Effective Blame for Information-Flow Violations

Dave King[†] Trent Jaeger[†] Somesh Jha[*] Sanjit A. Seshia[‡]

[†] The Pennyslvania State University
University Park, PA, 16802
{dhking,tjaeger}@cse.psu.edu

[*] University of Wisconsin
Madison, WI, 53706
jha@cs.wisc.edu

[‡] University of California
Berkeley, CA, 94720
sseshia@eecs.berkeley.edu

ABSTRACT

Programs trusted with secure information should not release that information in ways contrary to system policy. However, when a program contains an illegal flow of information, current information-flow reporting techniques are inadequate for determining the cause of the error. Reasoning about information-flow errors can be difficult, as the flows involved can be quite subtle. We present a general model for *information-flow blame* that can explain the source of such security errors in code. This model is implemented by changing the information-flow verification procedure to: (1) generate supplementary information to reveal otherwise hidden program dependencies; (2) modify the constraint solver to construct a blame dependency graph; and (3) develop an explanation procedure that returns a *complete* and *minimal* error report. Our experiments show that information-flow errors can generally be explained and resolved by viewing only a small fraction of the total code.

1. INTRODUCTION

It is essential for programs, especially those that are entrusted with critical personal information, to enforce security goals. Information-flow security is a key requirement for secure programs; if a program is information-flow secure, then secret information cannot flow to public channels at runtime [9]. *Security-typed languages* can be used to build applications that are information-flow secure [10, 19, 20, 21]. Once the programmer has annotated the security requirements on variables and I/O channels, the compiler verifies that these annotations do not permit an illegal information flow.

Although secure systems have been built using security-typed languages, at present these have only been built from scratch [2, 6, 15]. To scale to real systems, we must be able to retrofit legacy code for security. However, such retrofitting is difficult for two reasons. Because the original code was not necessarily written with information-flow requirements in mind, it will contain a large number of security violations, some of which may correspond to expected system behavior (releasing the result of a password check), and some of which may not (the unintended release of a patient's social security number). These errors can be quite complex, as any runtime program path may cause an error. Program paths can span procedures, and conditionals and exceptions create subtle dependencies between data. In even moderately-sized code, the number of possible program paths means that deducing the source of an error, given just the location of where an error manifested itself, can be daunting.

The *information-flow blame problem* is to identify the possible sources of an error in a program. An effective solution to the blame problem is an error report that is both complete (everything that caused the error is reported) and minimal (what is reported does not contain spurious information). The blame problem has been well-studied for runtime errors [5, 14], typing errors [28], and incorrect value computation [16, 27, 29]. However, current methods for determining causes of information-flow errors are either incomplete and non-minimal [8], or require computationally expensive auxiliary solvers for high precision [13].

In this paper, we present a model for information-flow blame and an algorithm for extracting the core reason for program inconsistency from a constraint solver. We implement our model by extending the Jif information-flow compiler [20] with an interprocedural constraint-based program analysis. We then demonstrate how our analysis performs on a variety of Java programs and briefly describe the kinds of information-flow errors that these programs permit. This paper makes the following contributions:

- We show how to build a *blame dependency graph* of constraints that enables queries to retrieve the slice of the graph that contributes to a violation.
- We show how the dependency graph can be used to generate a *complete* and *minimal* set of constraints that caused an error.
- We expand the Jif compiler to display error explanations generated by Java code.
- We apply our analysis to a variety of programs, showing that the errors reported accurately describe the cause of an illegal information flow. We find that our blame model reduces each error to viewing a small fraction of the program (less than 0.5% of the original source for each benchmark), making error resolution straightforward. Our experience with our analysis suggests that the percentage of constraints needing to be examined remains constant as the size of the program increases.

The remainder of the paper is organized as follows: Sec-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-559593-995-1 ...$5.00.

tion 2 outlines the current blame procedure in security-typed languages and why it is presently inadequate. We then outline our model for determining blame for security errors in programs. The formal background for a constraint solver commonly used for security-typed language extensions is detailed in Section 3. In Section 4, we describe the *blame dependency graph*[1], a data structure that allows tracing the reason why a program contains an illegal flow of information. In Section 5, we outline the modifications we needed to make to the Jif compiler for our analysis. In Section 6, we describe the result of our experiments on several open-source and non-trivial Java programs (> 1000 lines of code). In Section 7, we describe some related work, and we give concluding remarks in Section 8.

2. PROBLEMS WITH CURRENT BLAME

We now outline some current problems in finding and resolving information-flow errors in program code. As an example of a typical information-flow analysis[2], we investigate the security-typed language Jif [20], an extension of JFlow [19]. We first give some background on how Jif verifies the information-flow security of its programs.

A program is *information-flow secure* if high security variables do not affect the values stored in low security variables [9]. Explicit information flows occur when high security data is directly written to low security data, while implicit information flows occur when low security data is otherwise affected by high security data. If `h` is a high security variable and `l` is a low security variable, then the assignment `l` := `h` enables the *explicit* flow of information from `h` to `l`, while the conditional **if** (`h` == 0) `then` `l` := 0 enables the *implicit* flow of information from `h` to `l`.

To determine if a program contains an illegal flow, Jif generates *label constraints* for the statements in the code. An assignment statement `v` := `e` generates the constraint $L_e \leq L_v$, where L_v is the security level of `v`, L_e is the security level of `e`, and $L_e \leq L_v$ is read as "the security level of L_e is less or equal to that of L_v". This constraint requires the security level of the expression `e` to be lower than the security level of the variable `v`. If the security level of `e` is not known (as it may not be explicitly labeled), a variable β_e is introduced and used in place of L_e.

