

Extracting Conditional Confidentiality Policies

Michael Carl Tschantz
Computer Science Department
Carnegie Mellon University
mtschant@cs.cmu.edu

Jeannette M. Wing
Computer Science Department
Carnegie Mellon University
wing@cs.cmu.edu

Abstract

Sensitive information, such as medical records, should be kept reasonably confidential. How do we determine what confidentiality policy a given program enforces? To address such questions, 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 accommodates 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.

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: 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 frag-

```

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

ment of code shown in Figure 1, he would rather have a tool to automate the process as much as possible. 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 [7]. 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 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. 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 [7]. 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 [14]. 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 provide our formulation of conditional confidentiality in Section 3. Using this formulation, we formally describe our conditional information flow analysis for a small programming language in Section 4. In Section 5, we discuss our implementation for a subset of C and our experiments with it on C programs. We discuss the application of our algorithm to computing the difference between programs in Section 6 and related work in Section 7. We end with directions for future work.

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 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 [9].

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 [7]. 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 allowed to know if the system has received any sensitive inputs. For example, consider the following program:

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

The first line reads in the contents of a secret file as input. The second line simply prints “hi” to the untrusted user. 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. For example:

- 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 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. Both the student and a professor recommending him must enter information into the application database. Once the 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 [10], restrictiveness [10], and separability [11]) 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 [23, 13], we are the first, to our knowledge, to distinguish them from incident-sensitive requirements 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 [21].

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 $\llbracket s \rrbracket(\vec{i})$. The behavior of s is an interleaving of the input sequence \vec{i} with high- and low-level outputs from O . Thus, $\llbracket s \rrbracket$ 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,

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

$$\lfloor \emptyset \rfloor = \emptyset$$

where $\text{level}(a)$ is the level of a , \emptyset 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^* ,

$$\lfloor \vec{i}_1 \rfloor = \lfloor \vec{i}_2 \rfloor \text{ implies } \llbracket \llbracket s \rrbracket(\vec{i}_1) \rrbracket = \llbracket \llbracket s \rrbracket(\vec{i}_2) \rrbracket$$

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 $\vec{i}_1 = [a^H]$ and $\vec{i}_2 = \emptyset$ where a^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 hi^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 $\wr \cdot \wr$. 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 $*^H$ that only carries the information that a high-level input was there. For example, both $\wr [a^H, b^L] \wr$ and $\wr [c^H, b^L] \wr$ blur to $[*^H, b^L]$ while $\wr [b^L] \wr$ blurs to $[b^L]$. We will define incident-insensitive noninterference to require that a program produce the same low-level outputs on $[a^H, b^L]$ and $[c^H, b^L]$ while being free to produce different outputs on $[b^L]$. Formally,

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

$$\wr \emptyset \wr = \emptyset$$

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

$$\wr \vec{i}_1 \wr = \wr \vec{i}_2 \wr \text{ implies } \llbracket \llbracket s \rrbracket(\vec{i}_1) \rrbracket = \llbracket \llbracket s \rrbracket(\vec{i}_2) \rrbracket$$

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. By induction, for all \vec{i}_1 and \vec{i}_2 , $\wr \vec{i}_1 \wr = \wr \vec{i}_2 \wr$ implies $\lfloor \vec{i}_1 \rfloor = \lfloor \vec{i}_2 \rfloor$. 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 [21] for a more formal proof.) \square

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. Since the program allows the contents of `xray.jpg` to affect the value printed to `stdout` whenever a doctor's login is provided, it does not obey either form of noninterference.

Despite not obeying noninterference, the program does not violate confidentiality by only allowing doctors to view the X-ray. To capture such situations, Goguen and Meseguer provided a conditional version of incident-sensitive noninterference [7]. 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` 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. 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.

