Inference of expressive declassification policies

Jeffrey A. Vaughan
University of California, Los Angeles
jeff@cs.ucla.edu

Stephen Chong
Harvard School of Engineering and Applied Sciences
chong@seas.harvard.edu

***Abstract*—We explore the *inference of expressive human-readable declassification policies* as a step towards providing practical tools and techniques for strong language-based information security.**

Security-type systems can enforce expressive information-security policies, but can require enormous programmer effort before any security benefit is realized. To reduce the burden on the programmer, we focus on *inference* of expressive yet intuitive information-security policies from programs with few programmer annotations.

We define a novel security policy language that can express *what* information a program may release, under what conditions (or, *when*) such release may occur, and which procedures are involved with the release (or, *where* in the code the release occur). We describe a dataflow analysis for precisely inferring these policies, and build a tool that instantiates this analysis for the Java programming language. We validate the policies, analysis, and our implementation by applying the tool to a collection of simple Java programs.

***Keywords*-declassification policies, information flow, language based security, inference of security policies.**

I. Introduction

Many computer systems manipulate sensitive information, and it is important to reason about the information security of these systems. Recent work on language-based information-flow security provides both expressive information security policies, and techniques for enforcing these policies (e.g., [1, 2, 3, 4, 5, 6]). However, it can be difficult for programmers to obtain the security guarantees offered by these expressive policies, for at least two reasons. First, the security policies are sometimes unintuitive, requiring a sophisticated understanding of program semantics and noninterference-based semantic security conditions. Second, the most common enforcement technique, security-type systems [7], often requires enormous programmer effort before any security guarantees are achieved. For example, Askarov and Sabelfeld [8] report 150 person-hours to develop a security-typed implementation of a cryptographic protocol, compared to 60 person-hours for a non-security-typed implementation.

In this work, we move towards practical language-based information-flow security by:

1) Defining an expressive yet intuitive information security policy language.
2) Describing a dataflow analysis for precisely inferring these policies.
3) Building a tool that instantiates this analysis for the Java programming language.

We validate the policies, analysis, and our implementation by applying the tool to a collection of simple Java programs.

Inference of security policies. We focus on the *inference* of security policies for a program, as contrasted with the *a priori* specification of policies by a programmer via program annotations. During development programmers may not yet clearly understand program structure, and may have difficulty providing security-type annotations for program variables and function declarations.

Inference of security policies during the development process allows a programmer to understand the important information flows in a program, and to then decide if these flows are consistent with the security requirements of the program. If the inferred security policies describe information flows that the programmer or a security audit determine are suitable for the application, no further action needs to be taken. Otherwise, either the program contains insecure information flows, or the analysis is insufficiently precise; regardless, the programmer needs to invest more time and effort into the program's security, by modifying the program, or providing additional information to the analysis. Thus, by inferring security policies, the programmer can receive weak security guarantees with relatively little effort, and if those guarantees are too weak, can invest additional effort improving the program's security guarantees.

An additional benefit is that inference of security policies does not prevent program compilation or execution, and thus does not prevent functional testing or deployment of an application.

We aim to infer security policies for Java programs with few programmer annotations. More precisely, once the programmer has specified at least some of the program points where sensitive information enters the system, and information leaves the system, our tool can infer security policies that describe what sensitive information may be revealed to an observer of the system.

Expressive declassification policies. Our security policies give a concise and informative summary of the information flows in a program. Policies describe *what* sensitive information may be revealed to an observer of the system, *when* such information may be released, *which methods* in the code base were involved with release of sensitive information (a form of *where* declassification, according to the categorization by Sabelfeld and Sands [9]). The policy language is intended to be well-known and intuitive, balancing expressivity with ease of inference. (Of course, the same policy language could also be used for *a priori* specification.)

For example, consider the following Java-like program with annotations indicated by at signs (“@”).

```
1     public static void main(String args[]) {
2         ...
3         int creditCardNum = @input "cc" readCC();
4         String message = "No receipt.";
5         ...
6         if (@input "requestReceipt" requestReceipt()) {
7             message = lastFourDigits(creditCardNum);
8         }
9         System.out.println(@output "stdout" message);
10    }
11
12    static int lastFourDigits(int n) @track {
13        return n % 10000;
14    }
```

The **@input** annotations (lines 3 and 6) indicate where information enters the system, and provide names to refer to these inputs (“cc” and “requestReceipt” respectively). The **@output** annotation (line 9) indicates where information leaves the system, and provides a name to refer to the output.

The information that may be revealed at the output (i.e., by printing the contents of variable message) satisfies the security policy

$$\begin{array}{l}\mathsf{if}\ \mathsf{requestReceipt}[0]\ \mathsf{then}\\ \qquad \mathsf{Reveal}(\mathsf{cc}[0]\ \mathsf{mod}\ 10000),\end{array}$$

which intuitively means that if the most recent input named requestReceipt is true, then the last four digits of the most recent input named cc may be revealed; the value of the most recent input named requestReceipt may also be revealed, but no other information will be. Thus, the policy describes what information may be released ($\mathsf{cc}[0]\ \mathsf{mod}\ 10000$), and the conditions under which the release may occur ($\mathsf{requestReceipt}[0]$ evaluates to true). Specifically, this policy implies that if $\mathsf{requestReceipt}[0]$ is false, then an observer does not learn anything about $\mathsf{cc}[0]$.

This policy expresses *extensional* information security about the program, that is, security in terms of the input/output behavior of the program. Our policies can also represent certain kinds of *intensional* information security (i.e., security in terms of the implementation of the program), by describing which methods must be involved in the release of information. In the code above, the **@track** annotation (line 12) indicates that the programmer is interested in how the method lastFourDigits participates in information flows in the program. The output of the program also satisfies the more restrictive policy

$$\begin{array}{l}\mathsf{if}\ \mathsf{requestReceipt}[0]\ \mathsf{then}\\ \qquad \mathsf{if\text{-}executed}\ \mathsf{lastFourDigits}(\mathbf{int})\ \mathsf{then}\\ \qquad\qquad \mathsf{Reveal}(\mathsf{cc}[0]\ \mathsf{mod}\ 10000)\end{array}$$

which indicates that if $\mathsf{requestReceipt}[0]$ evaluates to true *and* the method $\mathsf{lastFourDigits}(\mathbf{int})$ was executed, then $\mathsf{cc}[0]\ \mathsf{mod}\ 10000$ may be revealed. This is indeed the policy that our inference algorithm infers for this program.

Policies

$$\begin{array}{rcll} p, q & ::= & \mathsf{Reveal}(e_1, \ldots, e_n) & \textit{Revelation policy}\\ & \mid & \mathsf{if}\ d\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 & \textit{Conditional policy}\\ & \mid & \mathsf{if\text{-}executed}\ r\ \mathsf{then}\ p & \textit{Track policy}\\ & \mid & p_1\ \mathsf{and}\ p_2 & \textit{Conjunctive policy}\end{array}$$

Input expressions

$$\begin{array}{rcl} e & ::= & \nu[i] \mid \nu[i{+}] \mid n \mid e_1 + e_2 \mid e_1 = e_2 \mid \ldots\\ \nu & \in & \textbf{ChannelName}\end{array}$$

Indices

$$i ::= 0 \mid 1 \mid 2 \mid \ldots$$

Precise input expressions

$$d ::= \nu[i] \mid n \mid d_1 + d_2 \mid d_1 = d_2 \mid \ldots$$

Track expressions

$$\begin{array}{rcl} r & ::= & k \mid r_1 \wedge r_2 \mid r_1 \vee r_2\\ k & \in & \textbf{Mark}\end{array}$$

Figure 1. Grammar for security policies

The rest of the paper is structured as follows. In Section II we describe the security policies in more detail, and use a simple imperative language to provide a formal semantics for the policies. In Section III we describe a dataflow analysis that can precisely infer security policies, and in Section IV we discuss our implementation of the analysis for the Java programming language. We have used this implementation to infer security policies for several Java programs, and we discuss our experience in Section V. Section VI summarizes related work, and we discuss possible extensions and conclude in Section VII.

II. Policies and security

Our security policies are intended to be expressive yet intuitive, and amenable to precise inference. In this section, we describe our policies intuitively, then present a formal semantics for the policies in terms of a simple imperative language.

A. Policies

Policies describe what an observer of a program may learn about inputs to the program. The syntax of our security policies is described by the grammar in Fig. 1.

An *input expression* $\nu[i]$ indicates the ith most recent input on the channel named ν. For example, $H[0]$ indicates the most recent input on channel H, and $H[1]$ indicates the next-to-last input on channel H. A *lifted index* $i+$ indicates all inputs prior to and including i. For instance $L[0+]$ indicates all input on channel L and $L[1+]$ indicates all inputs except for the most recent. A *precise* input expression is any input expression that contains no lifted indices. Thus, $H[0]$ is a precise input expression whereas $H[1+]$ is not.

A *revelation policy* $\mathsf{Reveal}(e_1, \ldots, e_n)$ says that an observer may learn the value of any and all of the input expressions e_1 to e_n. For example, policy $\mathsf{Reveal}(H[0])$ states that an observer of the program may learn everything about the most recent input on channel H. This policy is an accurate description of the information flow of the following program, for an observer of the output of channel L.

```
input x from H;  output x to L
```

Policy $\mathsf{Reveal}(H[0] \bmod 10000, H[1])$ is an accurate description of the following program for an L observer, who may learn both the last four digits of the most recent input, and everything about the next-to-last input.

```
input x from H;
input y from H;
z := (“Answers are ” . (y mod 10000) . “ and ” . x);
output z to L
```

A *conditional policy* $\mathsf{if}\ d\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$ indicates that an observer may learn the evaluation of precise input expression d. Moreover, in an execution in which d evaluates to true, p_1 describes additional information the observer may learn, and in an execution in which d evaluates to false, p_2 describes additional information that may be learned.