To detect implicit information flows, the compiler keeps track of the security of the *program counter*; the security level of the program counter is the level of information released by executing a particular line in a program. Assignments that occur inside a conditional must be to variables of a level no lower than the program counter. For example, if program variables `i`, `j`, and `k` have security levels α_i, β_j, γ_k respectively, then conditional **if** (`i` == 0) `then` `j` := `k` generates the label constraint $\alpha_i \sqcup \gamma_k \leq \beta_j$: both `i` and `k` must not have a higher security level than `j`. Nested assignments, loops, and exceptions are treated in a similar way: any expression that can affect the control-flow of the program taints the program counter.

[1]While the traditional program dependency graph tracks dataflow in the program, the blame dependency graph stores information from the constraint solver to enable an explanation after a constraint inconsistency is detected.

[2]Most implementations of information-flow checkers behave in the way described in this paper, though their underlying type systems may vary.

```
1  User[{Secret}] user; // information in User is Secret
2  OutputStream[{Public}] out; // output stream writes to Public
3
4  private void handleRetr() {
5      // fileIn is a secret file reader
6      BufferedReader fileIn = user.getReader(msgNum);
7      String currentLine = fileIn.readLine();
8      while (currentLine != null) {
9          this.write(currentLine);
10         currentLine = fileIn.readLine();
11     }
12 }
13
14 private void handleTop() {
15     // whether a message is deleted or not is secret information
16     if (user.getMessage(msgNum).isDeleted())
17         this.write(MESSAGE_NO_SUCH_MESSAGE);
18 }
19
20 private void write(String message) {
21     // writes a message out (Public sink)
22     out.print(message);
23 }
```

Figure 1: Fragment of code from the POP3 processer in the JES Email Server.

Once the label constraints for representing the information flows permitted by the program have been generated, Jif runs a modified implementation of the Rehof-Mogensen constraint solver [22] (described in Section 3.1), generating an assignment of variables to security levels such that each constraint is satisfied. If no such assignment exists, the solver reports the first constraint that it could not satisfy, and then fails. The constraint generation and solving process is together called *label checking*.

2.1 Current Problems With Blame

The primary difficulty with the above approach to information-flow blame is that blaming the first constraint at which an error was detected always blames a *sink* of secure information. Information-flow security annotations to programs fall into two different categories: *sources* or *sinks*. For example, if a field containing a PIN number is labeled as `Secret`, then that field is a source of `Secret` information. If a public output stream is labeled as requiring `Public` data, then it is a sink for `Public` information. A field or variable annotated by a security label is a source, while the variable on the left-hand-size of an assignment or a formal parameter of a method call are sinks. Sinks impose a security requirement on flows through the program: if a sink has security label L, then any source that flows to the sink must have a label L' such that L' is not more restrictive than L (written $L' \leq L$).

Figure 1 contains code from the JES, an open source Java email server[3]. The situation in the code is a common one: a secret source, here the user's mailbox, flows to an public output, here a socket's output stream, in more than one way. By examining the code, we can see that there are two information leaks: the leak of whether or not a message has been deleted (caused by line 16) and leaking the contents of a message (caused by line 9). We would prefer to treat each information flow separately: while information flows can be permitted by adding an explicit declassifier, it would not be good practice to automatically declassify any information leaving the system through the `write` method, as we lose

[3]http://www.ericdaugherty.com/java/mailserver/

track of exactly what information is being leaked by the system.

The approach that the Jif compiler currently takes to information-flow blame is to simply blame the first constraint that failed during label checking. Using that approach for analysis of a Java program, we would highlight the line `out.print(msg)` and indicate that this line might send out secret information. We claim that this error message is unsatisfactory, as it does not highlight the *cause* of the error. In the situation where a client is sending a message `msg` out over a `Public` socket, it may be possible that `Secret` data has affected `msg` (in at least one way), but this error message does not display how.

We claim that for a solver's error message to be satisfactory, it must be *complete*: everything that caused a constraint to fail must be available to the programmer. Moreover, the error message must be minimal: it should not show anything unrelated to a specific failure. Specifically, it should show how each line of the program's code contributes to an illegal flow. In the previous example, we should report either that `message` was raised to contain `Secret` data because of the implicit flow from line 16 to the output, or because of the explicit flow from line 9.

Current analyses for explaining information-flow errors are also unsatisfactory. Program slicing can be used to find and resolve information flows [13, 26], as the presence of a secret source in the backward slice of a public sink implies a possible runtime information-flow violation. However, the only existing exploration that we are aware of to use slicing to find witnesses for illegal information-flows [13] relies on *path conditions*, which are boolean conditions for determining when exactly a given program path is executed [24]. This requires an extra analysis to determine if an information-flow violation occurs. However, our experience programming in security-typed languages suggests that most information-flow violations do not require the precision that path conditions allow. Other analyses for explaining information-flow errors do not track implicit flows [12], do not operate on a full programming language [8], or require an advanced solver to determine the conditions under which a leakage occurs [13].

A final technical difficulty with using Jif for retrofitting existing Java code is that its analysis requires programmers to annotate, for all methods, the security value of each argument and the side effects the method may release. This can lead to labeling conflicts rather than true security errors [6, 15].

2.2 Our Approach

To find information-flow errors in existing Java codebases, we modified the Jif compiler to operate on Java programs; this required only minor changes to each aspect of the label checking procedure. Figure 2 summarizes our approach.

To use our tool, a programmer first gives security annotations on various program elements such as variables and fields. The blame algorithm will then output relevant parts of program paths that witness an information flow violation to the user. We find these program paths by instrumenting the solver contained in the Jif engine with a *blame dependency graph* that records information to determine why a label constraint became unsatisfiable. The constraint set is generated by an *interprocedural label analysis* that allows programmers to only label security-relevant code. The analysis infers the rest of the security annotations for the rest of the program, removing the need for the programmer to label every method with the security type of its arguments. Our experiments show that our blame algorithm, provided with an interprocedural label analysis, can accurately identify illegal information flows in programs.

