operating systems rely mostly 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 for processing. To address this issue, a new research trend called *language-based information-flow security* [6] has emerged. The idea is to use techniques from programming languages, such as program analysis, monitoring, rewriting and type checking, to enforce information-flow policies (e.g., information from a private file should not saved in a public file). Mechanisms that enforce such policies (e.g., [2, 3, 8]) are called *information-flow control mechanisms*.

Information-flow control mechanisms are usually designed and implemented completely by a human. Developing sound mechanisms can be a challenging task, particularly when dealing with complex programming languages, due to the numerous ways through which information may flow in a program. In order to make this process less laborious and reduce the risk of errors, we have created a tool called Ott-IFC that takes as input a programming language's specification (i.e., syntax and semantics) and produces a mechanism's specification (e.g., runtime monitor).

Contributions

- We present Ott-IFC in Section 3.
- We prove that the mechanisms generated by Ott-IFC are sound in Section 4.
- We discuss Ott-IFC's limitations in Section

2 Background

Most information-flow control mechanisms seek to enforce a policy called *non-interference* [5], which essentially states that private information may not interfere with the publicly observable behavior of a program. To enforce non-interference, a mechanism must take into account two types of information flows: *explicit flows* and *implicit flows* [4].

An insecure explicit information flow (Listing 1) occurs when private information flows directly into public information.

```
public := private
```

Listing 1. Insecure explicit flow

Explicit flows can be prevented by associating labels to sensitive information and propagating them whenever the information is used; a process known as *tainting*.

An insecure implicit information flow (Listing 2) occurs when private information influences public information through the control-flow of the application.

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

Listing 2. Insecure implicit flow

Implicit flows can be prevented using a program counter, *pc*, which keeps track of the context in which a command is executed.

3 Ott-IFC

As the name implies, the specifications that Ott-IFC takes as input (and outputs) are written in Ott [7]. Ott is a tool that can generate LaTeX, Coq or Isabelle/HOL versions of a programming language's specification. The specification is written in a concise and readable ASCII notation that ressembles what one would write in informal mathematics (see following Listings).

Hence, the development process of a mechanism using Ott and Ott-IFC would look 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.

pseudocode

identifying commands and expressions

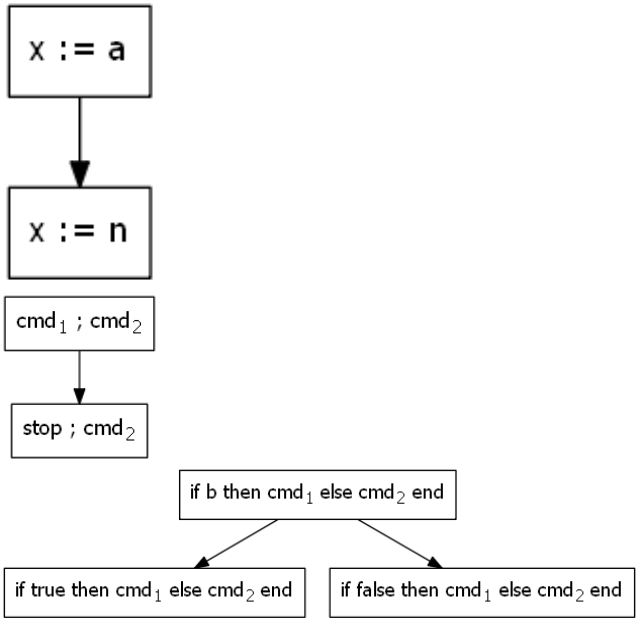
unfolding productions

For the moment, Ott-IFC supports only languages whose specification respects certain rules. Namely, that the syntax be composed of expressions, which may only read the memory, and commands, which may read or write the memory; and that the program configurations be of the form $\langle command, memory \rangle$.

To illustrate our approach, consider the imperative language whose syntax is defined in Listing 3 and (partial) semantics in Listings **??** and **??**.