In the following program, input from the user on channel L determines whether a sensitive input from channel H is revealed.

```
input secret from H;
input password from H;
input guess from L;
 if (guess = password) then x := secret;  else  x := 42;
output x to L
```

An observer of channel L may learn information described by the policy

$$\begin{array}{l} \mathsf{if}\ H[0] = L[0]\ \mathsf{then} \\ \quad \mathsf{Reveal}(H[1]) \\ \mathsf{else} \\ \quad \mathsf{Reveal}(42). \end{array}$$

There are several things to note about this policy. First, in addition to possibly learning the value of secret, the observer may learn whether guess is equal to password ($H[0] = L[0]$). Second, the revelation policy $\mathsf{Reveal}(42)$ indicates that nothing is revealed about inputs from channel H, and is equivalent to the policy $\mathsf{Reveal}()$. The semantics of policies is discussed further in Section II-C. We write $\mathsf{if}\ d\ \mathsf{then}\ p$ as syntactic sugar for the policy $\mathsf{if}\ d\ \mathsf{then}\ p\ \mathsf{else}\ \mathsf{Reveal}()$.

A *track policy* $\mathsf{if\text{-}executed}\ r\ \mathsf{then}\ p$ indicates that if an execution matches track expression r then p describes information an observer may learn. Track expressions are conjunctions and disjunctions of *marks*, which are events that may occur during execution, for example, a particular procedure being called. For expository purposes, we assume that event mark k occurs when a command **mark** k is executed.

Thus, an execution trace matches mark k if the command **mark** k was encountered during execution. An execution trace matches track expression $r_1 \wedge r_2$ if it matches both r_1 and r_2, and matches $r_1 \vee r_2$ if it matches either r_1 or r_2. For example, in the following program, the information that may be learned by an observer is described by the policy $\mathsf{if\text{-}executed}\ \mathsf{CorrectGuess}\ \mathsf{then}\ \mathsf{Reveal}(H[0])$.

```
input secret from H;
input guess from L;
 if (guess > 100) then
     mark CorrectGuess;
     x := secret;
else
     x := 0;

output x to L
```

That is, for a given execution of the program, the observer may learn the value of secret only if the mark CorrectGuess occurred. If the mark did not occur, then the observer cannot learn the value of the most recent input from channel H. Thus, the following program does not satisfy the policy $\mathsf{if\text{-}executed}\ \mathsf{CorrectGuess}\ \mathsf{then}\ \mathsf{Reveal}(H[0])$.

```
input secret from H; output secret to L
```

Track policies provide assurance that information release is contingent on a particular event, such as the execution of a particular section of code that is trusted to release information.

Finally, a *conjunctive policy* $p_1\ \mathsf{and}\ p_2$ describes that an observer may learn information according to policy p_1 and information according to policy p_2.

B. A simple language

We present a simple imperative language in order to give a formal semantics for our security policies. The syntax is given in Fig. 2. We define a trace-based semantics for this language. *Traces* are finite sequences of *events*. An event is either an *input event input* ν n, indicating that the value n was input from channel ν, an *output event output* ν n, indicating that value n was output on channel ν, or a *mark event mark* k indicating that mark k occurred. We write $\cdot$ for the empty trace.

Fig. 3 presents the semantics of the language as a large-step operational semantics. Channel input is modeled as a nondeterministic choice of an input value. A *memory* σ is a finite map from variables **Var** to integers. Notation $\sigma(a) = n$ indicates that expression a evaluated to n using memory σ

Arithmetic expressions

$$a ::= n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid \dots$$

Boolean expressions

$$\begin{aligned} b ::= {} & \mathsf{true} \mid \mathsf{false} \mid a_0 = a_1 \mid a_0 > a_1 \\ & \mid \neg b \mid b_0 \land b_1 \mid \dots \end{aligned}$$

Variables

$$x \quad \in \mathbf{Var}$$

Commands

$$\begin{aligned} c ::= {} & x \mathrel{\mathsf{:=}} a \mid c_0; c_1 \mid \textbf{while}\ b\ \textbf{do}\ c \\ & \mid \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1 \mid \textbf{skip} \\ & \mid \textbf{mark}\ k \\ & \mid \textbf{input}\ x\ \textbf{from}\ \nu \\ & \mid \textbf{output}\ a\ \textbf{to}\ \nu \end{aligned}$$

Figure 2. Language syntax

to look up the values of any variable in a. We define the *complete trace semantics* of command c as the set of all traces that c may generate,

$$\llbracket c \rrbracket \triangleq \{t \mid \exists \sigma, \sigma'.\ (c, \sigma) \Downarrow \langle t \mid \sigma' \rangle\}.$$

For example, $\llbracket \textbf{while}\ \mathsf{true}\ \textbf{do skip} \rrbracket = \emptyset$ and $\llbracket \textbf{skip} \rrbracket = \{\cdot\}$ and

$$\begin{aligned} & \llbracket \textbf{input}\ x\ \textbf{from}\ H\textbf{;}\ \textbf{output}\ x\ \textbf{to}\ L \rrbracket = \\ & \qquad \{(\mathit{input}\ H\ n, \mathit{output}\ L\ n) \mid n \in \mathbb{Z}\}. \end{aligned}$$

C. Semantics of policies

We define the semantics of a security policy as an equivalence relation over traces. The equivalence relation corresponding to a policy prescribes which execution traces an observer should be unable to distinguish.

In order to define the semantics of security policies, we first need to define semantics for input expressions and track expressions. We regard imprecise expressions as representing an (infinite) set of precise expressions. Intuitively, input expression $\nu[i{+}]$ represents the set of precise expressions $\{\nu[i], \nu[i+1], \nu[i+2], \dots\}$. Other imprecise input expressions are defined homomorphically. We write $\mathit{preciseExprs}(e)$ for the set of precise expressions represented by input expression e. The definition is given in Fig. 4.

Given trace t, the evaluation of a precise input expression d, written $t(d)$, is a value in the set $\{\bot, \mathsf{true}, \mathsf{false}\} \cup \mathbb{Z}$. Intuitively, input expression $\nu[i]$ evaluates to the ith most recent input on channel ν. The evaluation of other precise input expressions is defined homomorphically. Fig. 4 gives a formal definition. Given trace t, the evaluation of a track expression r, written $t(r)$, is a value in the set $\{\mathsf{true}, \mathsf{false}\}$, indicating whether the marks indicated occurred in the trace. Fig. 4 gives the semantics.

$$\begin{aligned} \mathit{preciseExprs}(n) &= \{n\} \\ \mathit{preciseExprs}(\nu[i]) &= \{\nu[i]\} \\ \mathit{preciseExprs}(\nu[i{+}]) &= \{\nu[i], \nu[i+1], \nu[i+2], \dots\} \\ \mathit{preciseExprs}(e_1\, \mathit{op}\, e_2) &= \{d_1\, \mathit{op}\, d_2 \mid d_1 \in \mathit{preciseExprs}(e_1) \\ &\qquad \text{and } d_2 \in \mathit{preciseExprs}(e_2)\} \end{aligned}$$

$$\begin{aligned} t(n) &= n \\ \cdot(\nu[i]) &= \bot \\ (t, \mathit{input}\ \nu\ n)(\nu[0]) &= n \\ (t, \mathit{input}\ \nu\ n)(\nu[i]) &= t(\nu[i-1]) \\ (t, \_)(\nu[i]) &= t(\nu[i]) \\ t(d_1\ \mathit{op}\ d_2) &= \begin{cases} n & n = t(d_1)\ \llbracket \mathit{op} \rrbracket\ t(d_2) \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

where $\llbracket \mathit{op} \rrbracket$ is the usual interpretation of arithmetic or boolean operators.

$$\begin{aligned} t(k) &= \mathsf{true} \text{ if } \mathit{mark}\ k \in t \\ t(k) &= \mathsf{false} \text{ if } \mathit{mark}\ k \notin t \\ t(r_1 \mathbin{\pmb{\wedge}} r_2) &= t(r_1) \land t(r_2) \\ t(r_1 \mathbin{\pmb{\vee}} r_2) &= t(r_1) \lor t(r_2) \end{aligned}$$

Figure 4. Semantics of input expressions and track expressions

We can now define the semantics of policies. Given security policy p, we write $\llbracket p \rrbracket$ for the equivalence relation over traces that represents p. We define this equivalence relation in Fig. 5. Intuitively, a security policy limits the information that an observer is allowed to learn about a program's execution. If two traces t_1 and t_2 are related by security policy p (i.e., $(t_1, t_2) \in \llbracket p \rrbracket$), then according to policy p, an observer should not be allowed to distinguish t_1 from t_2. If $(t_1, t_2) \notin \llbracket p \rrbracket$, then policy p permits an observer to distinguish the two traces.

For example, revelation policy $\mathsf{Reveal}(e_1, \dots, e_n)$ intuitively permits an observer to learn the values of input expressions $e_1, \dots, e_n$, and so t_1 and t_2 are related by the revelation policy if and only if the two traces agree on the evaluation of each input expression e_i for $i \in 1..n$—that is, if $t_1(d) = t_2(d)$ whenever $d \in \mathit{preciseExprs}(e_i)$.

Traces are related by conditional policy $\mathsf{if}\ d\ \mathsf{then}\ p\ \mathsf{else}\ q$ if they agree on the evaluation of expression d, and are related by p or q, as appropriate, based on the evaluation of d. Similarly, traces are related by track policy $\mathsf{if\text{-}executed}\ r\ \mathsf{then}\ p$ if they agree on the evaluation of track expression r, and, when r evaluates to true, they are related by p.

Finally, traces are related by conjunction policy $p\ \mathsf{and}\ q$ if they are related by both p and q.