3. SOLVER BACKGROUND

The first step in building our comprehensive blame model is to modify the constraint solver to extract the constraints that contributed to a failure. In this section, we provide a formal description of the constraint solver used by Jif. This will be necessary to understand the motivation behind the *blame dependency graph*, given in the next section.

3.1 Rehof-Mogensen Constraint Solver

To verify the information-flow security of program code, Jif uses a variant of the Rehof-Mogensen solver, a linear-time constraint solver [22]. The Rehof-Mogensen solver operates in two phases: the first adjusts variables based on constraints with a variable on the right-hand side, while the second makes sure that constraints with a label from the security lattice on the right-hand side are still satisfied. We now give some theoretical background for this constraint solver; we will later revisit how the Jif compiler uses it.

Let P be a partially-ordered set (poset) and F be a finite set of monotone functions $f : P^{a_f} \to P$ where $a_f \geq 1$ is the arity of f. The pair $\Phi = (P, F)$ is called a *monotone function problem* or *MFP*. Given a MFP Φ, the set of Φ-terms (denoted by T_Φ) is given by the following grammar:

$$\tau \;\;=\;\; \alpha \;\mid\; \beta \;\mid\; \cdots \;\mid\; L \mid f(\tau_1, \cdots, \tau_{a_f}) \;,$$

where L ranges over constants in P, $f \in F$, and $\tau, \tau_1, \cdots, \tau_{a_f}$ are Φ-terms. Let the set of all variables, assumed to be a denumerably infinite set, be $\mathcal{V}$. We use lowercase Greek letters as variables: $\alpha, \beta, \gamma, \ldots$. The set of variables occurring in a term τ is denoted by $\mathrm{Vars}(\tau)$. A constraint is of the form $\tau \leq \tau'$, where $\tau, \tau' \in T_\Phi$. A constraint set C is a finite set of constraints over Φ.

Let C be a constraint set. A *valuation* ρ is a function from $\mathcal{V}$ to P. Given a valuation ρ, $\rho(\tau)$ is the value of the term τ under ρ. A valuation ρ satisfies the constraint $\tau \leq \tau'$ iff $\rho(\tau) \leq \rho(\tau')$; we write this $P, \rho \models \tau \leq \tau'$. A valuation $\rho : \mathcal{V} \to P$ satisfies C iff ρ satisfies every constraint in the set C. The set of all valuations that satisfy C, denoted by $sol(C)$, is called the set of all solutions to C. Given a MFP $\Phi = (P, F)$, the decision problem Φ-SAT is defined as follows:

> Given a constraint set C over Φ, determine whether C is satisfiable.

For the rest of the paper, we assume that P is a lattice $\mathcal{L}$ with bottom element $\bot$. The *join* and *meet* operators for $\mathcal{L}$ are denoted by $\sqcup$ and $\sqcap$, respectively. Let $\Phi = (\mathcal{L}, F)$ be a MFP. Let an *atom* be a variable (α or β) or a constant $L \in \mathcal{L}$. A constraint set C over Φ in which every inequality is of the form $\tau \leq A$, with an atom on the right hand side, is called *definite*. A definite set $C = \{\tau_i \leq A_i\}_{i \in I}$ can be written as $C = C_{var} \cup C_{cnst}$ where C_{var} are constraints in C that have a variable on the right-hand-side (rhs) and C_{cnst} are constraints in C that have a constant on the rhs. The algorithm for Φ-SAT where C is a definite set of constraints is shown in Figure 3. A detailed explanation for the algorithm, including proofs of its tractability and completeness,

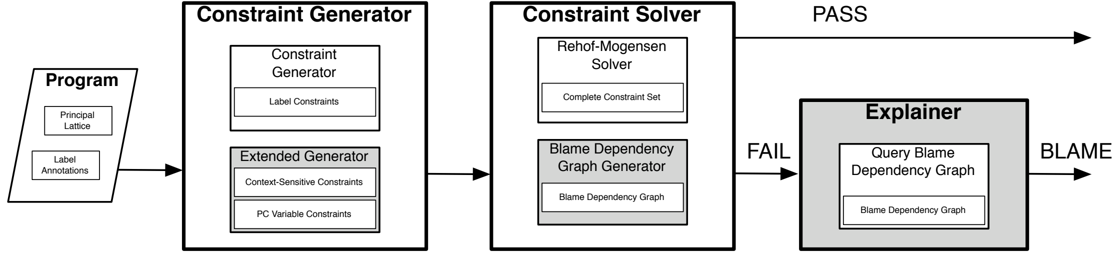

Figure 2: Modified Label Checking Process: (1) a partially-labeled program is input to a *constraint generator* that generates the label constraints from the code using an interprocedural analysis; (2) a *constraint solver* determines if there is a satisfying assignment for the label variables, building a *blame dependency graph* to determine the causes of a constraint failure; (3) if not, the *explainer* generates the security-relevant slice of code that witness security violations by repeatedly querying the blame dependency graph.

RMSolve(C)

1. $\rho(\beta) \leftarrow \bot$ for all $\beta \in \mathcal{V}$
2. $W \leftarrow \{\tau \le \beta \mid \tau \le \beta \in C$ such that $\mathcal{L}, \rho \not\models \tau \le \beta\}$
3. **while** W is non-empty
4. $\quad \tau \le \beta \leftarrow$ Pop(W)
5. $\quad$ **if** $\mathcal{L}, \rho \not\models \tau \le \beta$
6. $\quad\quad \rho(\beta) \leftarrow \rho(\beta) \sqcup \rho(\tau)$
7. $\quad\quad$ **for each** $\tau' \le \alpha \in C$ with $\beta \in \text{Vars}(\tau')$
8. $\quad\quad\quad W \leftarrow$ Push($W, \tau' \le \alpha$)
9. **for each** $\tau \le L \in C$
10. $\quad$ **if** $\mathcal{L}, \rho \not\models \tau \le L$
11. $\quad\quad$ **raise** exception
12. **return** ρ

Figure 3: The Rehof-Mogensen constraint solver

is given by Rehof and Mogensen [22]. Given a fixed run of the solver, a partial valuation ρ_t refers to the t-th valuation; ρ_t is the approximation of the final valuation ρ produced by the solver after the t-th time that line 6 is executed.