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

$$\ulcorner \vec{i}_1 \urcorner = \ulcorner \vec{i}_2 \urcorner \text{ implies } \llbracket [s] \urcorner(\vec{i}_1) \rrbracket = \llbracket [s] \urcorner(\vec{i}_2) \rrbracket$$

When \vec{i}_1 is not in P , the requirement of unconditional incident-insensitive noninterference must hold. However, when \vec{i}_1 is in P , then the low-level user is allowed to learn what inputs the high-level user has produced.

If nontermination cannot be observed, then $\llbracket [s] \urcorner(\vec{i}_1) \rrbracket$ may deviate from equality and merely be a prefix of $\llbracket [s] \urcorner(\vec{i}_2) \rrbracket$ without posing a risk to confidentiality as we explain elsewhere [22].

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 may 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 through the control flow graph may yield information flows. If \hat{P} does not include a description of a path, then flows cannot occur during executions taking that path and the program will maintain the confidentiality of high-level information.

\hat{P} resembles approximately the weakest precondition for the postcondition *an information flow exists*. However, our analysis differs from analyses for standard weakest preconditions since the postcondition *an information flow exists* cannot be directly expressed in standard Hoare logic since it requires comparing the behavior of a program across two executions [1].

In our motivating example from Section 1, only executions that take the `then` branch result in information flows from the high-level `xray.jpg` to the low-level `stdout`. Thus, the extracted syntactic policy \hat{P} for that program states that the key condition `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, 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 with side effects and expressions e without side effects:

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

where v ranges over a set of variable names, n over a set of numbers, and ℓ over a set of confidentiality levels. A program is just a single statement.

A program goes through 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 $\llbracket s \rrbracket(\vec{i})$ the I/O actions found in order in the trace generated by s given the input sequence \vec{i} . The appendix of our related technical report provides a more detailed semantics for WhileIO [22].

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. Using \top for true, the control constraint c is represented as $e \mapsto \top$. (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 $e \mapsto \text{F}$ with F standing for false. Likewise for a `while` loop with the controlling expression e , we use $e \mapsto \top$ for the case where the body is entered and $e \mapsto \text{F}$ 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 s_a :

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

the syntactic policy

$$\hat{P}_a = x>0 \mapsto \text{F} \vee (x>0 \mapsto \top \wedge y<3 \mapsto \top \wedge y<3 \mapsto \text{F})$$

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 branches makes sense since the `if` statement lies within a loop body. ($e \mapsto \text{F}$ is *not* the negation of $e \mapsto \top$.)

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 $\llbracket \hat{P} \rrbracket^s$ where $\llbracket \hat{P} \rrbracket^s = \{ \vec{i} \in I \mid \langle\langle s \rangle\rangle(\vec{i}) \Rightarrow \hat{P} \}$. A program obeys a syntactic policy \hat{P} if it obeys the policy $\llbracket \hat{P} \rrbracket^s$ under Definition 3. For example, if s_a obeys the syntactic policy \hat{P}_a , then information will not flow from H to L unless the input is such that either the loop body is never entered, or the loop body is entered and both the branches of the `if` statement are taken.

4.3. Algorithm Specification

We now present an inference system that produces a syntactic policy \hat{P} such that $\llbracket \hat{P} \rrbracket^s$ is a sound over-approximation of the most restrictive policy P that a given program s obeys as a conditional incident-insensitive non-interference policy.

We call variables and confidentiality levels 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 non-termination. We assume that `nt` shows up nowhere in the analyzed program.