D. Policy ordering and normalization

The semantics of policies induces a partial order on security policies. We say policy p *reveals no more information than* policy q, written $p \sqsubseteq q$, if $\llbracket p \rrbracket \supseteq \llbracket q \rrbracket$. Intuitively, if p

$$\frac{\sigma(a) = n}{(x := a, \sigma) \Downarrow \langle \cdot \mid \sigma[\, x \mapsto n \,] \rangle} \qquad \frac{}{(\textbf{skip}, \sigma) \Downarrow \langle \cdot \mid \sigma \rangle} \qquad \frac{(c_1, \sigma) \Downarrow \langle t_1 \mid \sigma_1 \rangle \quad (c_2, \sigma_1) \Downarrow \langle t_2 \mid \sigma_2 \rangle}{(c_1\textbf{;}\ c_2, \sigma) \Downarrow \langle t_1, t_2 \mid \sigma_2 \rangle} \qquad \frac{\sigma(b) = \mathsf{false}}{(\textbf{while}\ b\ \textbf{do}\ c, \sigma) \Downarrow \langle \cdot \mid \sigma \rangle}$$

$$\frac{\begin{array}{c}\sigma(b) = \mathsf{true} \quad (c, \sigma) \Downarrow \langle t_1 \mid \sigma_1 \rangle \\ (\textbf{while}\ b\ \textbf{do}\ c, \sigma_1) \Downarrow \langle t_2 \mid \sigma_2 \rangle\end{array}}{(\textbf{while}\ b\ \textbf{do}\ c, \sigma) \Downarrow \langle t_1, t_2 \mid \sigma_2 \rangle} \qquad \frac{\sigma(b) = \mathsf{true} \quad (c_1, \sigma) \Downarrow \langle t_1 \mid \sigma_1 \rangle}{(\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \sigma) \Downarrow \langle t_1 \mid \sigma_1 \rangle} \qquad \frac{\sigma(b) = \mathsf{false} \quad (c_2, \sigma) \Downarrow \langle t_2 \mid \sigma_2 \rangle}{(\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \sigma) \Downarrow \langle t_2 \mid \sigma_2 \rangle}$$

$$\frac{}{(\textbf{mark}\ \ k, \sigma) \Downarrow \langle \mathit{mark}\ \ k \mid \sigma \rangle} \qquad \frac{}{(\textbf{input}\ x\ \textbf{from}\ \nu, \sigma) \Downarrow \langle \mathit{input}\ \nu\ n \mid \sigma[\, x \mapsto n \,] \rangle} \qquad \frac{\sigma(a) = n}{(\textbf{output}\ a\ \textbf{to}\ \nu, \sigma) \Downarrow \langle \mathit{output}\ \nu\ n \mid \sigma \rangle}$$

Figure 3. Language semantics

$$\begin{array}{lcll}
(t_1, t_2) & \in & [\![\mathsf{Reveal}(e_1, \ldots, e_n)]\!] & \text{if} \quad \forall d \in \bigcup_{i=1..n} \mathit{preciseExprs}(e_i).\ t_1(d) = t_2(d) \\
(t_1, t_2) & \in & [\![\mathsf{if}\ d\ \mathsf{then}\ p\ \mathsf{else}\ q]\!] & \text{if} \quad (t_1(d) = t_2(d) = \mathsf{true}\ \text{and}\ (t_1, t_2) \in [\![p]\!]) \\
 & & & \quad\ \ \text{or} \quad (t_1(d) = t_2(d) = \mathsf{false}\ \text{and}\ (t_1, t_2) \in [\![q]\!]) \\
(t_1, t_2) & \in & [\![\mathsf{if\text{-}executed}\ r\ \mathsf{then}\ p]\!] & \text{if} \quad (t_1(r) = t_2(r) = \mathsf{true}\ \text{and}\ (t_1, t_2) \in [\![p]\!]) \\
 & & & \quad\ \ \text{or}\ (t_1(r) = t_2(r) = \mathsf{false}) \\
(t_1, t_2) & \in & [\![p\ \mathsf{and}\ q]\!] & \text{if} \quad (t_1, t_2) \in [\![p]\!] \cap [\![q]\!]
\end{array}$$

Figure 5. Semantics of policies

$$\begin{array}{rcll}
(\cdot, \cdot) & \in & [\![M]\!]_{obs} & \\
(t_1, t_2) & \in & [\![M]\!]_{obs} & \text{if} \quad (t_2, t_1) \in [\![M]\!]_{obs} \\
(t_1, t_3) & \in & [\![M]\!]_{obs} & \text{if} \quad (t_1, t_2) \in [\![M]\!]_{obs}\ \text{and}\ (t_2, t_3) \in [\![M]\!]_{obs} \\
(t_1, (t_2, \mathit{mark}\ k)) & \in & [\![M]\!]_{obs} & \text{if} \quad (t_1, t_2) \in [\![M]\!]_{obs} \\
(t_1, (t_2, \mathit{input}\ \nu\ n)) & \in & [\![M]\!]_{obs} & \text{if} \quad (t_1, t_2) \in [\![M]\!]_{obs}\ \text{and}\ \nu \notin \mathit{obs} \\
(t_1, (t_2, \mathit{output}\ \nu\ n)) & \in & [\![M]\!]_{obs} & \text{if} \quad (t_1, t_2) \in [\![M]\!]_{obs}\ \text{and}\ \nu \notin \mathit{obs} \\
((t_1, \mathit{input}\ \nu\ n), (t_2, \mathit{input}\ \nu\ n)) & \in & [\![M]\!]_{obs} & \text{if} \quad (t_1, t_2) \in [\![M]\!]_{obs} \\
((t_1, \mathit{output}\ \nu\ n_1), (t_2, \mathit{output}\ \nu\ n_2)) & \in & [\![M]\!]_{obs} & \text{if} \quad (t_1, t_2) \in [\![M]\!]_{obs}\ \text{and}\ M(\nu) = p,\ \text{and} \\
 & & & \qquad (t_1, t_2) \in [\![p]\!]\ \text{implies}\ n_1 = n_2
\end{array}$$

Figure 6. Semantics of policy maps

reveals no more information than q, then any information that policy p permits an observer to learn, is also permitted by policy q: if p allows an observer to distinguish traces t_1 and t_2, then $(t_1, t_2) \notin [\![p]\!]$, and so $(t_1, t_2) \notin [\![q]\!]$, meaning that q also allows an observer to distinguish the two traces.

Given this ordering, the least upper bound of p and q is $p\,\mathsf{and}\,q$, and the bottom element of this partial order allows no information at all to be revealed: $\mathsf{Reveal}()$.

Many of the security policies are equivalent under this ordering. For example, the policies $\mathsf{Reveal}()$, $\mathsf{Reveal}(42)$, and $\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ \mathsf{Reveal}()\ \mathsf{else}\ \mathsf{Reveal}(1)$ are all equivalent, as are the policies $\mathsf{Reveal}(H[0])$ and $\mathsf{Reveal}(H[0] + 7)$.

Fig. 7 presents some policy equivalences, expressed as inference rules. Read from left to right, these equivalences provide rewrite rules to simplify policies while preserving semantic meaning. We refer to the process of applying rewrite rules as *normalization*: when a policy can no longer have any rewrite rules applied to it, it is in *normal form*. Normalization is critical during inference as it reduces the number of policies that occur during the analysis, and ensures termination of the analysis.

E. Security

We assume there is an observer who can see the inputs and outputs occurring on some set of channels $\mathit{obs} \subseteq$ **ChannelName**. Given trace t we define the *projection of t on channels obs*, written $\lfloor t \rfloor_{obs}$, to be the subsequence of t consisting of all and only events on channels in obs.

$$\begin{array}{rcll}
\lfloor \cdot \rfloor_{obs} & = & \cdot & \\
\lfloor t, \mathit{input}\ \nu\ n \rfloor_{obs} & = & \lfloor t \rfloor_{obs}, \mathit{input}\ \nu\ n & \text{if}\ \nu \in \mathit{obs} \\
\lfloor t, \mathit{input}\ \nu\ n \rfloor_{obs} & = & \lfloor t \rfloor_{obs} & \text{if}\ \nu \notin \mathit{obs} \\
\lfloor t, \mathit{output}\ \nu\ n \rfloor_{obs} & = & \lfloor t \rfloor_{obs}, \mathit{output}\ \nu\ n & \text{if}\ \nu \in \mathit{obs} \\
\lfloor t, \mathit{output}\ \nu\ n \rfloor_{obs} & = & \lfloor t \rfloor_{obs} & \text{if}\ \nu \notin \mathit{obs} \\
\lfloor t, \mathit{mark}\ k \rfloor_{obs} & = & \lfloor t \rfloor_{obs} &
\end{array}$$

We say that traces t_1 and t_2 are *observationally equivalent* to an observer obs if and only if $\lfloor t_1 \rfloor_{obs} = \lfloor t_2 \rfloor_{obs}$.

$$\frac{i \le j}{\mathsf{Reveal}(H[j], H[i+]) = \mathsf{Reveal}(H[i+])} \qquad \frac{j = i + 1}{\mathsf{Reveal}(H[j], H[i+]) = \mathsf{Reveal}(H[j+])} \qquad \frac{p = p' \quad q = q'}{\mathsf{if}\ d\ \mathsf{then}\ p\ \mathsf{else}\ q = \mathsf{if}\ d\ \mathsf{then}\ p'\ \mathsf{else}\ q'}$$

$$\frac{}{\mathsf{if}\ d\ \mathsf{then}\ p\ \mathsf{else}\ p = \mathsf{Reveal}(d)\ \mathsf{and}\ p} \qquad \frac{}{p\ \mathsf{and}\ p = p} \qquad \frac{}{\mathsf{if\text{-}executed}\ r\ \mathsf{then}\ \mathsf{if\text{-}executed}\ r'\ \mathsf{then}\ p = \mathsf{if\text{-}executed}\ (r \wedge r')\ \mathsf{then}\ p}$$

$$\frac{}{(\mathsf{if}\ d\ \mathsf{then}\ p_1\ \mathsf{else}\ q_1)\ \mathsf{and}\ (\mathsf{if}\ d\ \mathsf{then}\ p_2\ \mathsf{else}\ q_2) = \mathsf{if}\ d\ \mathsf{then}\ (p_1\ \mathsf{and}\ p_2)\ \mathsf{else}\ (q_1\ \mathsf{and}\ q_2)}$$

Figure 7. Selection of policy equivalences

In order to specify what it means for a program to be secure with respect to security policies, we need some way of specifying what information an observer of the system should be allowed to learn. A *policy map* M is a map from channel names to security policies. Intuitively, $M(\nu)$ is a security policy that describes the information that may be released over channel ν.

