Extracting Conditional Confidentiality Policies

Michael Carl Tschantz and Jeannette M. Wing

May 10, 2008
CMU-CS-08-127

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present a static analysis that extracts from a program's source code a sound approximation of the most restrictive *conditional confidentiality policy* that the program obeys. To formalize conditional confidentiality policies, we present a modified definition of noninterference that depends on runtime information. We implement our analysis and experiment with the resulting tool on C programs. While we focus on using our analysis for policy extraction, the process can more generally be used for information flow analysis. Unlike traditional information flow analysis that simply states what flows are possible in a program, our tool also states what conditions must be satisfied by an execution for each flow to be enabled. Furthermore, our analysis is the first to handle interactive I/O while being compositional and flow sensitive.

This research was partially sponsored by the Army Research Office through grant number DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab and by a generous gift from the Hewlett-Packard Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity. This material was based on work partially supported by the National Science Foundation, while the second author is working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Sections 2 and 3 of this document includes work first published by the authors as a techincal report on February 9, 2007 [TW07].

Keywords: Confidentiality, Privacy, Noninterference, Information Flow

```
read(roles, "roles.db");
read(xray, "xray.jpg");
read(bill, "bill.txt");
read(login, stdin);
if(roles[login] == "doc")
    out := append(bill,xray);
else
    out := bill;
write(out, stdout);
```

Figure 1: Code Snippet for the Doctor's Office

1 Introduction

On-line banking, databases of electronic medical records, and social networking sites all store large amounts of sensitive information. Many of these systems run legacy code written without a systematic means of specifying or enforcing confidentiality policies. Instead, these programs attempt to protect confidentiality using *ad hoc* approaches such as conditional statements that check if the user is authorized to access sensitive information. Since such checks are spread throughout the program, determining the confidentiality policy that a program enforces is a difficult task.

Users should be wary of these programs. With no specification of how the program protects confidentiality, the user would benefit from a summary of the conditions under which the program will release his information. We have developed a tool that automatically produces such a summary, or *conditional confidentiality policy*, from source code. Since confidentiality is closely related to information flow, our tool is more general: it performs conditional information flow analysis.

Motivating Example. Consider a doctor's office where the doctor takes a digital X-ray of a patient. The doctor stores the X-ray and the bill for the procedure in a computer system. At this point, the patient becomes worried about the confidentiality of his X-ray and asks the doctor how the system will protect it. That is, the patient wants to know what confidentiality policy the system obeys.

Given that the system runs legacy code, neither the doctor nor his system administrator are exactly sure how the system treats X-rays. To answer this question, the administrator looks at the relevant part of the program (shown in Figure 1) and reasons as follows: First, the code loads a database that assigns roles to user logins. Then it loads the X-ray and the bill. Next it receives the login of the user via `stdin`. If the roles database lists the login as one of a doctor, the program will store the bill and the X-ray in the variable `out`; otherwise, just the bill is stored. Finally, the program prints the contents of `out` to the user via `stdout`. Since `out` holds the X-ray only if the user provides a doctor's login, the administrator concludes that program only allows the doctor access to the X-ray and notifies the patient that the confidentiality of his X-ray is protected. (Proving that only doctors login as doctors is an issue of *authentication*, not *authorization*, and outside the scope of this paper.)

Information flow analysis formalizes the administrator's reasoning. He followed the flow of the X-ray from the input file `xray.jpg` to the variable `xray` to the variable `out` to the output buffer `stdout`. However, his reasoning differed from standard information flow analysis in one important aspect:

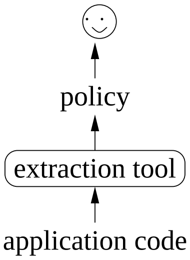

Figure 2: Possible uses with one body of code.

he noted that the flow from `xray` to `out` is only possible if the condition `roles[login] == "doc"` is true. That is, his information flow analysis kept track of the conditions that enabled the flow.

While the administrator can reasonably run such a conditional information flow analysis by hand on the small fragment of code shown in Figure 1, he would rather have a tool to automate the process as much as possible (Fig. 2). Unfortunately, while many tools for information flow analysis exist, none keep track of the conditions that enable each flow of information that is relevant to confidentiality.

In this paper, we present such a tool. Given the above program, `xray.jpg` as the source, and `stdout` as the sink, our tool will find all flows of information from `xray.jpg` to `stdout` and report which conditions must hold to enable each flow. In this case, it would report that `xray.jpg` only flows to `stdout` when `roles[login] == "doc"` holds.

Before we can describe our approach to extracting confidentiality policies as conditional information flows, we must first consider what it means for a user's information to remain confidential or, equivalently, what it means for information to flow from an input to an output.

Confidentiality Requirements. Confidentiality requires that sensitive information does not flow to untrusted users. What exactly "flow to" means varies from context to context. We call each possible meaning a different *confidentiality requirement*.

One of the most well known and earliest confidentiality requirements is *noninterference* as defined by Goguen and Meseguer [GM82]. Informally, this confidentiality requirement holds if the outputs the program provides to untrusted users remain the same whether the program received sensitive information as input or not.

Such a requirement is often too stringent, that is, it places so much emphasis on privacy that it prevents some systems from achieving a reasonable level of functionality. In many realistic systems, allowing an untrusted user to learn that sensitive information has entered the system is acceptable as long as the untrusted user does not learn about the contents of the input. For example, the patient from the doctor's office example would not mind if the billing department of the doctor's office learns that he had an X-ray taken: it is the image of the X-ray he wants protected. In Section 2.1, we provide more examples of such systems.

Motivated by these examples, we present a weakened form of noninterference that protects only the contents of inputs. We call this weakened confidentiality requirement *incident-insensitive noninterference* since untrusted users are allowed to learn of the incident of the input. Likewise, we call the original noninterference requirement of Goguen and Meseguer *incident-sensitive noninterference*.

Conditional Confidentiality. The confidentiality requirements described above have not been conditional on user input: they require that information does not flow regardless of user inputs. As shown in our motivating example, the presentation of certain inputs, such as a doctor's login, can change the access rights of a user. Expressing such scenarios requires *conditional confidentiality requirements*. Such a requirement is equivalent to a *conditional information flow*, a flow of information that only occurs when some condition is met at runtime. Along with noninterference, Goguen and Meseguer introduced a form of conditional confidentiality requirement [GM82]. Our definition, presented in Section 3, generalizes theirs.

Policy Extraction by Conditional Information Flow Analysis. Using our formalization of confidentiality, we develop a static analysis for finding all the conditional information flows in a program. Equivalently, we extract from the program the conditional confidentiality requirements it obeys. We call these requirements collectively a *policy*. Thus, we say our analysis performs *policy extraction*.

The algorithm presents the user with the key conditional statements of the program that affect whether an information flow will occur during program execution. The algorithm tracks the flow of information through the program in a manner similar to type systems that track information flow [SM03]. However, our approach is flow sensitive allowing the same variable to carry both sensitive and nonsensitive information without considering the nonsensitive information sensitive. While this feature matters little in the context of writing a program with type analysis in mind, it becomes important while extracting policies from legacy code.

Road Map and Contributions. To mirror the development of this introduction we first motivate and present incident-insensitive noninterference in Section 2. Then we present our formulation of conditional confidentiality in Section 3. Using this formulation, we formally present our conditional information flow analysis as a set of inference rules that operate on a small programming language in Section 4. In Section 5, we discuss our implementation for a subset of C and experimented with it on C programs. We discuss many applications of our algorithm and tool in Section 6 and related work in Section 7. We end the paper with directions for future work in Section 8.

Our work provides two novel contributions:

- We have identified the problem of policy extraction and formalized it using conditional incident-insensitive noninterference.
- We provide a static analysis for policy extraction with an implementation.

Our algorithm, presented in Section 4, is the first to our knowledge to handle interactive I/O and be compositional while also being flow sensitive. It is the first to extract the conditions that enable both direct flows (from assignments) and indirect flows (from conditional statements) of information. Extracting both types of flows is crucial for policy extraction. The only other work to extract the conditions that enable information flows does so only for direct flows of information [KR07]. Most previous work that use the conditional statements in a program to improve information flow analysis, such as path conditions [SRK06], only rule out infeasible flows of information. Unlike our work, they do not characterize *when* a flow may be possible.