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

Listing 3. Ott syntax of a simple imperative language



In order to automatically apply the techniques described in Section 2, Ott-IFC starts by inserting a typing environment `E` (which maps variables to their label) and a program counter `pc` in each semantic rules. That is, the configurations `<c, m>` are changed to `<c, m, E, pc>`.

To prevent explicit flows, it identifies the semantic rules that may modify the memory `m` (e.g., Listing **??**). In each of those rules, it inserts a condition to ensure that private information is not stored into a public variable and updates the modified variable's (`x` here) label with the label of the expressions that are used in the rule.

To prevent implicit flows, it identifies commands that may influence the control-flow of the application. That is, commands for which a program configuration may lead to two different program configurations (e.g., Listing **??**). It then updates to the program counter `pc` with the level of the expressions that are present in the rule (only `b` in this case).

4 Soundness

For our purposes, we assume that the levels of information are organized in a finite lattice $(\mathcal{L}, \sqsubseteq)$ which contains at least two elements: L for the bottom of the lattice (least important) and H for the top of the lattice (most important), i.e. $\forall l \in \mathcal{L}, L \sqsubseteq l \wedge l \sqsubseteq H$.

Definition 4.1 (Non-interference)**.** 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).

```
<a, m, o> || <n, m, o>
-----------------------------------------------------

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

```
<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]>
```

```
m(x) = n
-----------------------------------------------------

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

```
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>
```

```
<b, m, o> || <true, m, o>
<c1, m, o> || <stop, m1, o1>
-----------------------------------------------------

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

```
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>
```

```
<b, m, o> || <false, m, o>
<c2, m, o> || <stop, m2, o2>
-----------------------------------------------------

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

```
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>
```

5 Future Work

We have implemented a prototype of our algorithm [1] and validated that it works on two imperative languages: one defined using small-step semantics and the other using big-step semantics. We have also begun to draft a soundness proof, that is, a proof showing that the generated mechanisms enforce non-interference.

Before our tool can be of real use to most researchers, much work remains to be done.

Language Support The restrictions on the syntax and configurations means that only certain types of languages can be used in Ott-IFC. For example, most functional languages would not be supported because, in those languages, commands can be expressions. We are currently in the process of building a repository of formalized languages so that we can test and extend our approach to a wider range of languages.

Parametrization For the moment, Ott-IFC only generates one type of information-flow control mechanism. We plan on parametrizing our tool so that users can choose the type of mechanism to generate (e.g., type system, monitor) and choose some of its features (e.g., flow-sensitivity, termination-sensitivity, progress-sensitivity).

Generating Formal Proofs We expect that some users will use the mechanisms generated by Ott-IFC as a foundation to build better and more precise mechanisms. One of the most grueling task when building an information-flow control mechanism is to prove its soundness. In order to help those users, we plan on generating a skeleton of the proof in Coq or Isabelle/HOL (both languages are supported by Ott).

Verifying Existing Mechanisms The same rules that Ott-IFC uses to generate sound mechanisms could be used to verify the soundness of existing mechanisms and identify potential errors.

Acknowledgments

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

References

[1] Andrew Bedford. 2017. Ott-IFC's Repository. https://github.com/andrew-bedford/ott-ifc. (2017).

[2] Andrew Bedford, Stephen Chong, Josée Desharnais, Elisavet Kozyri, and Nadia Tawbi. 2017. A progress-sensitive flow-sensitive inlined information-flow control monitor (extended version). *Computers & Security* 71 (2017), 114–131. https://doi.org/10.1016/j.cose.2017.04.001

[3] Andrey Chudnov and David A. Naumann. 2010. Information Flow Monitor Inlining. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. 200–214. https://doi.org/10.1109/CSF.2010.21

[4] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (1976), 236–243. https://doi.org/10.1145/360051.360056

[5] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. 11–20. https://doi.org/10.1109/SP.1982.10014

[6] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[7] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. 2010. Ott: Effective tool support for the working semanticist. *J. Funct. Program.* 20, 1 (2010), 71–122. https://doi.org/10.1017/S0956796809990293

[8] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 2/3 (1996), 167–188. https://doi.org/10.3233/JCS-1996-42-304