A program is secure for observer obs with respect to policy map M if the observer can learn at most information according to policies $M(\nu)$, for $\nu \in obs$. We define $[\![M]\!]_{obs}$ to be the equivalence relation representing the information an observer obs should be allowed to learn. Intuitively, traces t_1 and t_2 are equivalent according to $[\![M]\!]_{obs}$ when the following conditions hold.

(i) t_1 and t_2 agree on the order of input and output events for all channels in obs.
(ii) t_1 and t_2 agree on the input values for pairs of corresponding input events on channels in obs.
(iii) All information output to channel ν is described by $M(\nu)$. That is, for each pair of corresponding ν-output events in traces t_1 and t_2, the output values agree whenever the traces immediately prior to the output event are related by $[\![M(\nu)]\!]$.

The formal definition of $[\![M]\!]_{obs}$ is given in Fig. 6.

A program is secure if observational equivalence is no more precise than the equivalence relation $[\![M]\!]_{obs}$.

Definition 1 (Security)**.** Program c is secure with respect to policy map M if for all $obs \subseteq$ **ChannelName**, and $t_1, t_2 \in [\![c]\!]$, $(t_1, t_2) \in [\![M]\!]_{obs}$ implies $\lfloor t_1 \rfloor_{obs} = \lfloor t_2 \rfloor_{obs}$.

Intuitively, the definition requires that for an observer of channels obs, if the policy map M says that the observer should not be able to distinguish two executions of the program ($(t_1, t_2) \in [\![M]\!]_{obs}$), then it is indeed the case that the observer cannot distinguish them ($\lfloor t_1 \rfloor_{obs} = \lfloor t_2 \rfloor_{obs}$). This definition is a standard noninterference-based [10] definition, and as such, cannot be expressed as a predicate over just a single execution trace [11].

III. Inference

Security policies describe what information may be learned by an observer of a program execution. However, specification of policies can be onerous (e.g., [8]), since it requires the programmer to explicitly state the security policies the program is intended to satisfy. Instead, using techniques inspired by security-type systems [7], we focus on the *inference* of security policies from programs with few annotations. Policy inference frees the programmer from needing *a priori* knowledge of the security policies a program should satisfy.

In this section we describe how our security policies can be precisely inferred with few security annotations from the programmer. We present examples using the imperative language of Section II-B, but describe how these techniques extend to more general languages. In Section IV we describe how we apply these techniques to Java programs.

The challenge is to soundly infer precise security policies with few programmer annotations. The expressiveness of our security policies presents opportunities for greater precision in the inferred policies than can be obtained by generalizing standard information-flow type-systems. We use a dataflow analysis to infer security policies for a program. We first define the dataflow information our analysis propagates, including how this information is merged, then describe transfer functions for the commands in the simple imperative language. We also describe how a policy map can be extracted from the dataflow information, thus inferring security policies for the program, and discuss the soundness of our analysis.

A. Contexts

The dataflow information our analysis associates with every program point is a context $\langle \Gamma, pc \rangle$, where Γ is a *variable context* that maps every program variable to a security policy, and *pc* is a *program counter map*. A variable context is a function from program variables to security policies that records for each variable an upper bound on the information that may be stored in that variable at that program point. A program counter map (or *pc-map* for brevity) is used to track what information may be learned by knowing that execution has reached this program point. It is similar to the *program counter policies* used in security-type systems [7], but contains additional structure to facilitate precise inference of our security policies.

Program counter maps. The domain of a program counter map is a subset of the branch points in the program's

```
input x from H;
input y from H;
z := ...;  // some complicated computation involving y
 if  (x > 42) then  w := 0
else
    if  (z = 0)  then {   P: w := 1 } else { w := 2   }
```

Figure 8. Program counter map example

control flow graph. Given a pc-map pc for some program point, a branch point b is in the domain of pc if and only if the program point is control dependent on b. In the simple imperative language of Section II-B, the only branch points are the guards of **if** and **while** commands. A program point is control dependent on the guard of an **if** command only if the program point is within one of the two branches of the command. Since we are concerned with termination-insensitive security, a program point is control dependent on the guard of a **while** loop only if the program point is within the body of the loop. (If we were concerned with termination sensitive security, program points after a possibly diverging **while** loop would also be control dependent on the loop guard.)

A branch point b in pc-map pc maps either to a security policy, or a precise input expression. If b maps to security policy p, then p is an upper bound of the information used to decide which control flow path to take at b: it is the policy associated with the guard of the **if** or **while** statement. If, however, we know (through some external analysis) that the control flow decision at b is determined by the evaluation of some precise boolean input expression d, then b maps to either d or $\neg d$, depending on which branch was taken.

Consider the example program in Fig. 8. Let P be the program point just prior to assignment w := 1. At P control flow is dependent on the branch points x $>$ 42 and z = 0, and thus both of these branch points are in the domain of the pc-map at program point P. The control flow path taken at branch point x $>$ 42 is determined by the precise input expression $H[1] > 42$, and moreover, at program point P, we know that $H[1] > 42$ is false. The control flow path taken at branch point z = 0 depends on information bounded above by the policy Reveal($H[0]$). However, (by assumption) we do not know whether path taken at this branch is determined by any precise security expression. The pc-map for P is thus

$$\{\mathsf{x} > 42 \mapsto \neg(H[1] > 42), \quad \mathsf{z} = 0 \mapsto \mathsf{Reveal}(H[0])\}.$$

Merging contexts. At program points where control flow paths merge (such as the program point following an **if** command, or the head of a loop), contexts must be soundly merged. The merge of two contexts $\langle\Gamma_1, pc_1\rangle$ and $\langle\Gamma_2, pc_2\rangle$ is $\langle\Gamma_1 \sqcup \Gamma_2, pc_1 \sqcup pc_2\rangle$, where $\Gamma_1 \sqcup \Gamma_2$ denotes the merge of the variable contexts, and $pc_1 \sqcup pc_2$ denotes the merge of the pc-maps.

Variable contexts are merged pointwise. That is, given two variables contexts Γ_1 and Γ_2, for each variable x, $\Gamma_1 \sqcup \Gamma_2$ maps x to the policy $p_1 \,\mathsf{and}\, p_2$, where $\Gamma_1(x) = p_1$ and $\Gamma_2(x) = p_2$.

The merge of two pc-maps, $pc_1 \sqcup pc_2$, is defined in Fig. 9. The merge is pointwise, with the exception of when one of pc_1 and pc_2 maps branch point b to a precise security expression, and the other maps b to something else, either a different precise security expression, or a security policy. In that case, we no longer have precise information about control flow, and the merged pc-map $pc_1 \sqcup pc_2$ maps b to a sound but less precise security policy.

Both variable context and program counter merges may yield policies with unnecessarily large representations. We normalize the policies, as described in Section II, to ensure policies are represented concisely. Normalization both improves policy readability and is needed to ensure termination of the analysis.

Improving precision of pc-maps. At control flow points that are immediate post dominators of branch points, we can improve the precision of program counter maps, by restoring the program counter map that was in place at the branch point. Intuitively, once the post dominator of branch point b has been reached, control flow is not dependent on b. From a (termination insensitive) information-flow perspective, knowing that execution has reached the post dominator of b reveals exactly as much information as knowing that execution reached branch point b.

Given a control flow graph of a program, we insert additional nodes to distinguish immediate post dominators of branch points. The transfer functions for these additional nodes improve the precision of the pc-map by restoring it to the pc-map of the post-dominated program point.

In the example program of Fig. 8, the final program point is the immediate post dominator of both branches x $>$ 42 and z = 0. We restore the pc-map for the program point immediately before the branch x $>$ 42.

In the following subsections, we describe the transfer functions for the remaining commands in our simple imperative language.

B. Transfer function for assignment

Intuitively, the transfer function for assignment x := a must update the variable context to record the information that may be learned by examining the contents of variable x after the assignment. In addition to learning the evaluation of expression a, one also learns that the assignment occurred. This is known as an *implicit information flow* [12], where information flows through the control structure of a program. Our analysis precisely tracks implicit information flows using the program counter maps.

Consider an assignment x := a occurring in context $\langle\Gamma, pc\rangle$. Let $p_a = \Gamma(x_1)\,\mathsf{and} \ldots \mathsf{and}\,\Gamma(x_n)$, where

$$pc_1 \sqcup pc_2(b) = \begin{cases} pc_1(b) & \text{if } b \in \mathit{dom}\ (pc_1) \wedge b \notin \mathit{dom}\ (pc_2) \\ pc_2(b) & \text{if } b \notin \mathit{dom}\ (pc_1) \wedge b \in \mathit{dom}\ (pc_2) \\ pc_1(b) & \text{if } pc_1(b) = pc_2(b) \\ p_1 \,\mathsf{and}\, p_2 & \text{if } pc_1(b) = p_1 \text{ and } pc_2(b) = p_2 \text{ for some security policies } p_1 \text{ and } p_2 \\ \mathsf{Reveal}(d, d') & \text{if } \{pc_1(b), pc_2(b)\} = \{d, d'\} \text{ for distinct precise input expressions } d, d' \\ \mathsf{Reveal}(d) \,\mathsf{and}\, p & \text{if } \{pc_1(b), pc_2(b)\} = \{d, p\} \text{ for some precise input expression } d \text{ and policy } p \end{cases}$$

Figure 9. Merge of program counter maps

$x_1, \ldots, x_n$ are the variables that occur in expression a. Policy p_a is an upper bound of the information that may be learned from the evaluation of a. Let the range of pc be $d_1, \ldots, d_n, p_1, \ldots, p_m$ for precise input expressions $d_1, \ldots, d_n$ and security policies $p_1, \ldots, p_m$. That is, any branch point in the domain of pc maps to one of these precise input expressions or security policies.