In this paper we always assume that Φ-terms τ have the form $\tau \equiv \beta_0 \sqcup \cdots \sqcup \beta_k \sqcup L_1 \sqcup \cdots \sqcup L_j$. As information-flow is a monotonically increasing property, terms that include the meet ($\sqcap$) do not make sense for our analysis. This assumption simplifies the presentation of our blame algorithm in the upcoming section.

3.2 Constraint Solving Example

Let $C = \{L_1 \le \beta_0, L_2 \sqcup \beta_0 \le \beta_1, \beta_0 \sqcup \beta_1 \le \beta_2\}$, where security labels L_1 and L_2 are incomparable. When run on C, the solver will construct a valuation ρ such that $\mathcal{L}, \rho \models \rho(C)$. Initially, $\rho(\beta) = \bot$ for all variables β. The solver first considers $L_1 \le \beta_0$. Because β_0 is currently mapped to $\bot$ under ρ, the solver raises $\rho(\beta_0)$ to L_1. Next, the solver considers $L_2 \sqcup \beta_0 \le \beta_1$. Because $\rho(\beta_1) = \bot$, the solver modifies $\rho(\beta_1) = L_2 \sqcup \rho(\beta_0) = L_1 \sqcup L_2$. Finally, the solver considers $\beta_0 \sqcup \beta_1 \le \beta_2$; since $\rho(\beta_2) = \bot$, it sets $\rho(\beta_2) = \rho(\beta_0) \sqcup \rho(\beta_1) = L_1 \sqcup L_2$. With this ordering of constraints, the **for** loop in line 7 is not executed as its condition is always false. As there are no constraints of the form $\tau \le L \in C$, the solver succeeds, returning ρ.

Suppose instead the constraint $\beta_0 \sqcup \beta_1 \le \beta_2$ was considered first. Because initially all variables are mapped to $\bot$, the solver does not modify β_2. However, after either β_0 or β_1 was modified, $\beta_0 \sqcup \beta_1 \le \beta_2$ would be added to the worklist W by line 8 and so β_2 would eventually be raised to $L_1 \sqcup L_2$.

For the constraint set $C' = C \cup \{\beta_2 \le L_2\}$, the solver initially performs as described above. However, after each constraint of the form $\tau \le \beta$ is considered, the solver attempts to check that $\beta_2 \le L_2$ holds under ρ. However, as $\rho(\beta_2) = L_1 \sqcup L_2$ and $L_1 \sqcup L_2 \not\le L_2$, the solver reports an error, specifically that it has failed on $\beta_2 \le L_2$.

From the constraint set, it is not obvious why β_2 was raised above L_2. We know that $\beta_0 \sqcup \beta_1 \le \beta_2$ modified β_2 (as it is the only constraint with β_2 on the right-hand side), but knowing why β_2 was raised above L_2 involves knowing why either β_0 or β_1 were raised above L_2. In larger constraint sets, determining which constraints caused an error can be even more difficult.

4. BLAME IN THE SOLVER

In this section, we present a structure for assessing blame at the solver level. This is a first step towards the blame model described in Section 2. The *blame dependency graph* is a data structure that acts as a backend for failure explanation in the Rehof-Mogensen solver that can be queried to provide complete error explanations. We then present an algorithm that, for each error, reports a set of constraints X with the property that the constraints in X cause the error (the error reported is *complete*), and each constraint in X contributes to the error (the error reported is *minimal*).

4.1 The Blame Dependency Graph

As described in the previous section, the Rehof-Mogensen solver is used to determine if a Jif program has any illegal flows. In order to assign blame to information-flow errors in Java programs, we construct the blame dependency graph during the run of the Rehof-Mogensen solver. The intuition behind the dependency graph is to record the run of the solver by keeping track of which constraints $\tau \le \beta$ raise the level of a variable β. If the constraint $\beta \le L$ fails, meaning that β was raised too high, the solver can use this information and determine whether or not the particular constraint $\tau \le \beta$ was responsible for this.

4.1.1 Definition

The blame dependency graph (hereafter referred to as the dependency graph) is a directed graph that records the history of the valuation produced by the solver. Let ρ_i be the valuation the i-th time that line 6 in Figure 5 is executed; we refer to this as *time* i. The dependency graph contains each of the ρ_i and information connecting each ρ_i with ρ_{i+1}.

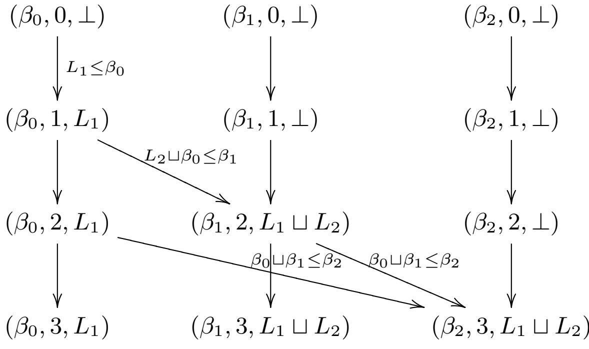

Figure 4: The dependency graph after a solver run

Definition 4.1 (Blame Dependency Graph) *The blame dependency graph $\mathcal{B} = (V, E)$ is a graph that maintains the history of the valuations produced by a run of the solver. The dependency graph has the following properties:*