2 Confidentiality Policies

Before we can formalize the idea of conditional information flows, we must formalize what counts as a flow of information. Since the primary application of our analysis is confidentiality policy extraction, we start with a well known formalization of confidentiality and introduce adjustments as needed.

2.1 What is Confidentiality?

Goguen and Meseguer introduced *noninterference* to formalize when a sensitive input to a system with multiple users is protected from untrusted users of that system [GM82]. Intuitively, noninterference requires that the system behaves identically from the perspective of untrusted users regardless of any sensitive inputs to the system.

This requirement is so strong that an untrusted user is not even allowed to know if the system has received any sensitive inputs. For example, consider the following simple program:

```
bool in = load("secret-file.db");
print("hi");
```

The first line reads in the contents of a secret file. The second line simply prints “hi” to the untrusted user. If we model the reading of the secret file as receiving sensitive input, then this program fails to meet the requirements of noninterference as defined by Goguen and Meseguer. The reason is that the untrusted user does not see the output “hi” unless the system has received the sensitive input, which allows the `load` statement to stop blocking and terminate. Thus, the untrusted user has learned that the system has received sensitive information. This violation occurs even though the untrusted user clearly does not learn anything about the contents of `secret-file.db`.

We believe that in many cases allowing an untrusted user to know that sensitive information has entered the system is acceptable as long as the untrusted user does not learn the contents of this information. Many such examples exist in practice:

- The simple example above may be extended to a realistic one: Consider a web server for on-line banking that upon startup receives financial records from a secure database before answering any queries from users. Even if the outputs that the unauthenticated users see reveal nothing about the contents of the financial records, noninterference is violated because the unauthenticated users know that the server has loaded sensitive financial records.

- The motivating example in Section 1 provides another example: If a clerk from the billing department logs in in and is not a doctor, he will never see the X-ray. However, he will learn that the doctor has stored one in the system.

- Consider a student who is applying for graduate school on-line. During the application process, both the student and the professor recommending him must enter information into the application database. Once the recommending professor has finished, the student receives a notice stating that the graduate school has received his recommendation. The applicant is not allowed access to his recommendation, but simply learning that the professor has entered the sensitive recommendation is enough to violate noninterference.

These examples make clear that often simply learning that some sensitive input has entered the system does not provide the untrusted users enough information to constitute a violation of confidentiality. However, not just Goguen and Meseguer's noninterference, but most confidentiality requirements (*e.g.*, generalized noninterference [McC87, McC88], restrictiveness [McC87, McC88, McC90] and separability [McL94]) are *incident sensitive*: they prohibit untrusted users from learning that any sensitive input has taken place.

What we desire are *incident-insensitive* requirements, ones that allow untrusted users to learn that sensitive input has taken place while protecting the *contents* of these sensitive inputs. Intuitively, a system obeys incident-insensitive noninterference if the contents of sensitive inputs have no effect on the outputs that an untrusted user sees. That is, an untrusted user must see the same outputs regardless of which sensitive inputs the system received. The untrusted user is, however, allowed to learn that sensitive information has entered the system.

While incident-insensitive requirements have appeared before [WJ90, OCC06], we are the first, to our knowledge, to distinguish them from incident-sensitive requires and explain their benefits.

2.2 Noninterference Formalized

First, we present a formal model of systems. Then, to make the differences clear, we present formalizations of both Goguen and Meseguer's incident-sensitive noninterference and our strictly weaker yet still safe incident-insensitive noninterference. For simplicity we limit ourselves to deterministic systems with only two sensitivity levels (high and low). In a related technical report, we provide a presentation not subject to these restrictions [TW07].

System Model. When modeling a system, we focus on the program controlling the system. The program is modeled as a deterministic transducer. The program accepts inputs from some set I. Each input is marked with either H for high-level (sensitive) or L for low-level (nonsensitive). The system provides outputs from a set O to a high-level trusted user and a low-level untrusted user. A high-level user is free to enter low-level information as when a high-level doctor enters a low-level bill. However, the system will only protect information entered at the high-level.

Given a system s and an interleaving of high- and low-level inputs $\vec{i}$ in I^*, we represent the behavior of s on $\vec{i}$ as $[\![s]\!](\vec{i})$. The behavior of s is an interleaving of the input sequence $\vec{i}$ with high- and low-level outputs from O. Thus, $[\![s]\!]$ is a function from I^* to A^* where A is the set of I/O actions, that is, $A = I \cup O$.

Incident-Sensitive Noninterference. For $\vec{a}$ in A^*, let $\lfloor\vec{a}\rfloor$ represent $\vec{a}$ restricted to only those actions that are low level. That is, it "purges" all high-level actions. Formally,

$$\lfloor a{:}\vec{a}\rfloor = \begin{cases} a{:}\lfloor\vec{a}\rfloor & \text{if } \mathsf{level}(a) = \mathsf{L} \\ \lfloor\vec{a}\rfloor & \text{otherwise} \end{cases}$$
$$\lfloor[]\rfloor = []$$

where $\mathsf{level}(a)$ is the level of a, $[]$ is the empty sequence, and $a{:}\vec{a}$ is the sequence $\vec{a}$ with a prepended to it.

Definition 1. *A system s obeys incident-sensitive noninterference iff for every two input sequences $\vec{i}_1$ and $\vec{i}_2$ in I^*,*

$$\lfloor\vec{i}_1\rfloor = \lfloor\vec{i}_2\rfloor \textit{ implies } \lfloor[\![s]\!](\vec{i}_1)\rfloor = \lfloor[\![s]\!](\vec{i}_2)\rfloor$$

Intuitively, this definition says that if the same low-level inputs are provided to the system, then the same low-level outputs should result from the system with complete disregard for what if any high-level inputs were provided to the system. Thus, the low-level user can determine nothing about the high-level inputs, not even their existence.

Incident-Insensitive Noninterference. The requirement that any two input sequences $\vec{i}_1$ and $\vec{i}_2$ produce the same outputs provided $\lfloor \vec{i}_1 \rfloor = \lfloor \vec{i}_2 \rfloor$ is too strong. Consider when $\vec{i}_1 = [\mathsf{a}^{\mathsf{H}}]$ and $\vec{i}_2 = []$ where $\mathtt{a}^{\mathsf{H}}$ is a high-level input with the contents `a`. $\vec{i}_1$ provided to the two-line program showed in Section 2.1 would result in the output of $\mathtt{hi}^{\mathsf{L}}$ since $\vec{i}_1$ contains a high-level input that allows the `load` statement to stop blocking and the `print` statement to execute. However, $\vec{i}_2$ would result in no output since it contains no high-level inputs to allow the `load` to stop blocking. Thus, $\vec{i}_1$ and $\vec{i}_2$ demonstrate that this program violates incident-sensitive noninterference. Yet, as argued earlier, we would not consider this program to leak high-level information.

Thus, we relax the noninterference requirement by raising the bar on how two input sequences must be similar before we require them to result in identical low-level outputs. We now require that not only do the two input sequences have the same low-level inputs, but also the same number and interleaving of high- and low-level inputs.

We formalize this notion with the *blur* operator $\wr \cdot \int$. Like the purge operator $\lfloor \cdot \rfloor$, blur takes an input sequence and leaves the low-level inputs unchanged. However, rather than removing the high-level inputs, it "blurs" them. That is, it replaces them with a symbol $*^{\mathsf{H}}$ that only carries the information that a high-level input was there. For example, both $\wr [\mathsf{a}^{\mathsf{H}}, \mathsf{b}^{\mathsf{L}}] \int$ and $\wr [\mathsf{c}^{\mathsf{H}}, \mathsf{b}^{\mathsf{L}}] \int$ blur to $[*^{\mathsf{H}}, \mathsf{b}^{\mathsf{L}}]$ while $\wr [\mathsf{b}^{\mathsf{L}}] \int$ blurs to $[\mathsf{b}^{\mathsf{L}}]$. We will define incident-insensitive noninterference to require that a program produce the same low-level outputs on $[\mathsf{a}^{\mathsf{H}}, \mathsf{b}^{\mathsf{L}}]$ and $[\mathsf{c}^{\mathsf{H}}, \mathsf{b}^{\mathsf{L}}]$ while being free to produce different outputs on $[\mathsf{b}^{\mathsf{L}}]$. Formally,

$$\wr i{:}\vec{i} \int = \begin{cases} i{:}\, \wr \vec{i} \int & \text{if } \mathsf{level}(i) = \mathsf{L} \\ *^{\mathsf{H}}{:}\, \wr \vec{i} \int & \text{otherwise} \end{cases}$$

$$\wr [] \int = []$$

Definition 2. *A system s obeys incident-insensitive noninterference iff for every two input sequences $\vec{i}_1$ and $\vec{i}_2$ in I^*,*

$$\wr \vec{i}_1 \int = \wr \vec{i}_2 \int \textit{ implies } \lfloor [\![s]\!](\vec{i}_1) \rfloor = \lfloor [\![s]\!](\vec{i}_2) \rfloor$$

The following theorem shows that incident-sensitive noninterference is a strictly stronger property than incident-insensitive noninterference.