The transfer function for assignment maps variable x to the policy

$$\mathsf{if}\ d_1 \wedge \cdots \wedge d_n\ \mathsf{then}\ p_a\ \mathsf{and}\ p_1\ \mathsf{and} \ldots \mathsf{and}\ p_m.$$

(If there are no precise input expressions in the range of pc, i.e., $n = 0$, then x is mapped to the policy $p_a\ \mathsf{and}\ p_1\ \mathsf{and} \ldots \mathsf{and}\ p_m$.)

This rule records that the variable x contains information bounded above by the policy $p_a\ \mathsf{and}\ p_1\ \mathsf{and} \ldots \mathsf{and}\ p_m$ only if the input expressions $d_1, \ldots, d_n$ are all true. Intuitively, the security policy now associated with x describes what information may be learned by examining the contents of x: one learns information only if input expressions $d_1, \ldots, d_n$ are all true, and if so, one learns both the evaluation of a (p_a), and the fact that the assignment occurred ($p_1\ \mathsf{and} \ldots \mathsf{and}\ p_m$, and implicitly, $d_1 \wedge \cdots \wedge d_n$).

As an example, consider again the program in Fig. 8. At the program point immediately following the command w := 1, the context maps variable w to the policy $\mathsf{if}\ \neg(H[1] > 42)\ \mathsf{then}\ \mathsf{Reveal}(H[0])$.

Improving precision at merge points. Consider the following program.

```
input x from H; input y from H; input z from H;
 if (z = 7) then w := x else w := y
```

At the program point immediately following the assignment w := x, the context maps program variable w to the policy $\mathsf{if}\ H[0] = 7\ \mathsf{then}\ \mathsf{Reveal}(H[2])$. At the program point after assignment w := y, the context maps w to the policy $\mathsf{if}\ \neg(H[0] = 7)\ \mathsf{then}\ \mathsf{Reveal}(H[1])$. Immediately following the **if** command, the contexts from the two branches are merged, resulting in w mapping to policy

$$\mathsf{if}\ H[0] = 7\ \mathsf{then}\ \mathsf{Reveal}(H[2])\ \mathsf{else}\ \mathsf{Reveal}(H[1]),$$

a precise summary of what information may be learned by observing the contents of w at that program point. Namely, the value of expression $H[0] = 7$ may be learned, and if that expression is true, then the 3rd most recent input on channel H may be learned ($\mathsf{Reveal}(H[2])$), otherwise the 2nd most recent input on channel H may be learned ($\mathsf{Reveal}(H[1])$).

The following program is semantically equivalent to the program above.

```
input x from H; input y from H; input z from H;
w := y;
 if (z = 7) then w := x else skip
```

However, using the transfer functions and merging described so far, at the end of the command, w would have the policy

$$\mathsf{Reveal}(H[1])\ \mathsf{and}\ \mathsf{if}\ H[0] = 7\ \mathsf{then}\ \mathsf{Reveal}(H[2]),$$

which is sound, but not as precise as the previous policy.

To improve precision, when control flow paths merge, if variable x may have been updated on one path but not the other, we insert a self-assignment $x := x$ at the end of the path on which x was not updated. Thus, we analyze the example above as if it were the following.

```
input x from H; input y from H; input z from H;
w := y;
 if (z = 7) then w := x else w := w
```

This additional analysis is sound (since the self-assignment is semantically a no-op), and provides additional precision. The resulting policy for w is precisely inferred to be $\mathsf{if}\ H[0] = 7\ \mathsf{then}\ \mathsf{Reveal}(H[2])\ \mathsf{else}\ \mathsf{Reveal}(H[1])$.

C. Mark commands

A mark command **mark** k is semantically a no-op. The transfer function for mark commands is just the identity function, and so a mark command does not directly update the context.

However, we use mark commands to produce track policies $\mathsf{if\text{-}executed}\ r\ \mathsf{then}\ p$. We use a dominance algorithm to identify assignments that are dominated by mark command **mark** k (that is, in any execution of the program the mark command must execute before the assignment). For any such assignment $x := a$, we insert an additional node into the control flow graph immediately following the assignment that indicates this. If the context on entry to one of these inserted nodes is $\langle \Gamma, pc \rangle$, the transfer function for the node updates the policy of variable x to $\textsf{if-executed}\, k\, \textsf{then}\, p$, where $\Gamma(x) = p$. This records that the information that may be learned by examining the contents of variable x at this program point is conditional on the execution of mark command **mark** k.

D. Transfer function for input commands

An input command **input** x **from** ν reads an input and assigns it to variable x. Thus, following the command, variable x contains the most recent input. The transfer function for an input command thus treats it like an assignment to the variable x of an expression with a security policy $\textsf{Reveal}(\nu[0])$. The actual policy associated with x following the input depends on the program counter map, as described in Section III-B.

In addition, the transfer function for an input command increments the index of inputs on channel ν in all policies in the context and program counter policy.

For example, if program variable y is mapped to policy

$$\textsf{if}\ \nu[1] = 42\ \textsf{then}\ \textsf{Reveal}(\nu[0])$$

before command **input** x **from** ν, then after the command the analysis will associate y with policy

$$\textsf{if}\ \nu[2] = 42\ \textsf{then}\ \textsf{Reveal}(\nu[1]).$$

Bounded precision for inputs. In order to ensure that the analysis terminates, we cannot track the history of inputs with unbounded precision. For each input channel ν, we assume an upper bound b_ν on the indices that we track precisely, and summarize all indices greater than or equal to b_ν with an imprecise input expression $\nu[b_\nu+]$. For example, if $b_\nu = 2$ and program variable z maps to $\textsf{Reveal}(\nu[1] \mod 5)$ before command **input** x **from** ν, then after the command the analysis will associate z with policy $\textsf{Reveal}(\nu[2+] \mod 5)$.

A conditional policy, say $\textsf{if}\ d\ \textsf{then}\ p_1\ \textsf{else}\ p_2$, is required to have a precise expression its guard. If a guard becomes imprecise, we must change the policy to a conjunction. Specifically, if d is a precise expression, and e is the imprecise expression obtained by incrementing the indices for inputs on some channel, then conditional policy $\textsf{if}\ d\ \textsf{then}\ p_1\ \textsf{else}\ p_2$ becomes $\textsf{Reveal}(e)\ \textsf{and}\ p'_1\ \textsf{and}\ p'_2$, where p'_1 and p'_2 are the result of incrementing the indices of inputs on policies p_1 and p_2 respectively.

For example, if program variable y maps to policy

$$\textsf{if}\ \nu[1] = 42\ \textsf{then}\ \textsf{Reveal}(\nu[0])$$

before command **input** x **from** ν, and $b_\nu = 2$, then after the command the analysis will associate y with policy

$$\textsf{Reveal}(\nu[2+] = 42)\ \textsf{and}\ \textsf{Reveal}(\nu[1]).$$

E. Output commands and policy maps

The command **output** a **to** ν evaluates a and outputs it on channel ν. The transfer function for this rule does not modify the context. Instead the static analysis uses output commands to build a policy map explaining the information flows in a program.

Intuitively, observation of the result of output command **output** a **to** ν may reveal both the evaluation of expression a, and also the fact that the output occurred. This is similar to the information flow that occurs through assignment, and by a similar argument, a suitable policy for the output is

$$\textsf{if}\ d_1 \wedge \cdots \wedge d_n\ \textsf{then}\ p_a\ \textsf{and}\ p_1\ \textsf{and} \ldots \textsf{and}\ p_m$$

where $d_1, \ldots, d_n, p_1, \ldots, p_m$ is the range of the pc-map at the output command, and p_a describes the information that may be learned from the evaluation of a.

The ultimate goal of the inference is to generate a policy map M that soundly describes what information may be learned by observing the output channels of the program when it executes. We define this policy map M as follows.

Suppose that the dataflow algorithm has terminated, and found appropriate contexts for every program point. Further suppose that the program contains exactly n output commands to channel ν. Let $p_1, \ldots, p_n$ be the security policies describing the information that may be learned by examining the result of each of these n output commands. Then policy map M maps channel ν to the conjunction of these policies:

$$M(\nu) = p_1\ \textsf{and} \ldots \textsf{and}\ p_n$$

F. Soundness and termination

The inference algorithm is sound if the program c is secure with respect to the policy map M produced at the end of the dataflow, where security is as defined in Definition 1.

We argue that our inference algorithm is sound. The intuition behind the algorithm is that at each program point, the context conservatively describes the information that may be learned by knowing that control flow has reached that program point, and the information that may be learned by examining the contents of each variable at that program point. The transfer functions and merge operation generalize information-flow typing rules for security type systems such as Jif [13], and we thus believe them to be sound. Standard results for the soundness of dataflow analyses then entail soundness of our analysis.

We also argue that our dataflow analysis always terminates. There are only finitely many contexts that may occur during dataflow analysis. This is due to the following facts.

1) Only finitely many precise input expressions are used during the dataflow analysis, since a precise input expression is used in the dataflow only if it was part of the output of a previous analysis (see Section III-A).
2) Only finitely many input expressions are used during the dataflow analysis, since indices of inputs $\nu[i]$ are

bounded, and only expressions occurring within the program are used in policies.