- *If at time t, the valuation $\rho_t(\alpha) = L$, then the node $(\alpha, t, L) \in V$.*
- *For all t, if $(\alpha, t-1, L), (\alpha, t, L') \in V$, then the edge $((\alpha, t-1, L), (\alpha, t, L')) \in E$.*
- *If α is set to L' at time t because of a constraint $\tau \le \alpha$, then for each $\beta \in \mathrm{Vars}(\tau)$ and node $(\beta, t-1, L_\beta) \in V$, $((\beta, t-1, L), (\alpha, t, L')) \in E$, and this edge is labeled with the constraint $\tau \le \alpha$.*

A visualization of the dependency graph for the solver run over constraint set C from Section 3.2 is given in Figure 4. Initially each of the variables $\beta_0, \beta_1, \beta_2$ begin at $\bot$. The dependency graph indicates that the constraint $L_1 \le \beta_0$ is responsible for raising β_0 from $\bot$ to L_1. Next, it shows that β_1 was raised because of the constraint $L_2 \sqcup \beta_0 \le \beta_1$. Finally, it identifies that the constraint $\beta_0 \sqcup \beta_1 \le \beta_2$ was responsible for raising β_2 to $L_1 \sqcup L_2$.

4.1.2 Using the Blame Dependency Graph

As an abstract data type, the dependency graph supports two operations.

- Record$(t, \beta, L, \tau \le \beta)$ records that the variable β has been modified at time t to the label L because of the constraint $\tau \le \beta$.
- FailureCause$(\tau \le L)$ returns the constraint $\tau' \le \beta$ after which $\tau \le L$ first became unsatisfiable under the valuation ρ constructed by the solver.

The operation Record$(t, \beta, L, \tau \le \beta)$ adds the node (β, t, L) to the dependency graph, and adds an edge for each $\alpha \in \mathrm{Vars}(\tau)$ from $(\alpha, t-1, \rho_{t-1}(\alpha))$ to (β, t, L) labeled with the constraint $\tau \le \beta$. To implement FailureCause$(\tau \le L)$, we trace through the dependency graph to find the time j at which $\rho_j(\tau) \le L$, but $\rho_{j+1}(\tau) \not\le L$. The variable β changed at time j was modified because of a constraint $\tau' \le \beta$. FailureCause returns the constraint $\tau' \le \beta$; at time j, $\rho_j(\tau) \le L$ was true, while afterwards $\rho_{j+1}(\tau) \not\le L$. The constraint $\tau \le L$ fails because the variable β (occurring in τ) was raised to a security level above L by the constraint $\tau' \le \beta$. This gives an explanation for the constraint's failure that can be determined from the blame dependency graph.

4.1.3 Generating the Blame Dependency Graph

To populate the dependency graph during the run of the

RMSolve-DependencyGraph(C)

1. $\rho(\beta) \leftarrow \bot$ for all $\beta \in \mathcal{V}$
2. $W \leftarrow \{\tau \le \beta \mid \tau \le \beta \in C$ such that $\mathcal{L}, \rho \not\models \tau \le \beta\}$
3. $t \leftarrow 0$
4. **while** W is non-empty
5. $\quad \tau \le \beta \leftarrow$ Pop(W)
6. $\quad$ **if** $\mathcal{L}, \rho \not\models \tau \le \beta$
7. $\qquad$ Record$(t, \beta, \rho(\beta) \sqcup \rho(\tau), \tau \le \beta)$
8. $\qquad \rho(\beta) \leftarrow \rho(\beta) \sqcup \rho(\tau)$
9. $\qquad t \leftarrow t + 1$
10. $\qquad$ **for each** $\tau' \le \alpha \in C$ with $\beta \in \mathrm{Vars}(\tau')$
11. $\qquad\quad W \leftarrow$ Push$(W, \tau' \le \alpha)$
12. **for each** $\tau \le L \in C$
13. $\quad$ **if** $\mathcal{L}, \rho \not\models \tau \le L$
14. $\qquad X \leftarrow$ RecursiveExplain$(\rho(\tau) \le L)$
15. $\qquad$ **raise** exception "$\tau \not\le L$ because X"
16. **return** ρ

RecursiveExplain$(\tau \le L)$

1. **if** $\tau \le L$ is unsatisfiable
2. $\quad$ **return** $\emptyset$
3. $\tau' \le \beta \leftarrow$ FailureCause$(\tau \le L)$
4. **return** $\{\tau' \le \beta\} \cup$ RecursiveExplain$(\tau' \le L)$

Figure 5: The Rehof-Mogensen Solver extended with error explanation

RMSolve algorithm, after each execution of line 6 in the original solving algorithm, we record that at time t, β was modified from $\rho(\beta)$ to $\rho(\beta) \sqcup \rho(\tau)$ because of the equation $\tau \le \beta$ and increment the current time. The modified solver algorithm is presented as RMSolve-DependencyGraph, shown in Figure 5.

4.2 Reporting the Cause of an Error

We described a procedure FailureCause that returns, for a single constraint $\tau \le L$, the constraint $\tau' \le \beta$ that caused it first to fail. Because of our assumption that labels contain only joins, this constraint $\tau' \le \beta$ suffices to cause the failure of $\tau \le L$. However, $\tau' \le \beta$ may not provide every reason as to why the constraint $\tau \le L$ failed. For example, τ' may contain variables, making it unclear why τ' itself was raised above L.

The algorithm RecursiveExplain, given in Figure 5, recursively explains why a constraint $\tau \le L$ failed, returning every constraint that contributed to its failure. When $\tau \le L$ fails, RecursiveExplain consults the dependency graph to see why $\tau \le L$ has failed, receiving the answer $\tau' \le \beta$. Since $\tau \le L$ failed because β was raised above $\rho(\tau')$, to determine why τ' was raised that high, the algorithm makes a recursive call to determine which constraints caused $\tau' \le L$ to fail. This continues until the algorithm is called on an unsatisfiable constraint. In Section 4.3, we will see that this procedure can be optimized to run in $O(n)$ time, where n is the number of constraints given to the solver.