Theorem 1. *If a system obeys incident-sensitive noninterference, then it will obey incident-insensitive noninterference; the converse is not true.*

Proof. It can be shown by induction that for all $\vec{i}_1$ and $\vec{i}_2$, if $\wr \vec{i}_1 \int = \wr \vec{i}_2 \int$, then $\lfloor \vec{i}_1 \rfloor = \lfloor \vec{i}_2 \rfloor$. Thus, incident-sensitive noninterference requires at least as many input sequences to look the same. The example program at the beginning of this section provides a counterexample to the converse. (See [TW07] for a more formal proof.) □

3 Conditional Policies

While the above formulation of incident-insensitive noninterference is sufficient to formalize standard information flow analysis, we desire to formalize *conditional* information flow analysis to extract *conditional* policies.

3.1 Motivation

We motivate the need for conditional confidentiality by referring to the doctor example from the introduction. As described before, the system should allow access to the X-ray only if the user is a doctor.

To model this program, let `xray.jpg` be a high-level input. Let all other inputs (`roles.db`, `bill.txt`, and the login from `stdin`) be low-level. Let `stdout` send output to a low-level user.

The program does not obey incident-insensitive noninterference. To see this, consider any input sequence in which the login provided by `stdin` is one that the roles database maps to `"doc"`. In this case, the program sends the contents of `xray.jpg` to the low-level user via `stdout`. Thus, two such sequences of input will produce different low-level output even if identical in all ways except for the contents of `xray.jpg`.

Despite not obeying noninterference, the program does not violate confidentiality since it only allows doctors to view the X-ray. To capture such situations, Goguen and Meseguer presented a conditional version of incident-sensitive noninterference [GM82]. Informally, it allows a high-level input to be accessible to the low-level user if the inputs that precede it satisfy some predicate. We generalize their definition to allow this predicate to depend on inputs that occur after the high-level input. The program must protect the high-level input until this predicate is satisfied. If the future inputs never satisfy the predicate, the high-level input will always be protected. Our generalization allows the high-level input `xray.jpg` of our motivating example to be released to the low-level user if that user provides a doctor's login, an event that happens after `xray.jpg` is loaded.

3.2 Formalization

We first model the predicates used to determine if a high-level input should be protected as *policies*. A policy represents a predicate by being the set $P \subseteq I^*$ that contains each and every input sequence that satisfies that predicate. If an input sequence is in P, then that input sequence enables the low-level user to gain access to the high-level inputs. Thus, those $\vec{i}$ in P are exempt from the requirements of noninterference. Since each $\vec{i}$ in P represents an exception to a low-level user not gaining access to the high-level inputs, the larger P is, the more permissive P is. Likewise, the smaller P becomes, the more restrictive P becomes meaning that fewer and fewer programs would obey P.

We define conditional incident-insensitive noninterference using prefix relation $\sqsubseteq$ on action sequences. That is, $\vec{a}_1 \sqsubseteq \vec{a}_2$ if $\vec{a}_1$ is a prefix of $\vec{a}_2$ or equal to $\vec{a}_2$.

Definition 3. *A system s obeys a conditional incident-insensitive noninterference under the policy P iff for every two input sequences $\vec{i}_1$ and $\vec{i}_2$ in I^*, either $\vec{i}_1 \in P$ or*

$$\wr\vec{i}_1\int = \wr\vec{i}_2\int \ \textit{ implies } \lfloor[\![s]\!](\vec{i}_1)\rfloor \sqsubseteq \lfloor[\![s]\!](\vec{i}_2)\rfloor$$

This definition differs from the unconditional definition in two ways. First, we check if $\vec{i}_1$ is in P and hold it to no further requirements if it is. Second, rather than checking if $\lfloor[\![s]\!](\vec{i}_1)\rfloor$ is equal to $\lfloor [\![s]\!](\vec{i}_2) \rfloor$, we check if $\lfloor [\![s]\!](\vec{i}_1) \rfloor$ is a prefix of $\lfloor [\![s]\!](\vec{i}_2) \rfloor$. Given that we check if $\vec{i}_1$ is in P rather than $\vec{i}_2$, it is key that we put $\lfloor [\![s]\!](\vec{i}_1) \rfloor$ on the left-hand side of the prefix relation. We are using $\vec{i}_1$ to stand for the input sequence the system actually received. If $\vec{i}_1$ is not in P and $\wr \vec{i}_1 \int = \wr \vec{i}_2 \int$, then the low-level user should be incapable of determining that the system did not receive $\vec{i}_2$. We use the prefix relation since the low-level user cannot observe termination and, thus, if he sees a sequence of outputs consistent with a prefix of $\lfloor [\![s]\!](\vec{i}_2) \rfloor$, he cannot rule out that the system received $\vec{i}_2$ and is just being slow about producing the rest of the output.

4 Policy Extraction

Now that we have formalized conditional confidentiality policies in terms of conditional incident-insensitive noninterference, we can formally present an analysis that extracts them from source code. Our analysis finds information flows and the conditions that determine whether each flow will actually occur during an execution.

Given two policies P_1 and P_2 such that $P_1 \subseteq P_2$, P_1 would be more restrictive (allow fewer flows of information) than P_2. Since obeying P_1 would imply obeying P_2, we would rather extract P_1. Indeed, our goal is to extract the most restrictive (smallest) policy that the program obeys. However, because of undecidability, we must settle for a sound over-approximation of the most restrictive policy.

Rather than extract the policy P directly, we provide a *syntactic policy* $\hat{P}$. $\hat{P}$ describes what paths though the control flow graph yield information flows. For example, in our motivating example from Section 1, only executions in which the `then` branch is taken result in information flows from the high-level `xray.jpg` to the low-level user. Thus, the extracted syntactic policy $\hat{P}$ for that program would include the key condition that `roles[login] == "doc"` must hold for an information flow to be possible.

After describing a programming language over which the analysis will soundly work and formalizing the idea of a syntactic policy, we give a formal description of our analysis as a set of inference rules and demonstrate them on our motivating example. In Section 5, we discuss the actual implementation for a subset of C.

4.1 WhileIO

Due to the many complex behaviors of C (*e.g.*, arbitrary dereferencing, segfaults, stack smashing), we have little hope of creating a sound yet accurate analysis for C. Thus, to explain the theoretical aspects of our algorithm, we instead use the simple language WhileIO. WhileIO offers `while` loops, `if` statements, and operators for input and output. The syntax of WhileIO consists of statements s and expressions e:

$$\begin{aligned} s &::= v\texttt{:=}e \mid \texttt{write(}e\texttt{, }d\texttt{)} \mid \texttt{read(}v\texttt{, }d\texttt{)} \mid s\texttt{;}s \\ &\quad\mid \texttt{if(}e\texttt{)\{}s\texttt{\}\{}s\texttt{\}} \mid \texttt{while(}e\texttt{)\{}s\texttt{\}} \\ e &::= v \mid n \mid e\texttt{+}e \mid \cdots \end{aligned}$$

where v ranges over a set of variable names V and n over a set of numbers N. d ranges over a set of domains D that represent I/O buffers. Since only the confidentiality level of each domain matters to the analysis, we do not distinguish between the two. In the limited case of having only high-level and low-level information, D would be $\{\mathsf{H}, \mathsf{L}\}$.

