Ott-IFC: Automatically Generating Information-Flow Control Mechanism Specifications

***Abstract*—Developing information-flow control mechanisms can be a difficult and time-consuming task due to the numerous and subtle ways through which information may flow in a program, particularly when dealing with complex programming languages.**

In order to make this task easier, we present in this paper a tool called *Ott-IFC* that can automatically generate information-flow control mechanism specifications (e.g., semantics of a runtime monitor) from programming language specifications (i.e., syntax and semantics).

***Index Terms*—language-based security; information-flow control; development tool**

I. 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* [1] 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 be saved in a public file). Mechanisms that enforce such policies (e.g., [2], [3], [4], [5]) are called *information-flow control mechanisms*.

Most information-flow control mechanisms seek to enforce a policy called *non-interference* [6], 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* [7].

An insecure explicit information flow occurs when private information influences public information through a data dependency. For example in Listing 1, the value that is written to `publicFile` depends on the value of `x`, which in turn depends on the value of `privateValue`. Hence, any output of `x` will reveal something about `privateValue`.

```
x := privateValue + 42;
write x to publicFile
```

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 occurs when private information influences public information through the control-flow of the application. For example in Listing 2, the value that is written to `publicFile` depends on the condition `privateValue > 0`.

```
if (privateValue > 0) then
  write 0 to publicFile
else
  write 1 to publicFile
end
```

Listing 2. Insecure implicit flow

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

While techniques to prevent insecure explicit and implicit flows are well-known and widely used [1], they are, to the best of our knowledge, still being applied manually when designing information-flow control mechanisms. This can lead to errors, failed proof attempts and time wasted. In order to make this task less laborious and reduce the risk of errors, we present in this paper a tool called Ott-IFC that can, given a programming language's specification (i.e., syntax and semantics), automatically apply those techniques and generate information-flow control mechanism specifications.

Contributions:

- We present Ott-IFC and illustrate its use on a simple imperative language (Section II).
- We provide sketches for the proofs that show that the generated mechanisms are sound and do not change the semantics of the executed programs (Section III).

II. Overview of Ott-IFC

As the name implies, the specifications that Ott-IFC takes as input (and outputs) are written in Ott [8], [9]. Ott is a tool that can generate LaTeX, Coq [10] or Isabelle/HOL [11] versions of a programming language's specification. The specification is written in a concise and readable ASCII notation that resembles what one would write in informal mathematics (see the Listings of Section II).

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's specification.
3) Use Ott to export the mechanism's specification to LaTeX, Coq or Isabelle/HOL and complete the implementation.

For the moment, Ott-IFC requires that the language specification follows a certain format. Namely, we require:

- that the program configurations be of the form $\langle c, m, o \rangle$, where c is the command to be evaluated, m is the current memory and o is the current output trace;
- that the syntax be composed of *commands*, which may read/write the memory or produce outputs, and *expressions*, which may only read the memory;
- that the outputs are appended to the output trace using the notation $o :: (channel, value)$, where *value* represents the output's value and *channel* its location.

These requirements mean that Ott-IFC won't work with most functional languages; only imperative languages are supported. This is because in functional languages, the distinction between commands and expressions is less apparent and sometimes inexistent. Note that this is not a technical limitation, but rather a choice that we made in order to simplify the approach.

An example of a language that satisfies these requirements is the imperative language whose syntax is defined in Listing 3 and semantics in Listings 4, 5 and 6.

```
arith_expr, a ::= x | n | a1 + a2 | a1 * a2
bool_expr, b ::= true | false | a1 < a2
commands, c ::= stop | 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

```
%%% Variable %%%
m(x) = n
------------------------------------- :: lookup
<x, m, o> --> <n, m, o>

%%% Constant %%%
---------------------------------- :: int_constant
<n, m, o> --> <n, m, o>

%%% Addition %%%
<a1, m, o> --> <a1', m, o>
--------------------------------- :: add_aexp_aexp
<a1 + a2, m, o> --> <a1' + a2, m, o>

<a2, m, o> --> <a2', m, o>
---------------------------------- :: add_int_aexp
<n1 + a2, m, o> --> <n1 + n2, m, o>

n1 + n2 = n3
---------------------------------- :: add_int_int
<n1 + n2, m, o> --> <n3, m, o>

