Ott-IFC: Automatically Generating Information-Flow Control Mechanisms

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

Abstract

We present Ott-IFC a tool that can automatically generate information-flow control mechanisms from programming language specifications (i.e., syntax and semantics).

1 Problem and Motivation

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* [5] has emerged. The idea being 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 information-flow policies (e.g., [1]) are called *information-flow control mechanisms*.

Due to the numerous ways through which information may flow in a program, developping sound information-flow control mechanisms can be a challenging and error-prone task, particularly when dealing with complex programming languages.

2 Background and Related Work

Most information-flow control mechanisms seek to enforce a policy called *non-interference* [3]. It 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 flow* and *implicit flow* [2]. An insecure explicit information flow (Listing 1) occurs when a private information flows directly into public information. An insecure implicit information flow (Listing 2) occurs when private information influences public information through the control-flow of the application.

```
public := private
```

Listing 1. Insecure explicit flow

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

Listing 2. Insecure implicit flow

Techniques for tracking and controlling explicit and implicit flows at the language level are well known [4, 7], and are used in this work.

3 Approach and Uniqueness

We have developped a tool called Ott-IFC that can generate basic information-flow control mechanisms from programming language specifications (i.e., syntax and semantics). To the best of our knowledge, we are the first ones to achieve this; they are usually designed and implemented from the ground-up by a human.

As the name implies, Ott-IFC uses Ott [6] as its input/output language. Ott is a tool that takes as input a programming language's syntax and semantics, written in a concise ASCII notation that ressembles what one would write in informal mathematics, and generates LaTeX, Coq or Isabelle/HOL versions of the definition.

Ott-IFC makes two assumptions about the language's specification: (1) that its syntax be composed of expressions, which may only read the memory, and commands, which may read or write the memory; and (2) that its semantics program configurations be of the form $\langle command, memory \rangle$.

To illustrate our approach, consider the simple imperative language of Listing 3.

```
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
```

Listing 3. Ott syntax of a simple imperative language

We start by inserting in all the rules a typing environment `E` which maps variables to their level of information, and a program counter `pc` is used to keep track of the level of information that influenced the control-flow.

POPL 2018, January 07–13, 2018, Los Angeles, CA, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

To prevent explicit flows, we identify the semantics rules that may modify the memory `m` (e.g., Listing 4).

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

Listing 4. Ott big-step semantics of the assign command

In each of those rules, we insert a condition to ensure that we are not storing private information in a public variable.

To prevent implicit flows, we identify 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 5).

```
<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>
```

Listing 5. Ott big-step semantics of the if command

We then update to the program counter `pc` with the level of the expressions that are present in the rule (only `b` in this case).

```
<b, m> || <true, m>
E |- b : tb
<c1, m, E, pc |_| tb> || <skip, m1, E1, pc1>
-------------------------------------------
<if b then c1 else c2 end, m, E, pc> || <skip,
  m1, E1, pc>

<b, m> || <false, m>
E |- b : tb
<c2, m, E, pc |_| tb> || <skip, m2, E2, pc2>
-------------------------------------------
<if b then c1 else c2 end, m, E, pc> || <skip,
  m2, E2, pc>
```

Listing 6. Instrumented semantics of the if command

4 Current Status and Future Work

We have implemented a prototype of our algorithm 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 Our assumptions restricts the types of languages that can be used in Ott-IFC. For example, most functional languages would not be supported because, in those languages, functions 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: a monitor. 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, 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

[2] 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

[3] 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

[4] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. 186–199. https://doi.org/10.1109/CSF.2010.20

[5] 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

[6] 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

[7] 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