We require that expressions e always evaluate to a number and have no side effects. Statements always evaluate to unit (often called "void"). A program is just a single statement.

A program goes though a sequence of *reductions* that produce outputs and consume inputs while altering the contents of memory. We call a finite sequence of reductions a *trace*. We denote by $[\![s]\!](\vec{i})$ the I/O actions found in order in the trace generated by s given the input sequence $\vec{i}$. Appendix A provides a more detailed semantics for WhileIO.

4.2 Syntactic Policies

A syntactic policy $\hat{P}$ describes a set of paths through the control flow graph (CFG) of a program. A normal policy P may be recovered from a syntactic policy $\hat{P}$ by finding those input sequences that result in the program taking one of these paths.

Since a program with loops may have an infinite number of paths, a syntactic policy does not describe these paths directly. Rather, it provides a logical formula that constrains which branches of control statements (`if` and `while`) the paths include. To describe only those paths that take the `then` branch of some `if` statement, we use a *control constraint* c. Let e be the expression that controls which branch of the `if` statement is taken. The control constraint c is represented as $e{\mapsto}\mathsf{T}$. (We assume that each controlling expression e occurs only once in each program. Thus, they uniquely identify the `if` statement that it controls.) If instead we are interested in only paths that take the `else` branch, we would use the control constraint $e{\mapsto}\mathsf{F}$. Likewise for a `while` loop with the controlling expression e, we use $e{\mapsto}\mathsf{T}$ for the case where the body is entered and $e{\mapsto}\mathsf{F}$ for the case where body is not entered.

A syntactic policy $\hat{P}$ is a boolean formula over control constraints. The policy describes those paths in the CFG that satisfies it. For example, given the program:

```
while(x>0){
  if(y<3){ ... }{ ... }
}
```

the syntactic policy

$$\hat{P} \;=\; \mathsf{x{>}0}{\mapsto}\mathsf{F} \;\vee\; (\mathsf{x{>}0}{\mapsto}\mathsf{T} \wedge \mathsf{y{<}3}{\mapsto}\mathsf{T} \wedge \mathsf{y{<}3}{\mapsto}\mathsf{F})$$

identifies those paths in which either the loop body is never entered, or the loop body is entered and both the `then` and `else` branches of the `if` statement are taken. Taking both the `then` and `else` branches makes sense since the `if` statement lies within a loop body. ($e{\mapsto}\mathsf{F}$ is *not* the negation of $e{\mapsto}\mathsf{T}$.)

We write $\hat{P}_1 \Rightarrow \hat{P}_2$ if any path satisfying $\hat{P}_1$ also satisfies $\hat{P}_2$. If $\hat{P}_1$ logically implies $\hat{P}_2$, then $\hat{P}_1 \Rightarrow \hat{P}_2$ and we say that $\hat{P}_1$ is more restrictive than $\hat{P}_2$. We denote by $\langle\langle s\rangle\rangle(\vec{i})$, the most restrictive control constraint that the execution of s on $\vec{i}$ satisfies. Intuitively, $\langle\langle s\rangle\rangle(\vec{i})$ records every branch taken by the execution of s on $\vec{i}$.

Given a syntactic policy $\hat{P}$ and program s, we may identify the input sequences that result in s satisfying $\hat{P}$. We denote that policy P as $[\![\hat{P}]\!]^s$ where $[\![\hat{P}]\!]^s = \{\, \vec{i} \in I \mid \langle\langle s\rangle\rangle(\vec{i}) \Rightarrow \hat{P}\,\}$.

4.3 Algorithm Specification

We now present an inference system that produces a syntactic policy $\hat{P}$ such that $[\![\hat{P}]\!]^s$ is a sound over-approximation of the most restrictive policy P that a given program s obeys. Section 4.4 provides an example that explains these rules.

We call variables and domains collectively *identifiers*. For simplicity of presentation, we add a distinguished identifier nt to the set of identifiers. We use nt to track when the value of a sensitive input can result in nontermination. We assume that nt shows up nowhere in the analyzed program.

Let $\mathsf{ref}(e)$ be the set of identifiers referenced by an expression e. That is, $\mathsf{ref}(e)$ must contain all the identifiers whose value affects the value of e. Let $\mathsf{def}(s)$ be the set of identifiers defined by a statement s. That is, $\mathsf{def}(s)$ must contain all the identifiers whose value changes as a result of s executing. In both cases, we compute over-approximations. For ref we use

$$\begin{aligned}
\mathsf{ref}(n) &= \emptyset \\
\mathsf{ref}(v) &= \{v\} \\
\mathsf{ref}(e_1\texttt{+}e_2) &= \mathsf{ref}(e_1) \cup \mathsf{ref}(e_2) \\
&\ \ \vdots
\end{aligned}$$

For def we use

$$\begin{aligned}
\mathsf{def}(v\texttt{:=}e) &= \{v\} \\
\mathsf{def}(\texttt{read(}v\texttt{,}d\texttt{)}) &= \{v, d\} \\
\mathsf{def}(\texttt{write(}e\texttt{,}d\texttt{)}) &= \{d\} \\
\mathsf{def}(s_1\texttt{;}s_2) &= \mathsf{def}(s_1) \cup \mathsf{def}(s_2) \\
\mathsf{def}(\texttt{if(}e\texttt{)}\{s_1\}\{s_2\}) &= \mathsf{def}(s_1) \cup \mathsf{def}(s_2) \\
\mathsf{def}(\texttt{while(}e\texttt{)}\{s_1\}) &= \mathsf{def}(s_1) \cup \{\mathsf{nt}\}
\end{aligned}$$

Note that both v and d are in $\mathsf{def}(\texttt{read(}v\texttt{,}d\texttt{)})$ since the first element of the input buffer d is removed and assigned to v, altering both d and v. We add nt to $\mathsf{def}(\texttt{while(}e\texttt{)}\{s_1\})$ since `while` statements can bring about nontermination.

We use the judgment $x[s]^{\hat{P}}y$ where s is a statement, $\hat{P}$ a syntactic policy, and x and y are identifiers. Informally, $x[s]^{\hat{P}}y$ means that the value of x before the execution of s may affect the value of y after the execution of s if $\hat{P}$ is satisfied by the execution of s. In particular, $x[s]^{\hat{P}}\mathsf{nt}$ means that the value of x before s affects whether s will terminate. On the other hand, $\mathsf{nt}[s]^{\hat{P}}y$ means that non-termination before s is called (*i.e.*, s not getting a chance to execute) will result in y having a different value than if s did get to execute.

In Table 1, we provide the inference rules for $x[s]^{\hat{P}}y$. If the premises of a rule (the formulae above the bar) hold, then the conclusion (the formula below the bar) holds. A rule with no premises always holds.

The rules for `if` and `while` statements constrain the extracted policy by adding control constraints. For example, rule (9) adds the control constraint $e{\mapsto}\mathsf{T}$ to indicate that flows of information from the `then` branch only execute if the condition e is true at this execution point at least once.