%%% Multiplication %%%
<a1, m, o> --> <a1', m, o>
--------------------------------- :: mult_aexp_aexp
<a1 * a2, m, o> --> <a1' * a2, m, o>

<a2, m, o> --> <a2', m, o>
---------------------------------- :: mult_int_aexp
<n1 * a2, m, o> --> <n1 * n2, m, o>

n1 * n2 = n3
---------------------------------- :: mult_int_int
<n1 * n2, m, o> --> <n3, m, o>
```

Listing 4. Ott small-step semantics of arithmetic expressions

```
%%% Lower Than %%%
<a1, m, o> --> <a1', m, o>
--------------------------------- :: lt_aexp_aexp
<a1 < a2, m, o> --> <a1' < a2, m, o>

<a2, m, o> --> <a2', m, o>
---------------------------------- :: lt_int_aexp
<n1 < a2, m, o> --> <n1 < n2, m, o>

n1 < n2 = true
------------------------------- :: lt_int_int_true
<n1 < n2, m, o> --> <true, m, o>

n1 < n2 = false
------------------------------- :: lt_int_int_false
<n1 < n2, m, o> --> <false, m, o>
```

Listing 5. Ott small-step semantics of boolean expressions

```
%%% Skip %%%
---------------------------------------- :: skip
<skip, m, o> --> <stop, m, o>

%%% Assignment %%%
<a, m, o> --> <a', m, o>
---------------------------------- :: assign_aexp
<x := a, m, o> --> <x := a', m, o>

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

%%% Sequence %%%
<c1, m, o> --> <c1', m', o'>
------------------------------------------ :: seq1
<c1 ; c2, m, o> --> <c1' ; c2, m', o'>

---------------------------------------- :: seq2
<skip ; c2, m, o> --> <c2, m, o>

%%% Read %%%
m(ch) = n
---------------------------------------- :: read
<read x from ch, m, o> --> <stop, m[x |-> n], o>

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

%%% If %%%
<b, m, o> --> <b', m, o>
---------------------------------------- :: if_eval
<if b then c1 else c2 end, m, o> -->
<if b' then c1 else c2 end, m, o>

---------------------------------------- :: if_true
<if true then c1 else c2 end, m, o> -->
<c1, m, o>

---------------------------------------- :: if_false
<if false then c1 else c2 end, m, o> -->
<c2, m, o>