To concretely illustrate this procedure, we again revisit the example from Section 3.2 by examining the blame set returned by our explanation algorithm for the unsatisfiable set C', which contains the extra constraint $\beta_2 \le L$. When the solver is run on C', the solver fails attempting to verify $\rho(\beta_2) \le L_2$. Instead of reporting this constraint as the sole cause of the violation, RecursiveExplain searches for the constraint that first caused $\beta_2 \le L_2$ to fail. The first point

at which $\beta_2 \leq L_2$ fails is at time 3, when β_2 was raised to $L_1 \sqcup L_2$. The incoming edges to $(\beta_2, 3, L_1 \sqcup L_2)$ are marked with $\beta_0 \sqcup \beta_1 \leq \beta_2$, so we look for the time where $\beta_0 \sqcup \beta_1 \leq L_2$ first failed. This first failed at time 1, when $L_1 \leq \beta_0$ raised β_0 to L_1. Finally, we attempt to explain $L_1 \leq L_2$, which is unsatisfiable. The algorithm then reports: "$\beta_2 \not\leq L_2$ because $\{\beta_0 \sqcup \beta_1 \leq \beta_2, L_1 \leq \beta_1\}$". These are the constraints that caused $\beta_2 \leq L_2$ to fail.

4.2.1 Broader Explanations

The recursive procedure described above only returns the *first* constraint $\tau' \leq \beta$ that caused $\tau \leq L$. We can also use the dependency graph to perform broader error reporting. If a satisfiable constraint $\tau \leq L$ becomes unsatisfiable under ρ, then for some $\{\beta_0, \ldots, \beta_k\} \in \text{Vars}(\tau')$, we have $\rho(\beta_i) \not\leq L$ for $0 \leq i \leq k$. With the dependency graph, we can answer why each of the constraints $\beta_i \leq L$ failed. This will identify why each variable β_i was raised above level L, giving multiple, possibly redundant, reasons why one constraint $\tau' \leq L$ failed.

4.3 Error Reporting Properties

In this section, we show that RECURSIVEEXPLAIN satisfies a number of important properties. First, we show that a call to RECURSIVEEXPLAIN is guaranteed to terminate. We then show that the sets returned by RECURSIVEEXPLAIN are are complete and minimal, i.e. they actually witness an error and do not contain any smaller explanations.

Appendix **??** contains a proof that finding an error set of *minimum* size for a constraint $\tau \leq L$ is NP-complete. This shows that finding an error set with more global minimality properties is likely to be computationally difficult.

We first show that RECURSIVEEXPLAIN will not loop infinitely. This requires the auxiliary definition of *failure time*, the time (given by the first argument to RECORD) at which a constraint first became unsatisfiable.

Definition 4.2 (Failure Time) *The failure time for the constraint $\tau \leq L$, written $\mathsf{fail}(\tau \leq L)$, is the unique j at which $\rho_j(\tau) \leq L$, but $\rho_{j+1}(\tau) \not\leq L$.*

Lemma 4.3 RECURSIVEEXPLAIN *always terminates.*

PROOF. We show that if RECURSIVEEXPLAIN$(\tau \leq L)$ calls RECURSIVEEXPLAIN$(\tau' \leq L)$, then $\mathsf{fail}(\tau' \leq L) < \mathsf{fail}(\tau \leq L)$. Therefore, the failure time of a constraint returned by FAILURECAUSE will always be strictly decreasing. As the failure time for a constraint cannot be negative, eventually RECURSIVEEXPLAIN will be called on an unsatisfiable constraint and terminate.

Assume $\mathsf{fail}(\tau \leq L) \leq \mathsf{fail}(\tau' \leq L)$; we show a contradiction. Let $j = \mathsf{fail}(\tau \leq L)$, therefore $\rho_j(\tau) \leq L$ and $\rho_{j+1}(\tau) \not\leq L$. Note that $\beta \in \text{Vars}(\tau)$ (otherwise modifying β could not cause τ to fail), and so as τ is a collection of joins, $\rho_j(\beta) \leq L$.

Because $\tau \leq L$ does not fail before $\tau' \leq L$, $\rho_j(\tau') \leq L$. However, $\rho_{j+1} = \rho_j\{\beta \mapsto \rho_j(\tau') \sqcup \rho_j(\beta)\}$, so $\rho_{j+1}(\tau) = \rho_j(\tau[\tau' \sqcup \rho_j(\beta)/\beta])$. Because $\rho_j(\tau) \leq L$, $\rho_j(\tau') \leq L$, $\rho_j(\beta) \leq L$, and since τ is a collection of joins, we therefore have $\rho_{j+1}(\tau) \leq L$, which contradicts j as the first failure time for τ. The recursion is thus well-founded. □

We now show that using our explanation algorithm in conjunction with the dependency graph accurately identifies a cause of an error without displaying useless constraints. We first define what it means for a set of constraints to be *unsatisfiable.*

Definition 4.4 *A set C of constraints is unsatisfiable under security lattice $\mathcal{L}$ if there does not exist a valuation ρ such that $\mathcal{L}, \rho \models C$.*

Of particular interest for explanations of the an error caused by $\tau \leq L$ are sets of constraints X that, together with $\tau \leq L$, are unsatisfiable. In this case X contains every constraint that caused an error. The set X is thus a *complete* explanation for $\tau \leq L$. We use the term *error set* to refer to a complete explanation.

Definition 4.5 (Completeness) *Let C be a set of constraints and $X \subseteq C$. We say X is an error set for a constraint $\tau \leq L$ if $X \cup \{\tau \leq L\}$ is unsatisfiable. If X is an error set for $\tau \leq L$, then we say X is a complete explanation for $\tau \leq L$.*

We are interested in returning *minimal* error sets: these are error sets that do not contain any smaller error set. Minimal error sets are small witnesses to a specific information flow violation.