3) Only finitely many marks are used during the dataflow analysis, since there are only finitely many mark commands in a program.
4) Only finitely many policies are used during the dataflow analysis, since policies are normalized (Section II) before they are stored in the context, and given finitely many precise input expressions, input expressions and marks, there are only finitely many normalized policies.

We define a partial order $\leq$ over contexts, where $\langle \Gamma_1, pc_1 \rangle \leq \langle \Gamma_2, pc_2 \rangle$ if and only if both $\Gamma_1 \sqsubseteq \Gamma_2$ and $pc_1 \sqsubseteq pc_2$, where we extend the security policy partial order $\sqsubseteq$ pointwise to variable environments and pc-maps.[1] The transfer functions used in the dataflow analysis are monotonic with respect to the partial order $\leq$, and the context merge operation is an upper-bound operation for this partial order. Thus, a standard work-queue algorithm for the dataflow analysis will always find a fixed-point and terminate.

IV. Implementation

We have implemented the dataflow analysis for precise inference of security policies, described in Section III, as an interprocedural object-sensitive dataflow analysis for the Java programming language.

The implementation is a 35,100 non-comment non-blank line extension for the Polyglot compiler framework [14], a source-to-source compiler for the Java 1.4 programming language. This extension is factored into 3 parts: 10,400 lines implement an object-sensitive pointer analysis [15] and an interprocedural dataflow framework using this pointer analysis; 11,400 lines implement a generic information-flow dataflow analysis that can be specialized for different security policies; and 13,300 lines specialize this information-flow dataflow analysis for our security policies.

In this section, we discuss the extensions to the analysis of Section III required to build a practical security policy inference analysis for Java.

A. *Programmer annotations*

We aim to infer expressive security policies with few annotations from the programmer. However, our security policies rely on identifying sources of sensitive information (inputs) and observable sinks of information (outputs). While our tool could attempt to infer these inputs and outputs (for example, by determining which InputStreams and OutputStreams may be connected to the network), we have not done so. Instead, we rely on the programmer to provide annotations indicating these sources and sinks. We emphasize that the programmer does not need to provide annotations on fields, classes, or method headers before being able to use our tool.

The programmer indicates inputs and outputs using the annotations **@input** ν e and **@output** ν e respectively, where ν is a literal string indicating the name of the input or output, and e is an arbitrary Java expression. Input and output annotations may occur in a program anywhere a Java expression can occur. Input and output names are intended to be unique within a program. A programmer can optionally indicate the precision with which to track a particular input. By default, we use precision 1, meaning that for input channel ν, we track only input expressions $\nu[0]$ and $\nu[1+]$.

As shown in the Introduction's example code, programmers can use the **@track** annotation in a method header. This indicates that a method is security relevant and that its name should be used to build if-executed policies. Intuitively, this is similar to starting a method body with a **mark** *method-name* command.

Improving the inference results. If the results of the inference are in accordance with the programmer's expectations about the security requirements of the program, the programmer has received sufficient security assurance. However, if the results of the analysis do not meet the programmer's expectations, it could be due either to an actual security issue in the program, or to imprecision in the inference result. To address the later possibility, we provide additional mechanisms for the programmer to improve the precision of the analysis.

Implicit information flows can be a significant source of imprecision, and, as King et al. [16] demonstrate, particularly implicit information flows arising from conservative handling of unchecked exceptions. To mitigate this source of imprecision, we allow programmers to indicate that an expression or statement e cannot throw an exception of class E (or subclass thereof) by using the **@suppress** E e annotation. This improves the precision of the control flow graph, and also improves the precision of the inference analysis, since it reduces the spurious information flows that the inference conservatively assumes may occur due to exceptional control flow. If the programmer is incorrect in her assertion that an expression cannot throw a certain exception, then the analysis results may be unsound.

B. *Precise input expressions*

The analysis can track implicit flows precisely when a branching decision is determined by a precise input expression. Our analysis of Section III assumes that this information is available through some other analysis, and does not itself compute this information. We therefore implement an additional interprocedural analysis to determine which program expressions are equal to precise input expressions. This is another information flow analysis, albeit one that

[1] The co-domain of pc-maps is both security policies and precise input expressions; we extend the partial order over security policies by treating precise input expression d as if it were the security policy Reveal(d).

tracks only explicit information flows, and ignores implicit flows; it is similar to a constant propagation analysis.

C. Non-structured control flow

Unlike the simple imperative language of Section II-B, Java has unstructured control flow, due to exceptions and **break**, **continue**, and **return** commands. This complicates the dataflow analysis in a few ways.

First, control flow may be unstructured, and more work is required to determine the control flow graph for the program. Since our security policy inference analysis tracks implicit information flows, improving the precision of the control flow graph improves the precision of the inferred security policies. The key sources of imprecision in constructing the control flow graph are dynamic dispatch (where the method body to execute on method invocation depends on the runtime class of the receiver object), and exceptions (where control flow depends on the runtime class of the thrown exception). The pointer analysis provides information about the possible runtime classes of both receiver objects and thrown exceptions. In addition, we perform an interprocedural dataflow analysis to remove spurious exceptions. For example, if we can prove that the receiver of a field access or method invocation can never be null, then that operation can never throw a NullPointerException. Similarly, given an unsafe cast operation $(C)e$, if the points-to set of expression e indicates that e always points to a subclass of class C, then the cast can never throw a ClassCastException.

Second, the immediate post dominators of branch points can no longer be determined by examining just the syntax of the program. We instead implement an intraprocedural post-dominance algorithm [17] to determine the immediate post dominators of branch points. We use the output of this algorithm to restore the program counter map at the post dominator of a branch point to its value at the branch point, as described in Section III-A. An interprocedural post-dominance algorithm [18] would provide more precise results, allowing us to potentially restore more program counter maps. However, inter-procedural post-dominator analysis would improve precision only in specific instances, with fairly complicated control flow. In our experience, such patterns of control flow occur rarely in actual code, and we have found that the intraprocedural analysis provides sufficiently precise results.

Finally, we treat **return**s and **throw**s like assignments, in that we taint the value returned with the program counter map in use at the program point where the **return** or **throw** occurs. This is because the program point at which a **return** or **throw** occurs may reveal sensitive information, as demonstrated in the following code snippet, where the value of variable x depends on the sensitive input $H[0]$.

```
int foo(boolean sensitiveInfo) {
  if ( sensitiveInfo ) return 0;
  return 1;
}
...
int x = foo(@input "H''  ...);
```

D. Interprocedural object-sensitive analysis

We have implemented the object-sensitive pointer analysis of Milanova et al. [15]. We perform an object-sensitive interprocedural dataflow analysis, using the results of this pointer analysis to determine the different contexts in which to analyze code. The sensitivity of the pointer analysis can be adjusted, affecting the number of different contexts for code analysis: increased sensitivity in the pointer analysis leads to more contexts for code analysis (and thus more precision).

Object-sensitive pointer analysis allows our interprocedural dataflow analysis to maintain greater precision (over context-sensitive pointer analysis) for many common object-oriented coding patterns, such as collections, and field access.

However, using just object-sensitivity can lead to imprecision in our analysis for some common patterns, such as getter-methods and methods that compute a function of its arguments (such as math library methods like Math.max(**int**, **int**)). For example, consider the following code.

```
1    class C {
2        int f;
3        int getF() { return this.f; }
4    }
5    ...
6    C obj = new C();
7    if (@input "H''  ...)  {
8        int x = obj.getF();
9    }
10   int y = obj.getF();
```

Using object-sensitive interprocedural analysis, both calls to the method C.getF() on the object obj use the same analysis result, since they have the same receiver object. Thus, the method is analyzed using a pc-map that merges the pc-maps of both call sites. Since the pc-map of the first call site (line 8) contains sensitive information, and **return**s taint the returned value with the pc-map, we incorrectly assume that the value returned at the second call site (line 10) may reveal sensitive information.

To maintain precision with in these cases, we analyze methods at the leaves of the call graph based on both object-context and call site. That is, any method that does not invoke other code is analyzed separately for each call site and object context. This avoids the problem with imprecise pc-maps for many common cases, such as getter-methods and math library functions, without unreasonable additional computation.

Program	Lines of Code	Annotations	Inference time (sec)	Precise input expr. analysis time (sec)	Jif LoC	Jif type annotations
Chat	79	2	<1	<1		
Password Manager	184	3	<1	<1		
Access control	270	8	<1	<1		
Battleship	326	6	9	14	337	∼195
Go Fish	385	18	14	5		
Mental Poker	1369	6	470	15	3253	∼1175

Table I
CASE STUDIES

E. Library classes

We provide signatures for standard Java library classes. We must provide signatures for all analyses our tool performs, including pointer analysis, the analysis to remove of spurious exceptions (discussed in Section IV-C), and the information flow analyses—a single signature suffices for both the inference of security policies, and the analysis to determine what program expressions are equal to precise input expressions (discussed in Section IV-B).

The use of signatures improves the performance of the tool, since we do not need to analyze the standard Java library code. It also avoids difficulties analyzing the numerous **native** methods in the standard Java library. However, the signatures are a potential source of both unsoundness and imprecision, as there is no guarantee that the signatures accurately reflect the behavior of the library code.

V. CASE STUDIES

We have used the Java tool for inference of security policies on several Java programs. All the programs are of modest size, the largest having 1,400 non-comment non-blank lines of code. However, all have interesting information-security requirements. Two of the programs have equivalent versions written in the Jif programming language [19, 13], an extension of Java with information-security types.

In this section we report on our experiences using the tool, including useful idioms for annotating code, and the security policies the tool was able to infer.