%%% While %%%
---------------------------------------- :: while
<while b do c end, m, o> -->
<if b then c;while b do c end else skip end, m, o>
```

Listing 6. Ott small-step semantics of commands

To illustrate how our approach works, we will generate an information-flow control mechanism (a runtime monitor) from

this specification. Our approach can be decomposed into three steps: identifying commands and expressions, constructing evaluation-order graphs, rewriting the semantics.

A. Step 1: Identifying commands and expressions

To identify which non-terminals (e.g., `a`, `b`, `c`) of the syntax correspond to commands and which correspond to expressions, we need to identify the semantics rule associated to each non-terminal. To do so, we analyze each rule of the syntax and generate a set of strings that represents their possible values. For example, for the non-terminal `bool_expr` (a.k.a. `b`), it would return the set of strings {`true, false, a < a, x < a, a < x, n < a, a < n, a + a < a, ...`}. This set of string doesn't need to be exhaustive, only sufficiently so to be able to identify their use in the semantics.

Using this set of strings, we can then detect the rules associated to the non-terminals by looking that the first element of the initial state configurations. In the case of `bool_expr`, it would return the rules of Listing 5. Since none of these rules modify the memory or produce outputs, we can conclude that the non-terminals `bool_expr` and `b` are expressions.

Using the same reasoning, we can conclude that:

- the expression non-terminals are `arith_expr`, `a`, `bool_expr`, `b`, `x`, `n`, and that the rules associated to those are the ones in Listings 4 and 5;
- the commands non-terminals are `commands`, `c`, and that the rules associates to those are the ones in Listing 6.

B. Step 2: Constructing evaluation-order graphs

The next step consists in constructing *evaluation-order graphs* for each command. These graphs represent the order in which the semantics rules may be evaluated for a specific command (see Figure 1).

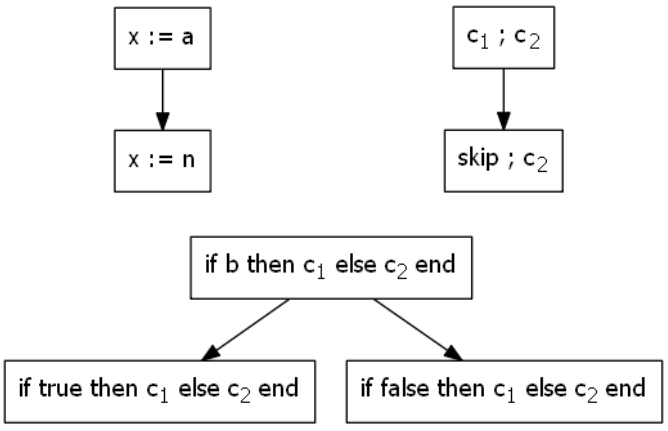

Fig. 1. Evaluation-order graphs of the assign, sequence and conditional commands.

To construct these graphs, we use the set of strings produced in Step 1 to detect that the rules associated to the assign command are `assign_aexp` and `assign_int`, and that `assign_aexp` will be evaluated first because `x := a` is more general than `x := n`. By more general, we mean that the set of strings that matches with `x := a` also matches with `x := n`, but the reverse is not true.

C. Step 3: Rule-based rewriting

The final step is to perform the actual rewriting of the semantics rules to insert a runtime monitor that enforces non-interference (i.e., a monitor that prevents insecure explicit and implicit flows of information). To do so, we try to replicate the thought process that a human would have when producing such a mechanism.

We start by inserting a program counter `pc`, and a typing environment `E`, which maps variables to their labels, in each command configurations. That is, configurations that have the form `<c, m, o>` are changed to `<c, m, o, pc, E>`.

To prevent explicit flows, we identify the commands that may modify the memory `m` (e.g., the assign command). In their rules, we update the modified variable's label with the label of the expression variables that are used in the rule. If they also produce an output, then we insert a guard condition to ensure that no leak of information occurs.

To prevent implicit flows, we identify the commands that may influence the control-flow of the application using the evaluation-order graphs. That is, commands for which a program configuration may lead to two different program configurations (e.g., the if command). In other words, commands for which there are multiple "terminal nodes" (i.e., nodes that have no successors) in the evaluation-order graph. We then update the program counter `pc` with the labels of the expression variables that are present in the rule.

So, for the language of our example, we would obtain the following rules (the changes are in red):

a) Skip: Because the `skip` command does essentially nothing, it does not read/write the memory or produce outputs, the only change in the rule is the addition of the `pc` and `E` variables (line 2).

```
1 ---------------------------------------- :: skip
2 <skip, m, o, pc, E> --> <stop, m, o, pc, E>
```

Listing 7. Ott-IFC's output for the "skip" command

b) Read: The `read` command reads the content of channel `ch` and assigns its value to `x`. For each expression variable found in the preconditions (i.e., `ch, n`), we insert a label variable definition (lines 2 and 3). Note that `|-` is the ASCII representation of $\vdash$ and `|_|` of the supremum operator $\sqcup$. Since the memory is modified, we propagate the label of the expressions that influence the assigned variable's value (i.e., the ones used in the rule) and take into account the context in which the assignment occurs (line 6).

```
1 m(ch) = n
2 E |- ch : lch
3 E |- n : ln
4 ---------------------------------------- :: read
5 <read x from ch, m, o, pc, E> -->
6 <stop, m[x |-> n], o, pc,
     E[x |-> pc |_| lch |_| ln]>
```

Listing 8. Ott-IFC's output for the "read" command

c) Write: The `write` command writes the value of `x` on the channel `ch`. Because this command produces an output (i.e., it appends a value to `o`), we add guard condition that ensures that no leak of information will occur at runtime (line 5). We do not update the label of the channel on which the output occurs even if it is modified because, if the execution is not stopped by the monitor, then it means that the channel's label is already greater than the label of the expression that is written.