Let $\text{ref}(e)$ be the set of identifiers referenced by an expression e : $\text{ref}(n) = \emptyset$, $\text{ref}(v) = \{v\}$, $\text{ref}(e_1 + e_2) = \text{ref}(e_1) \cup \text{ref}(e_2)$, and so forth. Let $\text{def}(s)$ be the set of identifiers defined by a statement s :

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

Both v and ℓ are in $\text{def}(\text{read}(v, \ell))$ since the first element of the input buffer used by ℓ is removed and assigned to v , affecting both ℓ and v . We add `nt` to $\text{def}(\text{while}(e) \{s_1\})$ 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, $x[s]^{\hat{P}}\text{nt}$ means that the value of x before s may affect whether s will terminate. On the other hand, $\text{nt}[s]^{\hat{P}}y$ means that non-termination before s is called (*i.e.*, s not getting a chance to execute) may result in y having a different value than if

$$\begin{array}{c}
\frac{x \in \text{ref}(e)}{x[v := e]^T v} (1) \quad \frac{x \neq v}{x[v := e]^T x} (2) \quad \frac{x \in \text{ref}(e)}{x[\text{write}(e, \ell)]^T \ell} (3) \quad \frac{}{x[\text{write}(e, \ell)]^T x} (4) \quad \frac{}{\text{nt}[\text{write}(e, \ell)]^T \ell} (5) \\
\\
\frac{}{d[\text{read}(v, \ell)]^T v} (6) \quad \frac{x \neq v}{x[\text{read}(v, \ell)]^T x} (7) \quad \frac{x[s_1]^{\hat{P}_1} z \quad z[s_2]^{\hat{P}_2} y}{x[s_1 ; s_2]^{\hat{P}_1 \wedge \hat{P}_2} y} (8) \quad \frac{x[s_1]^{\hat{P}} y}{x[\text{if}(e) \{s_1\} \{s_2\}]^{\hat{P} \wedge e \rightarrow T} y} (9) \\
\\
\frac{x[s_2]^{\hat{P}} y}{x[\text{if}(e) \{s_1\} \{s_2\}]^{\hat{P} \wedge e \rightarrow F} y} (10) \quad \frac{x[s_1]^{\hat{P}} y \quad x[s_2]^{\hat{P}} y}{x[\text{if}(e) \{s_1\} \{s_2\}]^{\hat{P}} y} (11) \quad \frac{x \in \text{ref}(e) \quad y \in \text{def}(s_i)}{x[\text{if}(e) \{s_1\} \{s_2\}]^T y} (12) \\
\\
\frac{}{x[\text{while}(e) \{s_1\}]^{e \rightarrow F} x} (13) \quad \frac{x[s_1]^{\hat{P}} y}{x[\text{while}(e) \{s_1\}]^{\hat{P} \wedge e \rightarrow T} y} (14) \quad \frac{x[s_1]^{\hat{P}} x}{x[\text{while}(e) \{s_1\}]^{\hat{P}} x} (15) \\
\\
\frac{x \in \text{ref}(e) \quad y \in \text{def}(s_1)}{x[\text{while}(e) \{s_1\}]^T y} (16) \quad \frac{x \in \text{ref}(e)}{x[\text{while}(e) \{s_1\}]^T \text{nt}} (17) \quad \frac{x[s]^{\hat{P}_1} z \quad z[s]^{\hat{P}_2} y}{x[s]^{\hat{P}_1 \wedge \hat{P}_2} y} (18) \quad \frac{x[s]^{\hat{P}} y}{x[s]^{\hat{P}}_+ y} (19) \\
\\
\frac{x[s]^{\hat{P}_1} y \quad x[s]^{\hat{P}_2} y}{x[s]^{\hat{P}_1 \vee \hat{P}_2} y} (20) \quad \frac{x[s]^{\hat{P}_2} y \quad \hat{P}_1 \Rightarrow \hat{P}_2}{x[s]^{\hat{P}_1} y} (21)
\end{array}$$

Table 1. Analysis Inference Rules

s did get to execute. In Table 1, we provide the inference rules for $x[s]^{\hat{P}}y$.

The rules for `if` and `while` statements constrain the extracted policy by adding control constraints. For example, rule (9) adds the control constraint $e \rightarrow T$ 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.

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

A reference from the condition of a `while` statement will by rule (17) flow to the pseudo-identifier `nt`. This reference is carried by `nt` to the written buffer d of any `write` statement that follows the `while` statement by rule (5).

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 $H[s]^{\hat{P}}L$. That is, \hat{P} is the disjunction of all \hat{P}' such that $H[s]^{\hat{P}'}L$. Since the rules find every possible flow, $[[\hat{P}]]^s$ will contain every input sequence that results in L gaining access to an input from H under the conditional incident-insensitive noninterference formalization of access. The absence of \vec{i} from $[[\hat{P}]]^s$ implies that inputs from H will surely not flow to L when the program s receives \vec{i} as input (thereby preserving the confidentiality of H).