Table I summarizes the programs we used, and the time taken to perform the analysis. Four are programs we wrote ourselves. We first wrote pure Java code, debugged our implementations, and then added annotations, which occasionally required minor refactoring of the code. The Battleship program is a Jif program bundled with the Jif distribution. We removed the Jif type annotations to obtain a Java program. The Mental Poker program was written by Askarov and Sabelfeld [8], who implemented two versions, one in Java (for which we inferred policies), and subsequently one in Jif. For the programs with Jif versions, we include the number of lines of code for the Jif version, and the approximate number of Jif type annotations.

The fourth and fifth columns of the table describe the time taken to perform, respectively, the inference analysis, and the precise input expression analysis (used to determine when program expressions are equivalent to a precise input expression). Times are averaged over five executions, all performed on a Mac Pro dual 2.26 GHz quad-core Intel Xeon with 8 GB of RAM. All analyses used a 2-full-object-sensitive pointer analysis with a heap context depth of 1 [15, 20] except for Battleship (3-object sensitivity) and Mental Poker (context insensitive). All pointer analyses took less than 1 second to complete.

A. Go Fish

This program is a card game in which a human player competes with a computer player. Each player is dealt cards from the deck, holds a secret hand of cards, and requests cards from the other player. Once we annotated the program, the inferred security policies show that the implementation of the computer player does not cheat: the computer player's decision of which cards to request does not depend on the human's cards, or the undealt part of the deck.

This program required the most annotations of all case study programs. A key use of annotations indicates when information is declassified and "relabels" the declassified data. For example, the code below shows the relabeling of a card drawn from the deck.

```
computerPlayer.receiveCard(
    @input "Computer drawn card" deck.drawOne());
```

The expression deck.drawOne() has a security policy that indicates it contains information about the undrawn portion of the deck. Relabeling loses this information, and is thus a potentially dangerous annotation, possibly resulting in misleading (but semantically correct) security policies. The benefit of relabeling is that is allows intentionally declassified information be described and tracked separately from the high-security information from which it is derived.

We arrived at the conclusion that relabeling is useful, but dangerous. Outputting the relabeled information mitigates some of the danger, as it means analysis will indicate what information is being relabeled. For example, the following code outputs deck.drawOne() before relabeling it.

```
computerPlayer.receiveCard(
   @input "Computer drawn card"
      @output "Computer drawn card" deck.drawOne());
```

We provide syntactic sugar for this common pattern; and the previous expression can be rewritten as follows.

```
computerPlayer.receiveCard(
    @relabel "Computer drawn card" deck.drawOne());
```

The corresponding inferred policy states,

```
Computer's Choices ↦
  . . . Reveal( Computers drawn cards[0+] . . . )
Computer drawn cards ↦
  . . . Reveal( The Deck[0+] . . . ),
```

indicating that the computer's moves depend on undealt part of the deck *only* via the cards drawn. (For concise exposition we elided other permissible flows.)

> **Design Pattern** Output and relabel declassified information. After intentionally declassifying data, use input annotations to identify the newly declassified information. Output the declassified data first to show flows into the new label.

In Go Fish, both the human and computer player are implemented as subclasses of the same class and share common code. Object sensitivity allows the analysis to distinguish data belonging to these two distinct objects.

B. Password Manager

The Password Manager program implements a simple password wallet that stores passwords protected by a master password. The user interacts with the password manager in a read-eval-print loop; he may issue commands to store and retrieve passwords, and to reset the master password. Passwords are stored encrypted with the master password, and retrieval requires decryption. We use symbolic encryption for debugging purposes in this program.

We added three annotations in total to the Password Manager. All output to the screen occurs through a single method that calls System.out.println. We added a single output annotation to that method, as shown below.

```
public void writeLine(String s) {
  System.out.println( @output "stdout'' s); }
```

We add an input annotation for the user's password input only after we have validated that the input is non-null.

```
String p = env. util .readLine();
if (p == null) throw new CommandFailed();
p = @input "password'' p;
```

Intuitively, if readLine() returns **null** then no password was read and the system contains no new sensitive data. Input annotations of this form constitute a useful design pattern.

> **Design Pattern** Apply annotations to valid information, not to buffers or invalid information.

This design pattern leads to clearer policy maps and more precise inference.

The third annotation is on the result of the method canDecrypt(c, master), which returns true only if ciphertext c can be decrypted with the master password entered by the user. We add an additional **@suppress** annotation to indicate that if canDecrypt returns true, then doDecrypt cannot throw any exceptions.

```
public static String decrypt(String s,
                             String masterpwd) {
  if (@relabel "decrypt ok''  canDecrypt(s,masterpwd)){
    return @suppress Exception
              doDecrypt(s, masterpwd);
  }
  return null;
}
```

The decrypt ok annotation is useful as it both gives a name to the implicit information flow stemming from decryption success or failure, and also allows for inference of a precise conditional policy.

> **Design Pattern** Annotate the decision to release information so that policies describe flows informatively.

With these annotations our tool infers the informative, yet concise policy,

```
output stdout ↦ if  (decryption ok[0]) then
                      Reveal(password[0+])
```

This indicates that the passwords are revealed to the user only if the passwords can be successfully decrypted using the master password entered by the user.

C. Chatbot

The Chatbot converses with the user, sometimes repeating user input in the style of the Eliza[2] program. The inferred security policy indicates that user input is always properly sanitized before being printed as output. The sanitize method is annotated with a **@track** annotation.

```
static String sanitize(String tainted) @track {
    // Strip dangerous characters from tainted
    // return the result.
    return ...; }
```

The Chatbot satisfies this policy,

```
output ↦ if-executed (ChatBot.sanitize(String)) then
              Reveal(input[0+])
```

and is an example of the following design pattern.

> **Design Pattern** Use **@track** annotations on data sanitizers, redaction methods, and trusted declassifiers, to ensure that these methods are involved with all release of sensitive data.

D. Access control

In this example, we built an access-controlled key-value store. Both users and stored values are associated with security levels, and a reference monitor ensures that read requests do not violate security-level constraints. Annotations for this program are broadly similar to those used in the Password Manager, and the analysis infers the following policy map.

```
out ↦ if  (authorizationCheckOk[0]) then
           Reveal(secret[0+], authorizationCheckOk[1+])
```

[2] http://en.wikipedia.org/wiki/ELIZA

This policy map indicates that the release of secret data requires a successful access-control check, ensuring that the reference monitor completely mediates data access.

E. Battleship

Battleship is a two player game, where each player places tokens, representing ships, on his own secret board. The players then take turns guessing where ships are located on their opponent's board. The first player to locate all his opponent's ships wins.

The Jif implementation of Battleship is broadly similar to Go Fish. As with Go Fish, both players have private information—their boards—but unlike Go Fish, there is no card deck causing these secrets to be correlated. We used annotations similar to the Go Fish annotations to show that a computer player did not cheat; its guesses are based on information available according to the rules of Battleship.

More precisely we annotate the guesses of Player 2 (the computer) with **@output** "p2Query" and the secret board of Player 1 (the human) with **@input** "p1Board". Other names (p2QueryOk, p2Hit?, p1HitP2?) are relabelings of legitimate transfers of information from Player 1 to Player 2. Given these annotations, our tool infers the policy

```
output p2Query ↦if (p2QueryOk[0]) then
                   Reveal(p2QueryOk[1+], p2Hit?[0+],
                          p1HitP2?[0+]) } .
```

Critically, Player 2's queries are independent of Player 1's board, p1Board, meaning that the computer is not using information about the other player's board to determine where to guess.

F. Mental Poker

Askarov and Sabelfeld [8] implemented two versions of a mental poker protocol, first in Java, and then in Jif. These programs allow two principals to play a fair game of poker without relying on a trusted third party to manage a deck of cards. All randomness and adherence to the game rules is enforced via cryptographic protocols. We annotated and analyzed their Java version.

Both players generate key pairs to digitally sign messages. We verify that the private key is only used to sign messages and is not disclosed. In the implementation, players communicate via a shared object called the *chain*. We annotated any modification to this shared object with an output annotation. We annotate the public and private keys of each player:

```
KeyPair pair = keyGenerator.generateKeyPair();
this.keyPair = new KeyPair(
                 @input "pubKey" pair.getPublic (),
                 @input "privKey" pair.getPrivate ());
```

We also re-label the use of private key for signatures, indicating that it is a permitted use of the private key:

```
dsa.initSign (@relabel "privKeyForSigning" privKey);
```

The inferred policy for the program indicates that the only sensitive information being communicated to the other player is the use of the private key for signature creation; the private key does not otherwise get revealed.

```
output chain ↦ Reveal(privKeyForSigning[0+])
```

We provided signatures for the cryptographic classes that are used in the program (e.g., java.security.KeyPair, java.security.Signature). Our signatures indicated that there was no information flow from, for example, a signing key to the message digest. This lack of information flow is correct under symbolic models of cryptography (and corresponds to only negligible information flows in computational models). However, in an implementation, the ciphertext is clearly computed from the plaintext, and so our analysis conservatively concludes that the ciphertext reveals (non-negligible) information about the plaintext. Thus, our analysis would not be able to infer the signatures of the cryptographic library from the library implementation.

G. Comparison with Jif policies

Two of the case studies, Battleship and Mental Poker, have equivalent Jif versions, with security annotations from the decentralized label model (DLM) [21]. The security guarantees offered by DLM policies differ significantly from the guarantees offered by our security policies. Our security policies focus on *what* information can be revealed by program execution, whereas DLM policies focus on *who* may learn and release information. For example, in the Jif Battleship program, Player 1's board is annotated as readable only by Player 1, and so releasing information to Player 2 requires an explicit declassification annotation in a code context with Player 1's authority. The Jif annotations do not directly describe what information is declassified.