```
1 m(x) = n
2 E |- x : lx
3 E |- n : ln
4 E |- ch : lch
5 lx |_| ln |_| pc <= lch
6 ---------------------------------------- :: write
7 <write x to ch, m, o, pc, E> -->
8 <stop, m[ch |-> n], o::(ch, n), pc, E>
```

Listing 9. Ott-IFC's output for the "write" command

d) Assign: While the `assign_aexp` rule does not directly modify the memory, one of its successor in the evaluation-order graph (i.e., `assign_int`) does. For this reason, we must update the label of the modified variable in, not only `assign_int`, but also `assign_aexp` (lines 6 and 12). This is to take into account the label of variables that are in the expression `a` (before they disappear). This means that, in this case, the generated monitor won't allow the label of variables to be "downgraded" as a result of an assignment.

```
1  <a, m, o> --> <a', m, o>
2  E |- x : lx
3  E |- a : la
4  ---------------------------------- :: assign_aexp
5  <x := a, m, o, pc, E> -->
6  <x := a', m, o, pc, E[x |-> lx |_| pc |_| la]>
7
8  E |- x : lx
9  E |- n : ln
10 ---------------------------------- :: assign_int
11 <x := n, m, o, pc, E> -->
12 <stop, m[x |-> n], o, pc, E[x |-> lx |_| pc |_| ln]
       >
```

Listing 10. Ott-IFC's output for the "assign" command

It may be interesting to note that, had the big-step version of the semantics been given to Ott-IFC instead, this would not be the case (see Listing 11). This is because in the big-step version, the assign command has only one rule. In other words, this means that the form of the specification given as input influences the permissiveness of the monitor that is returned by our tool (more on this in Section IV).

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

Listing 11. Ott-IFC's output the big-step version of the "assign" command

e) Sequence: The rules for the sequence do not modify the memory or produce outputs so, like the skip command, the only changes in the rules is the addition of the `pc` and `E` variables. Note however that we did not only update the configurations in the conclusion of the rule, but also in the precondition.

```
<c1, m, o, pc, E> --> <c1', m', o', pc, E'>
---------------------------------------- :: seq1
<c1 ; c2, m, o, pc, E> -->
<c1' ; c2, m', o', pc, E'>

---------------------------------------- :: seq2
<skip ; c2, m, o, pc, E> --> <c2, m, o, pc, E>
```

Listing 12. Ott-IFC's output for the "sequence" command

f) If: In this language, the `if` command is the only that can directly cause the control-flow of a program to branch-out. We can tell this by looking at its evaluation-order graph (Figure 1): there are two terminal nodes (i.e, the `if true` and `if false`). For this reason, we update the `pc` variables with the labels of the expression variables that are present in the rule (i.e., only `b` in this case).

We also insert a call to a function called `updateModifVars` which will have to be implemented by the user. What it does is that it identifies all the variables that could have been modified in either `c1` or `c2`, and update their labels with the label of `b` (line 3). This is to ensure that the labels of the variables after executing the `if` is always the same, no matter which branch that is taken during execution. Note that the command variables that are used in the rule (`c1` and `c2`) are detected the same way that expression variables are detected: using the set of strings generated in Step 1.

```
<b, m, o> --> <b', m, o>
E |- b : lb
E1 = updateModifVars(E, pc |_| lb, {c1,c2})
---------------------------------------- :: if_eval
<if b then c1 else c2 end, m, o, pc, E> -->
<if b' then c1 else c2 end, m, o, pc |_| lb, E1>

---------------------------------------- :: if_true
<if true then c1 else c2 end, m, o, pc, E> -->
<c1, m, o, pc, E>

---------------------------------------- :: if_false
<if false then c1 else c2 end, m, o, pc, E> -->
<c2, m, o, pc, E>
```

Listing 13. Ott-IFC's output for the "if" command

g) While: Like the skip and sequence commands, the semantics rule of the while command does not modify the memory or the output trace, hence the only change is the addition of the `pc` and `E` variables to the configurations. Note that, while the `pc` variable is not updated here with the label of the condition variable `b`, it will be when the `if` (present in the final configuration) is evaluated (line 3).