The rules for `if` and `while` statements also track indirect flows of information. For example, rule (12) adds a flow from a variable x referenced by the condition e to a variable y defined by a statement within either the `then` or `else` branch. Such indirect flows of information are key to confidentiality as the following program demonstrates:

```
if(x){ y := 1 }{ y := 0 }
```

The `if` statement copies the value of `x` into `y` without using a direct flow of information from either assignment statement.

$$\frac{x \in \mathsf{ref}(e)}{x[v\texttt{:=}e]^{\mathsf{T}}v}(1) \qquad \frac{x \neq v}{x[v\texttt{:=}e]^{\mathsf{T}}x}(2) \qquad \frac{x \in \mathsf{ref}(e)}{x[\texttt{write(}e\texttt{,}d\texttt{)}]^{\mathsf{T}}d}(3) \qquad \frac{}{x[\texttt{write(}e\texttt{,}d\texttt{)}]^{\mathsf{T}}x}(4)$$

$$\frac{}{\mathsf{nt}[\texttt{write(}e\texttt{,}d\texttt{)}]^{\mathsf{T}}d}(5) \qquad \frac{}{d[\texttt{read(}v\texttt{,}d\texttt{)}]^{\mathsf{T}}v}(6) \qquad \frac{x \neq v}{x[\texttt{read(}v\texttt{,}d\texttt{)}]^{\mathsf{T}}x}(7) \qquad \frac{x[s_1]^{\hat{P}_1}z \qquad z[s_2]^{\hat{P}_2}y}{x[s_1\texttt{;}s_2]^{\hat{P}_1 \wedge \hat{P}_2}y}(8)$$

$$\frac{x[s_1]^{\hat{P}}y}{x[\texttt{if(}e\texttt{)}\{s_1\}\{s_2\}]^{\hat{P} \wedge e \mapsto \mathsf{T}}y}(9) \qquad \frac{x[s_2]^{\hat{P}}y}{x[\texttt{if(}e\texttt{)}\{s_1\}\{s_2\}]^{\hat{P} \wedge e \mapsto \mathsf{F}}y}(10) \qquad \frac{x[s_1]^{\hat{P}}y \qquad x[s_2]^{\hat{P}}y}{x[\texttt{if(}e\texttt{)}\{s_1\}\{s_2\}]^{\hat{P}}y}(11)$$

$$\frac{x \in \mathsf{ref}(e) \qquad y \in \mathsf{def}(s_i)}{x[\texttt{if(}e\texttt{)}\{s_1\}\{s_2\}]^{\mathsf{T}}y}(12) \qquad \frac{}{x[\texttt{while(}e\texttt{)}\{s_1\}]^{e \mapsto \mathsf{F}}x}(13) \qquad \frac{x[s_1]_{+}^{\hat{P}}y}{x[\texttt{while(}e\texttt{)}\{s_1\}]^{\hat{P} \wedge e \mapsto \mathsf{T}}y}(14)$$

$$\frac{x[s_1]_{+}^{\hat{P}}x}{x[\texttt{while(}e\texttt{)}\{s_1\}]^{\hat{P}}x}(15) \qquad \frac{x \in \mathsf{ref}(e) \qquad y \in \mathsf{def}(s_1)}{x[\texttt{while(}e\texttt{)}\{s_1\}]^{\mathsf{T}}y}(16) \qquad \frac{x \in \mathsf{ref}(e)}{x[\texttt{while(}e\texttt{)}\{s_1\}]^{\mathsf{T}}\mathsf{nt}}(17)$$

$$\frac{x[s]^{\hat{P}_1}z \qquad z[s]_{+}^{\hat{P}_2}y}{x[s]_{+}^{\hat{P}_1 \wedge \hat{P}_2}y}(18) \qquad \frac{x[s]^{\hat{P}}y}{x[s]_{+}^{\hat{P}}y}(19) \qquad \frac{x[s]^{\hat{P}_1}y \qquad x[s]^{\hat{P}_2}y}{x[s]^{\hat{P}_1 \vee \hat{P}_2}y}(20) \qquad \frac{x[s]^{\hat{P}_2}y \qquad \hat{P}_1 \Rightarrow \hat{P}_2}{x[s]^{\hat{P}_1}y}(21)$$

Table 1: Analysis Inference Rules

Intuitively, the rules for $x[s]^{\hat{P}}_{+}y$ unroll the loop in which s is found an unbounded number of times.

By rule (17), the pseudo-identifier `nt` depends on identifiers referenced by the condition of a `while` loop. It is provable given the rules of Table 1 that for all s and all $\hat{P}$, $\mathtt{nt}[s]^{\hat{P}}\mathtt{nt}$. Since nontermination is only noticeable to a user in the presence of a `write` statement, we have suppressed irrelevant rules involving `nt` flowing to other types of statements leaving just rule (5). In summary, a reference from the condition of a `while` statement will flow to the written buffer d of any `write` statement that follows the `while` statement by way of `nt`.

These inference rules admit spurious flows. That is, $x[s]^{\hat{P}}y$ does not imply that the value of x will definitely affect the value of y during an execution of s where $\hat{P}$ holds. However, if x does affect the value of y during an execution of s where $\hat{P}$ holds, then $x[s]^{\hat{P}}y$ will definitely hold.

The syntactic policy that we extract is the logically weakest $\hat{P}$ such that $\mathsf{H}[s]^{\hat{P}}\mathsf{L}$. That is, $\hat{P}$ is the disjunction of all $\hat{P}'$ such that $\mathsf{H}[s]^{\hat{P}'}\mathsf{L}$. Since the rules find every possible flow, $[\![\hat{P}]\!]^s$ will contain every input sequence that results in L gaining access to a input from H. That is, the absence of $\vec{i}$ from $[\![\hat{P}]\!]^s$ implies that H will surely not flow to L when the program s receives $\vec{i}$ as input (thereby preserving the confidentiality of H). However, while sound, $[\![\hat{P}]\!]^s$ is an over-approximation and might contain extra input sequences that do not actually result in such a flow.

4.4 Example

We now demonstrate these inference rules on our motivating example from Section 1. For each statement and pair of identifiers, we apply every applicable rule to find the least restrictive syntactic policy for the statement and identifiers. We build the policies of composite statements from the policies of their sub-statements making our analysis compositional. Our implementation proceeds in a similar manner. To save space we use $e_{\mathtt{doc}}$ as shorthand for `roles[login] == "doc"`.

Let us start with the last statement, `write(out, stdout)`. Here we apply the rules for `write` to learn that `out` flows to `stdout` from rule (3) and that for all x, x flows to x from rule (4):

$$\frac{\mathtt{out} \in \mathsf{ref}(\mathtt{out})}{\mathtt{out}[\mathtt{write(out,\ stdout)}]^{\mathsf{T}}\mathtt{stdout}}(3) \qquad \frac{}{x[\mathtt{write(out,\ stdout)}]^{\mathsf{T}}x}(4)$$

This makes sense since the `write` statement copies the value of `out` into the buffer `stdout` without overwriting any data. All of these flows are under the policy T, which means they may always be possible. (Also, `nt` flows to `stdout` by rule (5) meaning that nontermination will be noticed by the absence of this `write` statement getting to execute. Since nontermination does not play a roll in this example, we will henceforth ignore `nt`.)

Examining the assignment `out := append(bill,xray)`, we learn by rule (1) that `bill` and `xray` flow to `out` under T:

$$\frac{\mathtt{bill} \in \mathsf{ref}(\mathtt{append(bill,xray)})}{\mathtt{bill}[\mathtt{out\ :=\ append(bill,xray)}]^{\mathsf{T}}\mathtt{out}}(1) \qquad \frac{\mathtt{xray} \in \mathsf{ref}(\mathtt{append(bill,xray)})}{\mathtt{xray}[\mathtt{out\ :=\ append(bill,xray)}]^{\mathsf{T}}\mathtt{out}}(1)$$

From rule (2), we learn that for all x other than `out`, x flows to x under T:

$$\frac{x \neq \mathtt{out}}{x[\mathtt{out\ :=\ append(bill,xray)}]^{\mathsf{T}}x \quad \forall x \neq \mathtt{out}}(2)$$

We do not allow `out` to flow to `out` since the value that was stored in `out` before the assignment is overwritten with `append(bill,xray)` by the assignment. Accounting for the overwriting of data is key to making our analysis flow sensitive.