VI. Related work

In this work, we use static analysis to infer declassification policies that a program satisfies. Our declassification policies specify strong information-flow security, yet are simple and intuitive. Sabelfeld and Myers [7] survey language-based techniques for static reasoning about information-flow security. We focus on two areas of related work: specification of declassification policies and inference of security policies.

Declassification policies. Declassification, or *information release*, occurs when sensitive information is made more public. Declassification violates the semantic security condition of noninterference [10], yet commonly occurs in systems that handle sensitive information. Sabelfeld and Sands [9] survey semantic security conditions for declassification, and categorize them based on: *what* information may be revealed, *when* the information may be revealed, *who* controls the release, and *where* in the system or program the release occurs. Our policies specify *what* information may be released, characterized as expressions whose evaluation an observer may learn. Through conditional policies ($\mathsf{if}\,d\,\mathsf{then}\,p_1\,\mathsf{else}\,p_2$) we can express certain conditions under which release may or may not occur (to wit, conditions determined by program inputs), a form of *when*-declassification. Our policies can also express what code *must* be executed in order for the release to occur: a limited form of *where*-declassification.

The most closely related security condition is *localized delimited release* [22, 23] which uses *escape hatch expressions* to specify what secret information may be released, and requires release to occur only at declassify commands. The security condition requires that when an output occurs, the observer can only learn the valuation of escape hatch expressions for which an appropriate declassify command has been executed. This is similar to satisfying the security policy $\mathsf{if\text{-}executed}\ \texttt{release-e}\ \mathsf{then}\ \mathsf{Reveal}(e)$, where release-e is a mark command that occurs immediately before any declassification of e.

Rocha et al. [24] use graphs to describe how output values are allowed to depend on inputs. Like us, they seek to ease the annotation burden on the programmer; whereas we focus on policy inference, they focus on allowing the user to specify the graph policy separately from the code, and then to verify whether unannotated code satisfies the policy.

Sabelfeld and Sands [9] present prudent principles for declassification security conditions. We satisfy *semantic consistency*, *conservativity*, and *non-occlusion*; the principle of *monotonicity of release* is not applicable, as our programs have no declassification annotations.

Inference of security policies. Backes et al. [25] present an analysis that automatically discovers an equivalence relation characterizing the secret information a program may reveal. They quantify a program's information flow by computing sizes of the equivalence classes. While potentially very precise, such quantitative policies may be difficult to interpret; in contrast our policies are qualitative and we have emphasized readability and intuition in their design.

Banerjee et al. [26] suggest model checking to determine if programs satisfy declassification policies expressed as abstraction functions; counter-examples produced by the model checker allow the declassification policy to be refined, and the process repeated, to determine of the least amount of information that a program declassifies. As with Backes et al., we believe the key difference is that our declassification policies represent a better trade-off between precision, intuitiveness, and ease of inference.

King et al. [16] statically infer information flows to investigate the precision of security-type checking with respect to implicit information flows. Although they perform a context-sensitive analysis, they infer flow-insensitive types: within a given context, a variable is assumed to always contain information at the same security level. By contrast, we infer finer-grained policies, similar to flow-sensitive security types [27].

Pottier and Conchon [28] describe how to extend existing type systems with information security, enabling standard type inference algorithms to infer security types. However, such algorithms are unable to take full advantage of our precise security policies. For example, conditional polices incorporate path-sensitive information that is unavailable in standard typing disciplines.

Liu and Milanova [29, 30] and Livshits et al. [31] present static analyses to infer explicit information flows. Although the analyses are efficient and practical, they do not track implicit flows, and so it is unclear what security guarantees they provide. For example, an analysis of the password checking program from the Introduction that ignores implicit flows may conclude that the attacker learns nothing about the secret password.

Smith and Thober [32] perform type-inference for a highly polymorphic object-oriented security-type system. Top-level security policies are specified independently of code, and inferred types are used to determine whether the program is secure, given the policy. Like us, they seek to reduce the programmer burden. Although they simplify policy enforcement, the programmer must explicitly state security policies. By contrast, we aim to infer policies, which further reduces programmer burden.

King et al. [33] present a model for *information-flow blame* that aids identification of code that violates security. Information-flow blame helps the programmer understand information flows in a program. However, it requires that program types are annotated with security policies, and it cannot infer policies or provide security guarantees if the program fails to type check.

Tschantz and Wing [34] develop a tool that analyzes C-language programs and infers *incident-insensitive noninterference* policies, which allow the existence of high-security data, but not the data itself, to be revealed. Their tool discovers a set of program traces that violate incident-insensitive noninterference, and works over a subset of C.

VII. Conclusion and future work

This work demonstrates that is possible to infer precise and expressive information-flow policies for Java programs. Key contributions include defining an expressive policy language that is suited for precise inference, defining an dataflow inference algorithm, and implementing the analysis for Java.

Several avenues for further research remain open.

It can be difficult to understand how security annotations affect inferred policies and and how non-local control flow (e.g., exceptions) lead to implicit flows. One possible way to mitigate these difficulties would be to be build more sophisticated user interfaces for the analysis tools.

Policy inference as described in this paper relies on expensive program analyses. Scalability may be improved by moving to a flow insensitive analysis. Fortunately, artifacts developed in this work provide a promising testbed for evaluating scalable techniques.

ACKNOWLEDGMENTS

We thank Aslan Askarov and Andrei Sabelfeld for sharing the mental poker source code. We thank our reviewers and shepherd for useful feedback. This research is sponsored by the Air Force Research Laboratory. This research is supported by the National Science Foundation under Grant No. 1054172.

REFERENCES

[1] A. Askarov and A. Sabelfeld, "Localized delimited release: combining the what and where dimensions of information release," in *Proc. 2007 Workshop on Programming Languages and Analysis for Security*. New York, NY, USA: ACM Press, 2007.

[2] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *CSFW '06*. IEEE, 2006.

[3] A. C. Myers, A. Sabelfeld, and S. Zdancewic, "Enforcing robust declassification," in *Proc. Computer Security Foundations Workshop*, 2004.

[4] R. Giacobazzi and I. Mastroeni, "Abstract non-interference: parameterizing non-interference by abstract interpretation," in *POPL '04*. New York, NY, USA: ACM, 2004.

[5] D. Clark, S. Hunt, and P. Malacaria, "Quantified interference for a while language," *Electronic Notes in Theoretical Computer Science*, vol. 112, Jan. 2005.

[6] S. Chong and A. C. Myers, "End-to-end enforcement of erasure and declassification," in *CSF '08*. IEEE, 2008.

[7] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, Jan. 2003.

[8] A. Askarov and A. Sabelfeld, "Security-typed languages for implementation of cryptographic protocols: A case study," in *ESORICS '05*, 2005.

[9] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *CSFW '05*. IEEE, 2005.

[10] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symposium on Security and Privacy*. IEEE Computer Society, Apr. 1982.

[11] M. R. Clarkson and F. B. Schneider, "Hyperproperties," in *Proc. 21st IEEE Computer Security Foundations Symposium*. IEEE Computer Society, Jul. 2008.

[12] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, Jul. 1977.

[13] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif: Java information flow," 2001–2009, software release. Located at http://www.cs.cornell.edu/jif.

[14] N. Nystrom, M. Clarkson, and A. C. Myers, "Polyglot: An extensible compiler framework for Java," in *Compiler Construction, 12th International Conference, CC 2003*, ser. Lecture Notes in Computer Science, G. Hedin, Ed., no. 2622. Warsaw, Poland: Springer-Verlag, Apr. 2003.

[15] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 1, 2005.

[16] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *Proc. International Conference on Information Systems Security (ICISS)*, ser. LNCS, vol. 5352, Dec. 2008.

[17] K. D. Cooper, T. J. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," *Software Practice & Experience*, vol. 4, 2001.

[18] B. De Sutter, L. Van Put, and K. De Bosschere, "A practical interprocedural dominance algorithm," *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 4, 2007.

[19] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *POPL '99*. New York, NY, USA: ACM Press, 1999.

[20] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2011, pp. 17–30.

[21] A. C. Myers and B. Liskov, "Complete, safe information flow with decentralized labels," in *Proc. IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 1998.

[22] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *Proc. IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2007.

[23] ——, "Tight enforcement of information-release policies for dynamic languages," in *Proc. 22nd IEEE Computer Security Foundations Workshop*, 2009.

[24] B. P. S. Rocha, S. Bandhakavi, J. d. Hartog, W. H. Winsborough, and S. Etalle, "Towards static flow-based declassification for legacy and untrusted programs," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2010.

[25] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *Proc. 2009 30th IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2009.

[26] A. Banerjee, R. Giacobazzi, and I. Mastroeni, "What you lose is what you leak: Information leakage in declassification policies," *Electronic Notes in Theoretical Computer Science*, vol. 173, 2007.

[27] S. Hunt and D. Sands, "On flow-sensitive security types," in *Conference Record of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages*. New York, NY, USA: ACM Press, Jan. 2006.

[28] F. Pottier and S. Conchon, "Information flow inference for free," in *Proc. 5th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM Press, 2000.

[29] Y. Liu and A. Milanova, "Static analysis for inference of explicit information flow," in *Proc. 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. New York, NY, USA: ACM, 2008.

[30] ——, "Practical static analysis for inference of security-related program properties," in *Proc. IEEE 17th International Conference on Program Comprehension*, May 2009.

[31] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," in *PLDI '09*, 2009.

[32] S. F. Smith and M. Thober, "Improving usability of information flow security in Java," in *Proc. 2007 Workshop on Programming Languages and Analysis for Security*. New York, NY, USA: ACM Press, 2007.

[33] D. King, T. Jaeger, S. Jha, and S. A. Seshia, "Effective blame for information-flow violations," in *Proc. 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.

[34] M. C. Tschantz and J. M. Wing, "Extracting conditional confidentiality policies," in *SEFM '08: Proc. 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*. Washington, DC, USA: IEEE Computer Society, 2008.