```
1 ---------------------------------------- :: while
2 <while b do c end, m, o, pc, E> -->
3 <if b then c ; while b do c end else skip end, m, o
     , pc, E>
```

Listing 14. Ott-IFC's output for the "while" command

III. Soundness

We have tested and validated that our approach works on multiple language specifications. However, to be sure that it works for *any* specification that meets our requirements, we must formally define non-interference and prove that the generated mechanisms are indeed sound. Before formally defining non-interference, we first need to introduce some helpful technical concepts.

For our purposes, we assume that the labels, which represent 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. $\forall l \in \mathcal{L}, L \sqsubseteq l \wedge l \sqsubseteq H$.

The *projection of trace o to label ℓ*, written $o \restriction \ell$, is its restriction to output events whose channels' labels are less than or equal to ℓ. Formally,

$$\epsilon \restriction \ell = \epsilon$$

$$(o :: (ch, v)) \restriction \ell = \begin{cases} (o \restriction \ell) :: (ch, v) & \text{if } \mathit{labelOfChan}(ch) \sqsubseteq \ell \\ o \restriction \ell & \text{otherwise} \end{cases}$$

where $\mathit{labelOfChan}(ch)$ denotes the label of channel ch (typically specified by the administrator).

We write $\langle c, m, \epsilon \rangle \downarrow o$ if execution of configuration $\langle c, m, \epsilon \rangle$ can produce trace o.

We say that two memories m and m' are ℓ-equivalent if they agree on the content of variables (including channel variables) whose labels are ℓ or lower.

Definition 1 (Non-interference[1])**.** *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 $o \restriction \ell$ is a prefix of $o' \restriction \ell$ (or vice versa).*

Theorem 1 (Soundness of enforcement)**.** *Let M be a monitor generated by Ott-IFC from a language specification S that satisfies the requirements of Section II, m be a memory and p be a program written in the language specified by S. The execution of p under monitor M and memory m, denoted $M(\langle p, m, \epsilon \rangle)$, will be non-interferent.*

Sketch. The arguments for the proof are similar to the ones presented in Section II.

We show that there can be no explicit flows because, any time that a variable's value is changed in the memory, we update its label (the only exception being channels) and that for any output, there is a guard condition.

We then show that there can be no implicit flows because, any time a branch in the control-flow could occur, we update the `pc` variable with the label of *all* expression variables that could have influenced the decision. We also point out that the `pc` variable is present every time that a label is updated and in every inserted guard conditions. □

We also show that the generated runtime monitor preserves the semantics of the original program. That is, a program p executed under the monitor M produces exactly the same output as p as long as it is allowed to continue; it may be stopped at some point to prevent a leak.

Theorem 2 (Semantics preservation)**.** *Let p be a program, m a memory, and o, o' output traces. Then*

$$M(\langle p, m, \epsilon \rangle) \downarrow o \Rightarrow \langle p, m, \epsilon \rangle \downarrow o$$
$$(\langle p, m, \epsilon \rangle \downarrow o \wedge M(\langle p, m, \epsilon \rangle) \downarrow o') \Rightarrow o \preceq o' \vee o' \preceq o$$

where $o' \preceq o$ means that o' is a prefix of o.

Sketch. The semantics generated by Ott-IFC are the essentially the same as the as the original semantics. The only difference being the additional variables in the configurations (`pc` and `E`), and the definitions of label variables and guard conditions in the preconditions. For this reason, the only non-trivial case is when we insert guard conditions to prevent commands that modify the output trace from leaking information.

If this guard condition inserted is true, then the output will occur as it would have in the non-monitored execution of the program. If the guard condition fails, then the execution of the monitored program will be stopped, and hence, no more outputs will occur (the non-monitored execution could produce other outputs). □

IV. Discussion

As far as we know, we are the first ones to propose a way to generate information-flow control mechanism specifications from programming language specifications. Our implementation is available online[2] and open-source. We expect that our tool will be particularly useful to researchers in language-based security that need to quickly develop mechanisms for complex languages.

For the moment, our tool can only produce one kind of mechanism: runtime monitors. We chose to generate runtime monitors because they seemed the simplest to generate: they are simply modified versions of the semantics given as input. However, the same logic could be used to produce other types of mechanisms, such as type systems.