The other assignment `out := bill` proceeds in the same manner to have `bill` flowing to `out` under T and x flowing to x under T for all x other than `out`:

$$\frac{\mathtt{bill} \in \mathsf{ref}(\mathtt{bill})}{\mathtt{bill}[\mathtt{out\ :=\ bill}]^{\mathsf{T}}\mathtt{out}}\,(1) \qquad\qquad \frac{x \neq \mathtt{out}}{x[\mathtt{out\ :=\ bill}]^{\mathsf{T}}x \quad \forall x \neq \mathtt{out}}\,(2)$$

With policies for both the `then` and `else` branches done, we can now compose them to get the policies for the `if` statement. The rules (9) and (10) allow us to add the flows from the `then` and `else` branches as long as we keep track of which branch they come from. To keep track, we add the control constraint $e_{\mathtt{doc}}{\mapsto}\mathsf{T}$ to the policy of any flow coming from the `then` branch. Likewise, we add $e_{\mathtt{doc}}{\mapsto}\mathsf{F}$ for flows from the `else` branch. Rule (11) allows us to simplify and not add these additional control constraints if a flow with the same policy comes from both the `then` and `else` branches. With this simplification, we get the following flows: `bill` to `out` under the policy T, `xray` to `out` under $e_{\mathtt{doc}}{\mapsto}\mathsf{T}$, and x to x for all x other than `out` under T:

$$\frac{\mathtt{bill}[\mathtt{out\ :=\ append(bill,xray)}]^{\mathsf{T}}\mathtt{out} \qquad \mathtt{bill}[\mathtt{out\ :=\ bill}]^{\mathsf{T}}\mathtt{out}}{\mathtt{bill}[\mathtt{if}(e_{\mathtt{doc}})\{\mathtt{out\ :=\ append(bill,xray)}\}\{\mathtt{out\ :=\ bill}\}]^{\mathsf{T}}\mathtt{out}}\,(11)$$

$$\frac{\mathtt{xray}[\mathtt{out\ :=\ append(bill,xray)}]^{\mathsf{T}}\mathtt{out}}{\mathtt{xray}[\mathtt{if}(e_{\mathtt{doc}})\{\mathtt{out\ :=\ append(bill,xray)}\}\{\mathtt{out\ :=\ bill}\}]^{\mathsf{T}\wedge e_{\mathtt{doc}}\mapsto\mathsf{T}}\mathtt{out}}\,(9)$$

$$\frac{x[\mathtt{out\ :=\ append(bill,xray)}]^{\mathsf{T}}x \quad \forall x \neq \mathtt{out} \qquad x[\mathtt{out\ :=\ bill}]^{\mathsf{T}}x \quad \forall x \neq \mathtt{out}}{x[\mathtt{if}(e_{\mathtt{doc}})\{\mathtt{out\ :=\ append(bill,xray)}\}\{\mathtt{out\ :=\ bill}\}]^{\mathsf{T}}x \quad \forall x \neq \mathtt{out}}\,(11)$$

We must also add indirect flows of information for `if` statements using rule (12). Since `login` and `roles` are referenced by the condition of the `if` statement and `out` is assigned in the `if` statement, this adds two flows: `login` to `out` under T and `roles` to `out` under T:

$$\frac{\mathtt{login} \in \mathsf{ref}(e_{\mathtt{doc}}) \qquad \mathtt{out} \in \mathsf{def}(\mathtt{out\ :=\ bill})}{\mathtt{login}[\mathtt{if}(e_{\mathtt{doc}})\{\mathtt{out\ :=\ append(bill,xray)}\}\{\mathtt{out\ :=\ bill}\}]^{\mathsf{T}}\mathtt{out}}\,(12)$$

$$\frac{\mathtt{roles} \in \mathsf{ref}(e_{\mathtt{doc}}) \qquad \mathtt{out} \in \mathsf{def}(\mathtt{out\ :=\ append(bill,xray)})}{\mathtt{roles}[\mathtt{if}(e_{\mathtt{doc}})\{\mathtt{out\ :=\ append(bill,xray)}\}\{\mathtt{out\ :=\ bill}\}]^{\mathsf{T}}\mathtt{out}}\,(12)$$

Now we combine the policies extracted from the `if` and `write` statements using rule (8). Intuitively, the rule connects flows from the first statement (the `if` statement) to flows from the second statement (the `write` statement). For example, since `xray` flows to `out` under $e_{\mathtt{doc}}{\mapsto}\mathsf{T}$ in the `if` statement and `out` flows to `stdout` under T in the `write` statement, `xray` flows to `stdout` under $e_{\mathtt{doc}}{\mapsto}\mathsf{T}$ in their sequential composition.

$$\frac{\begin{array}{c}\mathtt{xray}[\mathtt{if}(e_{\mathtt{doc}})\{\mathtt{out\ :=\ append(bill,xray)}\}\{\mathtt{out\ :=\ bill}\}]^{e_{\mathtt{doc}}\mapsto\mathsf{T}}\mathtt{out} \\ \mathtt{out}[\mathtt{write(out,\ stdout)}]^{\mathsf{T}}\mathtt{stdout}\end{array}}{\mathtt{xray}[\mathtt{if}(e_{\mathtt{doc}})\{\mathtt{out{:=}append(bill,xray)}\}\{\mathtt{out{:=}bill}\}\mathtt{;\ write(out,stdout)}]^{e_{\mathtt{doc}}\mapsto\mathsf{T}\wedge\mathsf{T}}\mathtt{stdout}}\,(8)$$

The same reasoning applies to the other flows, which we will not list.

Continuing this compositional reasoning and treating read statements in a manner similar to assignment, we produce a syntactic policy for each pair of identifiers for the whole program. The policy for flows from sensitive input `xray.jpg` to the untrusted user `stdout` is $e_{\mathtt{doc}} \mapsto \mathsf{T}$ as expected.

5 Implementation

We have implemented a policy extraction tool based on our inference rules. The algorithm recursively operates on the abstract syntax tree of a program in a depth-first fashion. It computes the policy of a node by composing together the policies of the node's children. This requires $O(n)$ recursive calls where n is the number of abstract syntax tree nodes in the program.

The algorithm works on programs written in a subset of C, which includes pointers (but no aliasing), non-recursive functions, while loops, and file operations. We do not model flows from runtime errors, unstructured control flow, or pointers accessing memory not allocated for them.

The algorithm represents syntactic policies as binary decision diagrams (BDDs) [Bry86]. A BDD is required for each pair of identifiers since policies between two identifiers other than L and H for a sub-statement may affect the policy of L and H for the program. Although this requires a number of BDDs that is quadratic in the number of identifiers, they share isomorphic sub-trees. BDDs provide a canonical form for policies making verifying properties about policies efficient [FKMT05].

The algorithm computes the non-reflexive transitive closure $x[s]^{C}_{+}y$ for `while` statements using the Floyd-Warshall algorithm (see, *e.g.*, [CLRS01]). This requires $O(v^3)$ steps where v is the number of variables in the program. The policy of a function is computed once and reused for each application of the function.

The algorithm distinguishes between pointers to arrays and the memory locations within an array. If information flowed to a pointer `y`, then an assignment to `y` will terminate that flow since the value of `y` is overwritten (as in rule (2) of our inference system). Since `y` would also now point to a different memory location, flows to the memory location to which `y` used to point would also be terminated. However, a write into this array, such as `y[z] := 0`, will not terminate any flows since the value of `y` is unchanged and other locations in the array may continue to hold information from a flow to the memory locations to which `y` points. Furthermore, we model that the value of `z` flows to these locations. This flow arises because the `z`th entry of `y` changes to `0` while the other entries remain the same. If every entry starts with a value other than `0`, the location of this change reflects the value of `z`.

To parse the C code, we use CIL [NMRW02], which unfortunately adds many temporary variables that slows down our algorithm. The main part of the algorithm is coded as a PLT Scheme program running in DrScheme [FCF$^+$02]. This code uses a foreign function interface [BO04] to the CUDD BDD package [Som], a C library, to construct and manage the BDDs.

Since the conversion from a syntactic policy $\hat{P}$ to the policy $[\![P]\!]^s$ may be done with standard weakest precondition calculations, we do not provide an implementation for this operation.

First, we experimented with our implementation on C programs that exercise the subset of C our implementation accepts. The most interesting one of these is based on the motivating doctor example. The program includes three helper functions, error code checking, command-line arguments, file operations, and numerous loops. After parsing by CIL, it has 93 atomic and composite statements using 23 variables, four files, and two output streams. The analysis extracted the correct policy in 2.68 seconds. It used 3MB of RAM on top of a 250MB baseline for DrScheme

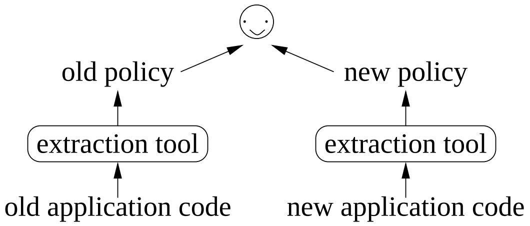

Figure 3: Change-impact analysis between two versions of one application.

and the operating system.