4.4. Example

We now demonstrate these inference rules on our motivating example from Section 1. 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_{doc} as shorthand for `roles[login] == "doc"`.

We start with `write(out, stdout)`, the last statement. We apply rule (3) to learn that `out` flows to `stdout` and rule (4) to learn that for all x , x flows to x :

$$\frac{\text{out} \in \text{ref}(\text{out})}{\text{out}[\text{write}(\text{out}, \text{stdout})]^T \text{stdout}} (3)$$

$$\frac{}{x[\text{write}(\text{out}, \text{stdout})]^T x} (4)$$

Indeed, the `write` statement copies the value of `out` into the buffer `stdout` without overwriting any data. All of these flows are under the true policy T , which means they may always be possible. Also, `nt` flows to `stdout` by rule (5) meaning that nontermination may be noticed by `stdout` if this `write` statement fails to execute:

$$\frac{}{\text{nt}[\text{write}(\text{out}, \text{stdout})]^T \text{stdout}} \quad (5)$$

Henceforth, we will only discuss those rules that contribute to proving that `xray.jpg` may flow to `stdout` under the condition $e_{\text{doc}} \mapsto T$.

Now consider the conditional statement. The assignment in the `then` branch introduces a flow from `xray` to `out` by rule (1). This flow implies that `xray` may flow to `out` when $e_{\text{doc}} \mapsto T$ by rule (9) as shown in Figure 2.

We use rule (8) to link the flow from `xray` to `out` arising from the `if` statement with the flow from `out` to `stdout` arising from the `write` statement. This produces a flow from `xray` to `stdout` under the condition $e_{\text{doc}} \mapsto T$.

Continuing this compositional reasoning and treating read statements in a manner similar to assignment, we produce the syntactic policy $e_{\text{doc}} \mapsto T$ governing the flow of information from `xray.jpg` to `stdout` as expected.

5. Implementation

We implemented a policy extraction tool based on our inference rules. The algorithm works on programs written in a subset of C, which includes pointers (but no aliasing), non-recursive functions, loops, and file operations. We do not model flows from runtime errors, unstructured control flow, or pointers accessing arbitrary memory locations.

Real programs are not neatly partitioned into two confidentiality levels, L and H. Thus, we extract for each input source (e.g., `stdin`) and output sink (e.g., `stdout`), the policy that governs flows from the source to the sink.

The algorithm recursively operates on the abstract syntax tree of a program in a depth-first fashion. It represents syntactic policies as binary decision diagrams (BDDs) [2] using the CUDD package [19]. A BDD is required for each pair of identifiers since policies between any two identifiers for a sub-statement may affect the policy of a source and sink 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 policy analysis efficient [6].

The algorithm computes the non-reflexive transitive closure $x[s]_+^C y$ for `while` statements using the Floyd-Warshall algorithm. This requires $O(v^3)$ steps where v is the number of variables in the program.

The implementation computes the policy of a function only once. It finds the flows from the function’s formal arguments to the return value. At an application site, the flows

to the function’s actual arguments are composed with the stored flows from the formal arguments to the return value in a manner similar to sequential composition. The algorithm uses the resulting flows for any identifier that depends on the return value.

The algorithm distinguishes between pointers to arrays and the memory locations within an array. If information flows to a pointer y , then an assignment to y later in the program 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 an 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 .

First, we experimented with our implementation on C programs that exercise the subset of C our implementation accepts. The most interesting one is based on the motivating doctor example. The program includes three helper functions, command-line arguments, file operations, and loops. After parsing by CIL [12], 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 the runtime (DrScheme [5]) and the operating system.