Note that the mechanisms generated by our approach may not always be as permissive as those designed by humans. For example, when we update the labels of modified variables, we update it using the labels of *all* the expression variables found in the rule when in fact only one or two could be sufficient. If the generated mechanism is not permissive enough for the needs of a user, it could still be used as a foundation/starting point to design a more permissive mechanism. For those users, the work of Hritcu et al. [12] might be of interest. They show how to use QuickCheck, a property-based random-testing tool, to quickly verify that a mechanism correctly enforces non-interference. Their tool identifies errors during the design phase of the mechanism, thereby allowing users to postpone any proof attempts until they are confident of the mechanism's soundness.

[1] Also known as termination-insensitive non-interference

[2] The link will be made available once the peer-review process is finished

One interesting thing about our approach, which we briefly mentioned in Section II, is that the form of the semantics given as input will have an impact on the permissiveness of the generated monitor. For instance, the monitor that is generated from the small-step semantics of the imperative language used in this paper is slightly less permissive than the one generated from the big-step semantics. Also, even though we have not yet added support for termination-sensitivity [13], we believe that monitors generated from small-step semantics will be naturally termination-sensitive while those generated from big-step semantics will be naturally termination-insensitive. This is because the big-step semantics assume that the execution of every command eventually terminates, while this is not the case in small-step semantics.

A. Future Work

Before our tool can be of real use to researchers in language-based security, some work remains to be done.

a) Language Support: Our requirements on specifications 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.

b) Parameterization: For the moment, Ott-IFC only generates one type of information-flow control mechanism: runtime monitors. We plan on parameterizing 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 [13], progress-sensitivity [14], value-sensitivity [15]).

c) Generating Formal Proofs: As previously mentioned, we expect that some users will use the generated mechanisms 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).

d) Verifying Existing Mechanisms: The same rules that Ott-IFC uses to generate mechanisms could be used to verify the soundness of existing mechanisms and identify potential errors. For example, we could raise a warning if an output is produced but no guard condition is present.

Acknowledgments

We would like to the anonymous reviewers for their comments.

References

[1] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003. [Online]. Available: https://doi.org/10.1109/JSAC.2002.806121

[2] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 167–188, 1996. [Online]. Available: https://doi.org/10.3233/JCS-1996-42-304

[3] A. Chudnov and D. A. Naumann, "Information flow monitor inlining," in *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, 2010, pp. 200–214. [Online]. Available: https://doi.org/10.1109/CSF.2010.21

[4] A. Askarov, S. Chong, and H. Mantel, "Hybrid monitors for concurrent noninterference," in *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, 2015, pp. 137–151. [Online]. Available: https://doi.org/10.1109/CSF.2015.17

[5] A. Bedford, S. Chong, J. Desharnais, E. Kozyri, and N. Tawbi, "A progress-sensitive flow-sensitive inlined information-flow control monitor (extended version)," *Computers & Security*, vol. 71, pp. 114–131, 2017. [Online]. Available: https://doi.org/10.1016/j.cose.2017.04.001

[6] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, 1982, pp. 11–20. [Online]. Available: https://doi.org/10.1109/SP.1982.10014

[7] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976. [Online]. Available: http://doi.acm.org/10.1145/360051.360056

[8] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa, "Ott: Effective tool support for the working semanticist," *J. Funct. Program.*, vol. 20, no. 1, pp. 71–122, 2010. [Online]. Available: https://doi.org/10.1017/S0956796809990293

[9] "Ott," http://www.cl.cam.ac.uk/~pes20/ott/.

[10] "The coq proof assistant," https://coq.inria.fr/.

[11] "Isabelle/hol," https://isabelle.in.tum.de/.

[12] C. Hritcu, L. Lampropoulos, A. Spector-Zabusky, A. A. de Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis, "Testing noninterference, quickly," *J. Funct. Program.*, vol. 26, p. e4, 2016. [Online]. Available: https://doi.org/10.1017/S0956796816000058

[13] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, 2008, pp. 333–348. [Online]. Available: https://doi.org/10.1007/978-3-540-88313-5_22

[14] S. Moore, A. Askarov, and S. Chong, "Precise enforcement of progress-sensitive security," in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 881–893. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382289

[15] D. Hedin, L. Bello, and A. Sabelfeld, "Value-sensitive hybrid information flow control for a javascript-like language," in *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, 2015, pp. 351–365. [Online]. Available: https://doi.org/10.1109/CSF.2015.31