To test the scalability of our analysis, we ran our implementation on 757 statements of the Sparse C parser (`http://www.kernel.org/pub/software/devel/sparse/`). The analysis took 510 seconds and used 95MB of RAM. We also tested our analysis on a 3137 statement part of the Privoxy privacy-enhancing web proxy (`http://www.privoxy.org/`). The analysis took 78 seconds and 60MB of RAM. The policies extracted from these programs provide an approximation of the real policies that the programs enforce; however, these programs use unmodeled features of C allowing for the possibility of unsoundness.

The analysis ran on a computer with two 1.2GHz AMD Athlon processors and 757MB of RAM. Our implementation and the analyzed programs may be downloaded from

`http://www.cs.cmu.edu/~mtschant/policy-extraction/`

6 Uses for Our Tool

Many uses exist for the extracted policy: A developer could examine the conditions in their own right to understand the program. A system administer could use them to compute the inputs that result in the program granting the untrusted user access to the sensitive inputs. An auditor could use the extracted policy to verify program correctness.

A user can view the extracted policy before deciding whether to use the application. For example, third-party applets that connect to a social networking website like Facebook typically receive access to far more sensitive data than they need [FE]. Often the access that is appropriate for the applet depends on conditional information. For example, it becomes appropriate for an applet to access the user's address only after the user requests the applet to perform a location specific task. A user could use our analysis to determine if an applet has reasonable confidentiality properties before installing it. These applets are attractive candidates for applying our analysis since they use a standard API to access sensitive information and are typically of moderate size and complexity.

A code maintainer could refactor a program to be configured by an externally specified policy and use the extracted policy as the configuration. Ganapathy et al. have developed tools to retrofit legacy code for this purpose [GJJ06].

The code maintainer could also perform change-impact analysis (Fig. 3): given application code before and after some set of edits, he can compare the policies extracted from both versions of the application to ensure that the program edits have not introduced unintended consequences. We have implemented a change-impact analysis algorithm. The comparison process relies on a BDD differencing algorithm previously presented for comparing policies [FKMT05].

Whereas confidentiality requires that protected data does not become known to untrusted users, *integrity* requires that protected data does not become tainted or corrupted by untrusted users. By reversing the roles of the sensitive and untrusted information, integrity becomes confidentiality. Thus, our tool can also produce conditional integrity policies.

7 Related Work

We cover related work by discussing work done on problems related to policy extraction. Interestingly, the most closely related work is on the very different problem of data model extraction.

Extracting Data Models. Data model extraction attempts to infer from untyped code (such as COBOL code), the data model that the programmer had in mind. Typically this model is presented as a class hierarchy explaining how the analyzed program uses buffers to store multiple pieces of data.

Although this problem is very different from ours, the information required to solve it is similar to the information we extract. Komondoor and Ramalingam extract from source code data flows and the conditions that enable them [KR07]. Their analysis differs from ours in that it does not consider indirect flows of information from conditional statements. While such flows are irrelevant to their goals, they matter greatly to ours. Also, their analysis is not compositional like ours and uses a very different algorithmic approach based on lattices.

Enforcing a Given Policy. The problem most related in motivation to ours is the problem of ensuring that a program will obey a specified policy. While we build on the theory underlying this work, our approach differs from those taken in this area.

Conditional information flow analysis is a method of preventing undesired information flows at runtime. These methods use tags present at runtime to track the flow of information. These tags are managed using either hardware (*e.g.*, [SLZD04]), code instrumentation (*e.g.*, [LMLW08]), or virtual machines (*e.g.*, [HCF05]). If these methods detect an undesired flow at runtime, the execution must be aborted. We instead offer a static analysis.

Many type systems for information flow analysis exist. (For a survey, see [SM03].) Most of these use a *batch-job* model of systems: systems take a set of inputs before execution and produce a set of outputs upon termination. We use an *interactive* model of systems where the program may interact with the user throughout the execution. O'Neill et al. provide the only type system of which we know for interactive programs [OCC06]. However, their work assumes a unconditional confidentiality policy.

Information-flow type systems for declassification use conditional confidentiality policies, or *declassification policies*. Of the many type systems for declassification (see [SS05] for an overview), the work of Chong and Myers most resembles ours [CM04]. They use a type system to ensure that sensitive information is only released to a untrusted user (is declassified) if some condition annotating the code holds. Rather than ensuring that conditions annotating the code hold before declassification, our analysis finds these conditions in unannotated code.

Model checking can verify that a given policy is obeyed. This requires composing a program with itself since noninterference is a *2-safety property* instead of a standard safety property [BDR04]. However, this method requires the intended policy as input whereas our analysis produces a policy.

Ruling Out Infeasible Flows: Path Conditions. Program dependence graphs (PDGs) represent how information flows from statement to statement in a program [FOW84, FOW87]. However, some of these flows might require statements along an infeasible path to execute. Path conditions provide a sound approximation of which flows in a PDG are actually feasible [Sne96].

Using PDGs with path conditions provides a sound way to determine if unconditional noninterference holds [SRK06]. However, they do not directly extend to conditional noninterference. The problem is that PDGs do not show the passive information flows that happen when statements are *not* executed. This does not affect the soundness of PDGs or path conditions for unconditional noninterference because every missing passive flow is paired with a present active flow from the assignment being executed. In the conditional case, however, we need both flows of information to maintain soundness. Furthermore, our approach is compositional unlike PDGs.

Extracting Business Logic. Just as a confidentiality policy may become buried within the code of a large program, the operating procedures of a business may also become hidden within large applications. Thus, others have created tools to extract these business rules from source code [HTB^{+}96, Sne01]. These tools use program slicing to track information, but they do not provide the conditions that enable them [Tip95].

8 Summary and Future Work

After presenting a formalization of conditional noninterference, we presented a sound method to extract from application source code an approximation of the most restrictive policy the program obeys. This is the first policy extraction algorithm proposed and also the first conditional information flow analysis that finds both direct and indirect flows of information. Moreover, our analysis is flow sensitive and composition while handling interactive I/O.

Possible future work includes handling language features like exceptions, aliasing, and recursion. Additional analyses for partitioning arrays into separate confidentiality levels would make our analysis more accurate.

We would also like to present policies to the user in an interactive manner with a query engine to verify properties of the policy. Lastly, we would like to explore further uses for the extracted policies such as refactoring.

Acknowledgments. We thank Jonathan Aldrich for scrutinizing our inference rules. We also thank him, Karl Crary, and Frank Pfenning for helpful comments.

References

[BDR04] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSFW '04: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, page 100, Washington, DC, USA, 2004. IEEE Computer Society.

[BO04] Eli Barzilay and Dmitry Orlovsky. Foreign interface for PLT Scheme. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the Fifth ACM SIGPLAN Workshop on Scheme and Functional Programming*, pages 63–74, Snowbird, Utah, September 22,

2004. Technical report TR600, Department of Computer Science, Indiana University. `http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR600`.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 25.2 The Floyd-Warshall Algorithm, pages 629–635. The MIT Press, Cambridge, Massachusetts, second edition, 2001.

[CM04] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 198–209, New York, NY, USA, 2004. ACM.

[FCF$^+$02] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, 2002.

[FE] Adrienne Felt and David Evans. Privacy protection for social networking APIs. `http://www.cs.virginia.edu/felt/privacy/`.

[FKMT05] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 196–205, New York, NY, USA, 2005. ACM Press.

[FOW84] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 125–132, London, UK, 1984. Springer-Verlag.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[GJJ06] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Retrofitting legacy code for authorization policy enforcement. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 214–229, Los Alamitos, CA, USA, May 2006. IEEE Computer Society.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, page 11. IEEE, 1982.

[HCF05] Vivek Haldar, Deepak Chandra, and Michael Franz. Practical, dynamic information-flow for virtual machines. In *2nd International Workshop on Programming Language Interference and Dependence*, September 2005.

[HTB$^+$96] Hai Huang, W. T. Tsai, S. Bhattacharya, X. P. Chen, Y. Wang, and J. Sun. Business rule extraction from legacy code. In *COMPSAC '96 - 20th Computer Software and Applications Conference*, pages 162–167, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[KR07] Raghavan Komondoor and G. Ramalingam. Recovering data models via guarded dependences. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 110–119, Washington, DC, USA, 2007. IEEE Computer Society.