To test the scalability of our analysis, we ran our implementation on 757 statements of the Sparse C parser (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 web proxy (www.privoxy.org/). The analysis took 78 seconds and 60MB of RAM. These programs use unmodeled features of C allowing for 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. Change-Impact Analysis

The policies extracted by our tool are often too large to examine by hand. However, change-impact analysis is still possible. Given application code before and after some set of edits, one can compare the policies extracted from both versions of the application to ensure that the program edits

$$\frac{\frac{\text{xray} \in \text{ref}(\text{append}(\text{bill}, \text{xray}))}{\text{xray}[\text{out} := \text{append}(\text{bill}, \text{xray})]^{\top} \text{out}}{(1)}}{\text{xray}[\text{if}(e_{\text{doc}})\{\text{out} := \text{append}(\text{bill}, \text{xray})\}\{\text{out} := \text{bill}\}]^{\top \wedge e_{\text{doc}} \mapsto \top} \text{out}}(9)$$

Figure 2. `xray` may flow to `out` when $e_{\text{doc}} \mapsto \top$

have not introduced any unintended consequences.

We have implemented a change-impact analysis algorithm. The comparison process relies on a BDD differencing algorithm previously presented for comparing policies [6]. The time it takes to compare two policies is negligible relative to the time it takes to extract them.

7. Related Work

Extracting Data Models. Data model extraction attempts to infer from untyped code (*e.g.*, COBOL), 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 types 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 [9]. 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 (*e.g.*, [20]). 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 [14].) 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 [13]. 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 [15] for an overview), the work of Chong and Myers most resembles ours [3]. 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 [1]. 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 [4]. However, some of these flows might require statements along an infeasible path to execute. Path conditions rule out some of these infeasible flows [17].

Using PDGs with path conditions provides a sound way to determine if unconditional noninterference holds [18]. However, they do not directly extend to conditional noninterference. 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, 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 a program. Thus, others have created tools to extract these business rules from source code [8, 16]. These tools use program slicing to track information, but they do not provide the conditions that enable them.

8. Summary and Future Work

After presenting a formalization of conditional noninterference, we presented a sound method to extract from appli-

cation 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 additional language features of C. Any standard alias analysis could run before our algorithm runs to conservatively handle aliasing. However, an analysis that tracks the conditions that enables aliasing would provide more accurate results. An analysis for partitioning data structures into separate confidentiality levels based on the data placed in each field or array location would make our analysis more accurate. Such an addition would complicate the ref and def functions but not substantially change the algorithm. Adding unstructured control flow such as `goto` statements, on the other hand, may make our compositional approach no longer possible. However, a more standard iterative approach should still be possible.

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.

References

- [1] G. Barthe, P. R. D’Argenio, and T. 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.
- [2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
- [3] S. Chong and A. 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.
- [4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
- [5] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: a programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, 2002.
- [6] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. 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.
- [7] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, page 11, 1982.
- [8] H. 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.
- [9] R. 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.
- [10] D. McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symposium on Security and Privacy*, pages 161–166, 1987.
- [11] J. 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.
- [12] G. C. Necula, S. McPeak, S. P. Rahul, and W. 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.
- [13] K. R. O’Neill, M. R. Clarkson, and S. 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.
- [14] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
- [15] A. Sabelfeld and D. 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.
- [16] H. 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.
- [17] G. Snelling. 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.
- [18] G. Snelling, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.
- [19] F. Somenzi. CUDD: The CU decision diagram package. <http://vlsi.colorado.edu/~fabio/CUDD/>.
- [20] G. E. Suh, J. W. Lee, D. Zhang, and S. 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.
- [21] M. C. Tschantz and J. M. Wing. Confidentiality policies and their extraction from programs. Technical Report CMU-CS-07-108, School of Computer Science, Carnegie Mellon University, Feb. 2007.
- [22] M. C. Tschantz and J. M. Wing. Extracting conditional confidentiality policies. Technical Report CMU-CS-08-127, School of Computer Science, Carnegie Mellon University, May 2008.
- [23] J. T. Wittbold and D. M. Johnson. Information flow in non-deterministic systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 144–161, Los Alamitos, CA, USA, 1990.