Definition 4.6 (Minimality) *Let X be an error set for the constraint $\tau \leq L$. We say X is a minimal error set if there is no $X' \subset X$ such that X' is an error set for $\tau \leq L$.*

We now show that the sets returned by RECURSIVEEXPLAIN$(\tau \leq L)$ accurately describe the cause of the failed constraint $\tau \leq L$. For notational convenience, if RECURSIVEEXPLAIN$(\tau \leq L) = X$, we write X as $X_{\tau \leq L}$. We first prove a lemma showing that, if the set returned by the recursive call RECURSIVEEXPLAIN$(\tau' \leq L)$ is an error set, then the set $\{\tau' \leq \beta\} \cup$ RECURSIVEEXPLAIN$(\tau' \leq L)$ is also an error set. This will form the inductive step of Theorem 4.8.

Lemma 4.7 *Let R be an error set for $\tau' \leq L$ and $\beta \in$ $\text{Vars}(\tau)$. Then $R \cup \{\tau' \leq \beta\}$ is an error set for $\tau \leq L$.*

PROOF. Proof by induction on the structure of τ. By the assumption $\beta \in \text{Vars}(\tau)$, we know τ cannot be a constant or a variable distinct from β. If $\tau = \beta$, since $R \cup \{\tau' \leq L\}$ is unsatisfiable, then $R \cup \{\tau' \leq \beta\} \cup \{\beta \leq L\}$ must be also be unsatisfiable; the result follows.

Otherwise $\tau \equiv \tau_1 \sqcup \cdots \sqcup \tau_n$, so $\beta \in \text{Vars}(\tau_i)$ for some τ_i. By induction $R \cup \{\tau' \leq \beta\}$ is an error set for $\tau_i \leq L$. Therefore $R \cup \{\tau' \leq \beta\}$ is an error set for $\tau \leq L$. (if an inequality $\tau \leq L$ is unsatisfiable, then for any τ' the inequality $\tau \sqcup \tau' \leq L$ is also unsatisfiable) □

With Lemma 4.7, we can show that RECURSIVEEXPLAIN$(\tau \leq L)$ returns an error set for $\tau \leq L$.

Theorem 4.8 (Completeness) *$X_{\tau \leq L}$ is an error set for $\tau \leq L$.*

PROOF. Proof by induction on $X_{\tau \leq L}$. If $X_{\tau \leq L} = \emptyset$, then $\tau \leq L$ must be unsatisfiable, and so $\emptyset$ is an error set for $\tau \leq L$.

Otherwise, $X_{\tau \leq L} = \{\tau' \leq \beta\} \cup X_{\tau' \leq L}$, where $\tau' \leq \beta$ is the constraint after which $\tau \leq L$ first failed. By induction, $X_{\tau' \leq L}$ is an error set for $\tau' \leq L$. Let k be the time at which $\tau \leq L$ first failed. Because β was the only variable modified at time k, $\beta \in \text{Vars}(\tau)$. By Lemma 4.7, $X_{\tau' \leq L} \cup \{\tau' \leq \beta\}$ is an error set for $\tau \leq L$. □

Next, we show that the sets returned by RECURSIVEEXPLAIN are minimal; they do not contain any smaller error sets. Minimal error sets for a failed constraint $\tau \le L$ are of great interest for error explanation; since they only contain constraints which caused the error, they are the "best" failure explanation that we can give. We first need a lemma regarding the structure of sets returned by RECURSIVEEXPLAIN($\tau \le L$).

Lemma 4.9 *If* RECURSIVEEXPLAIN($\tau \le L$) $= X$*, then* X *can be ordered as* $\langle \tau_0 \le \alpha_0, \tau_1 \le \alpha_1, \ldots, \tau_n \le \alpha_n \rangle$*, where:*

1. *for* $i < j$, $\mathsf{fail}(\tau_i \le L) < \mathsf{fail}(\tau_j \le L)$
2. *each* α_i *is distinct.*
3. *for* $i + 1 < j$, $\alpha_i \notin \mathrm{Vars}(\tau_j) \cup \mathrm{Vars}(\tau)$.

PROOF. Construct the ordering by sorting the $\tau_i \le \alpha_i$ according to their failure time; by Lemma 4.3, each recursive call has a distinct failure time.

To see that each α_i is distinct, observe that from the previous proof, $\mathsf{fail}(\alpha_i \le L) = j = \mathsf{fail}(\tau_{i+1} \le L)$, and so as the failure time $\mathsf{fail}(\tau_j \le L)$ is unique for all j, each variable also must be unique.

To show $\alpha_i \notin \mathrm{Vars}(\tau_j)$, observe $\alpha_i \in \mathrm{Vars}(\tau_j)$ implies $\mathsf{fail}(\tau_j \le L) \le \mathsf{fail}(\alpha_i \le L) = \mathsf{fail}(\tau_{i+1} \le L)$; therefore, each of the α_i must not occur in any later τ_j. The full statement follows from $\mathsf{fail}(\alpha_n \le L) = \mathsf{fail}(\tau \le L)$. □

In particular, once RECURSIVEEXPLAIN considers a valuation ρ_j, it does not need to consider any valuations ρ_k for $k > j$. Therefore, as mentioned at the end of Section 4.2, RECURSIVEEXPLAIN can be optimized to run in $O(t)$ time, where t is the running time of the solver. As the Rehof-Mogensen solver is a linear-time solver on the number of constraints n, RECURSIVEEXPLAIN can run in $O(n)$ time.

Theorem 4.10 (Minimality) *There is no* $X' \subset X_{\tau \le L}$ *such that* X' *is an error set for* $\tau \le L$.

PROOF. Suppose $X' \subset X_{\tau \le L}$ is an error set for $\tau \le L$. We show a contradiction. First, consider the possibility that $X' = \emptyset$; if so, then $\tau \le L$ is unsatisfiable; however, if this is so, then $X_{\tau \le L} = \emptyset$, contradicting $X' \subset X_{\tau \le L}$. We are left with the case where $X' \ne \emptyset$.