[LMLW08] Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing web applications with static and dynamic information flow tracking. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 3–12, New York, NY, USA, 2008. ACM.

[McC87] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symposium on Security and Privacy*, pages 161–166, 1987.

[McC88] Daryl McCullough. Noninterference and the composability of security properties. In *IEEE Symposium on Security and Privacy*, pages 177–186, Los Alamitos, CA, USA, 1988. IEEE Computer Society.

[McC90] D. McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6):563–568, 1990.

[McL94] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *SP '94: Proceedings of the 1994 IEEE Symposium on Security and Privacy*, page 79, Washington, DC, USA, 1994. IEEE Computer Society.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[OCC06] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 190–201, Washington, DC, USA, 2006. IEEE Computer Society.

[SLZD04] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, New York, NY, USA, 2004. ACM.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[Sne96] Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 332–348, London, UK, 1996. Springer-Verlag.

[Sne01] Harry M. Sneed. Extracting business logic from existing cobol programs as a basis for redevelopment. In *Proceedings. 9th International Workshop on Program Comprehension*, pages 167–175, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[Som] Fabio Somenzi. CUDD: The CU decision diagram package. `http://vlsi.colorado.edu/~fabio/CUDD/`.

[SRK06] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

[SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *CSFW '05: Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, pages 255–269, Washington, DC, USA, 2005. IEEE Computer Society.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[TW07] Michael Carl Tschantz and Jeannette M. Wing. Confidentiality policies and their extraction from programs. Technical Report CMU-CS-07-108, School of Computer Science, Carnegie Mellon University, February 2007.

[WJ90] J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 144–161, Los Alamitos, CA, USA, 1990. IEEE Computer Society.

A WhileIO Semantics

A WhileIO program consumes inputs and produces outputs in an interactive manner. We call inputs and outputs collectively *actions*. An action represents I/O as an ordered triple: the first component is i if the action is an input and o if it is an output, the second component is the domain of the action, and the third component is the contents of the action, a number from N. For example, $\langle \mathsf{i}, \mathsf{H}, 5\rangle$ could be an input from the domain H. We add to A a distinguished action τ, the internal transition, that no domain sees as output or creates as input. The set of actions A is $(\{\mathsf{i}, \mathsf{o}\} \times D \times N) \cup \{\tau\}$.

The program moves though a series of states $\langle \Gamma, s\rangle$ where s represents the code that still must be executed and Γ represents the current contents of memory. The judgment $\langle \Gamma, s\rangle \overset{a}{\hookrightarrow} \langle \Gamma', s'\rangle$ means that the statement s goes to s' while performing the action a and changing the memory from Γ to Γ'. We call such a step a *reduction*.

Table 2 gives the small-step semantics of WhileIO. We write unit as $\diamond$. We model the memory Γ as a store, that is, a function from variables to numbers. Let $\Gamma[v \mapsto n]$ be the store such that v is mapped to n and all v' other than v is mapped to $\Gamma(v')$. We extend stores to assign a number to expressions in the usual way. For example, let $\Gamma(e_1\texttt{+}e_2)$ be $\Gamma(e_1) + \Gamma(e_2)$ and $\Gamma(n) = n$ for numbers n.

We represent with $[\![s]\!](\vec{i})$ the I/O actions found in order in the trace generated by s given the input sequence $\vec{i}$. To formally define $[\![s]\!](\vec{i})$, we introduce $\langle \Gamma, s\rangle \xrightarrow{\vec{o}} \langle \Gamma', s'\rangle$, which means that the state $\langle \Gamma, s\rangle$ transitions to $\langle \Gamma', s'\rangle$ while producing the outputs $\vec{o}$ and not consuming any inputs. Formally, $\langle \Gamma, s\rangle \xrightarrow{o:\vec{o}} \langle \Gamma', s'\rangle$ if either $\langle \Gamma, s\rangle \overset{\tau}{\hookrightarrow} \langle \Gamma'', s''\rangle$ and $\langle \Gamma'', s''\rangle \xrightarrow{o:\vec{o}} \langle \Gamma', s'\rangle$, or $\langle \Gamma, s\rangle \overset{o}{\hookrightarrow} \langle \Gamma'', s''\rangle$ and $\langle \Gamma'', s''\rangle \xrightarrow{\vec{o}} \langle \Gamma', s'\rangle$. For the base case, $\langle \Gamma, s\rangle \xrightarrow{[]} \langle \Gamma', s'\rangle$

$$\frac{}{\langle \Gamma, x{:=}e\rangle \overset{\tau}{\hookrightarrow} \langle \Gamma[x \mapsto \Gamma(e)], \diamond\rangle} \qquad \frac{}{\langle \Gamma, \texttt{read(}x\texttt{, }d\texttt{)}\rangle \overset{\langle \mathsf{i}, d, n\rangle}{\hookrightarrow} \langle \Gamma[x \mapsto n], \diamond\rangle}$$

$$\frac{n = \Gamma(e)}{\langle \Gamma, \texttt{write(}e\texttt{, }d\texttt{)}\rangle \overset{\langle \mathsf{o}, d, n\rangle}{\hookrightarrow} \langle \Gamma, \diamond\rangle} \qquad \frac{\langle \Gamma, s_1\rangle \overset{a}{\hookrightarrow} \langle \Gamma', s_1'\rangle}{\langle \Gamma, s_1\texttt{;}s_2\rangle \overset{a}{\hookrightarrow} \langle \Gamma', s_1'\texttt{;}s_2\rangle} \qquad \frac{}{\langle \Gamma, \diamond\texttt{;}s_2\rangle \overset{\tau}{\hookrightarrow} \langle \Gamma, s_2\rangle}$$

$$\frac{\Gamma(e) \neq 0}{\langle \Gamma, \texttt{if(}e\texttt{)\{}s_1\texttt{\}\{}s_2\texttt{\}}\rangle \overset{\tau}{\hookrightarrow} \langle \Gamma, s_1\rangle} \qquad \frac{\Gamma(e) = 0}{\langle \Gamma, \texttt{if(}e\texttt{)\{}s_1\texttt{\}\{}s_2\texttt{\}}\rangle \overset{\tau}{\hookrightarrow} \langle \Gamma, s_2\rangle} \qquad \frac{\Gamma(e) = 0}{\langle \Gamma, \texttt{while(}e\texttt{)\{}s_1\texttt{\}}\rangle \overset{\tau}{\hookrightarrow} \langle \Gamma, \diamond\rangle}$$

$$\frac{\Gamma(e) \neq 0}{\langle \Gamma, \texttt{while(}e\texttt{)\{}s_1\texttt{\}}\rangle \overset{\tau}{\hookrightarrow} \langle \Gamma, s_1\texttt{; while(}e\texttt{)\{}s_1\texttt{\}}\rangle}$$

Table 2: Small-Step Semantics of WhileIO

if $\langle \Gamma, s\rangle \overset{\tau}{\hookrightarrow}^{*} \langle \Gamma', s'\rangle$ and there exists no $\langle \Gamma'', s''\rangle$ such $\langle \Gamma', s'\rangle \overset{\tau}{\hookrightarrow} \langle \Gamma'', s''\rangle$ where $\overset{\tau}{\hookrightarrow}^{*}$ represents zero or more τ transitions.

We define $\llbracket s \rrbracket(\vec{i})$ to be $\llbracket s \rrbracket(\Gamma_0, \vec{i})$ where Γ_0 is the store that maps every variable to zero and $\llbracket s \rrbracket(\Gamma, \vec{i})$ is defined as follows: $\llbracket s \rrbracket(\Gamma, i{:}\vec{i}) = \vec{o}_1{:}i{:}\vec{o}_2\llbracket s' \rrbracket(\Gamma', \vec{i})$ where

$$\langle \Gamma, s\rangle \xrightarrow{\vec{o}_1} \langle \Gamma'', s''\rangle \overset{i}{\hookrightarrow} \langle \Gamma''', s'''\rangle \xrightarrow{\vec{o}_2} \langle \Gamma', s'\rangle$$

and $\llbracket s \rrbracket(\Gamma, []) = \vec{o}$ where $\langle \Gamma, s\rangle \xrightarrow{\vec{o}} \langle \Gamma', s'\rangle$.