By Lemma 4.9, the error set X can be ordered by failure time as $\langle \tau_0 \le \alpha_0, \tau_1 \le \alpha_1, \ldots, \tau_n \le \beta \rangle$, with each of the variables occurring on the right-hand side of the equation is distinct. Let k be the first $\tau_k \le \alpha_k \in X_{\tau \le L}$ such that $\tau_k \le \alpha_k \notin X'$. For $m > k$, $\mathrm{Vars}(\tau_m) \cap \{\alpha_0, \ldots, \alpha_{k-1}\} = \emptyset$. Therefore, we can satisfy $X' \cup \{\tau \le L\}$ by running the Rehof-Mogensen solver over $\{\tau_0 \le \alpha_0, \ldots, \tau_{k-1} \le \alpha_{k-1}\}$ and leaving $\alpha_{k+1}, \ldots, \alpha_n$ at $\perp$. As τ does not share variables with $\alpha_0, \ldots, \alpha_{n-1}$ and X' is nonempty (so $\tau \le L$ must be satisfiable), the valuation ρ produced from this run will satisfy $\rho(\tau) \le L$. This is a contradiction. □

4.4 Error Traces From Code

Figure 6 and Figure 7 show the error sets returned by our blame algorithm for two of the errors as described in Figure 1 together with the lines of code that generated each constraint. An *error trace* is a subset of the program that witnesses an information-flow violation, similar to a program slice. In the error sets (at the top of each figure), a semicolon (;) represents the join of two labels, `{Secret}` and `{Public}` are explicit security labels, and everything else is a label variable. Variables beginning with `pc` represent the program counter at a specific program point: these are discussed more in Section 5.3. Definitional constraints, specified by contraints of the form $v =_{def} l$, are syntactic sugar for a constraint $l \le v$, where v first appears in this constraint.

```
failed: {message@callto:write:2; pc1} <= {Public}
  1: {message@callto:write:2} == {inst(NO_SUCH_MESSAGE);
                                   write_receiver:2; pc33}
  2: {pc33} ==_{def} {isDeleted:value_returned; pc32}
  3: {pc32} ==_{def} {getMessage:return_observed; pc31}
{getMessage:return_observed} ==_{def} {Secret}
```

```
failed: {message@callto:write:2; pc1} <= {Public}
failure site: out.println(message); [line 22]
  1: if >>(user.getMessage(msgNum).isDeleted())<<
  2: >>user.getMessage(msgNum).isDeleted()<<
  3: >>user.getMessage(msgNum)<<
why:
  {getMessage:return_observed} ==_{def} {Secret: }
```

Figure 6: Error set and error trace for first error from the code in Figure 1. The expression associated with each constraint is indicated offset by >> <<.

```
failed: {message@callto:write:2; pc1} <= {Public}
  1: {message@callto:write:2} == {currentLine; pc22}
  2: {readLine:value_returned; pc18} <= {currentLine}
  3: {readLine:value_returned} ==_{def} {L_var@1782}
  {L_var@1782} == {Secret}
```

```
failed: {message@callto:write:2; pc1} <= {}
failure site: out.println(message); [line 22]
  1: >>this.write(currentLine)<<;
  2: String currentLine = >>fileIn.readLine()<<;
  3: BufferedReader fileIn = >>user.getReader(msgNum)<<;
why:
  {L_var@BufferedReader} == {Secret: }
```

Figure 7: The error set for second error from the code in Figure 1.

For both figures, the failed constraint is the same: however, the reason is different. Our tool first returned the error set associated with Figure 6, indicating the implicit flow from whether a specific user's message was deleted. When we inserted a declassifier to allow this information flow, our analysis returned the error set associated with Figure 6, showing the explicit flow from the `BufferedReader` that operated on the message from the `user`. The variable `L_var@BufferedReader` is the variable representing the security of the `BufferedReader` returned by the `getReader` method.

By looking at each line of the error trace along with the error set, we can see the exact cause of the two reported information flow errors. For the first trace, the write out constraint fails because of an implicit flow from observing the return of the method `getMessage`, a method invocation performed on a `Secret` data structure. This implicit flow affects the value returned by `isDeleted`, which affects whether or not the `write` method is called. The second trace is caused by an explicit flow: the data written out is equal to `currentLine`, which is retrived from data stored in the instance of `BufferedReader` (`L_var@1782`); this is explicitly set equal to `Secret` by the annotation on line 1.

5. BLAME IN PROGRAM CODE

The Rehof-Mogensen solver together with the dependency graph can give explanations for why constraints failed, inde-

pendent of determining errors in Java code. We first show the type of error traces that our tool returns from Java code, and then detail modifications that we needed to make to the Jif constraint generator in order to effectively use our blame algorithm to find security errors in Java code, as described in Section 6.

5.1 Handling Jif Constraints

The information-flow constraints generated by the Jif engine have a different form than Rehof-Mogensen constraints; a constraint c need not be definite, and so the right-hand side of its inequality may be a join of multiple labels [19]. The current implementation of the Jif solver solves constraints of this form through an implementation of the Rehof-Mogensen solver that uses backtracking when it needs to satisfy constraints with more than one component on the right-hand side. This backtracking is done only rarely: most of the constraints that Jif generates are definite. Only one of our code examples required backtracking; the analysis of the JES Email Server attempted to perform backtracking on 21 of its constraints (out of a total of 9539 constraints). None of the other examples required backtracking.

The key property of the Rehof-Mogensen solver that we use in our definitions of the blame dependency graph is its monotonicity: if a variable α is assigned to label L, then α will not later be assigned to a label L' such that $L \not\leq L'$. While the Jif compiler uses an implementation of an extended version of the Rehof-Mogensen solver, it is still monotonic; we can therefore use the blame dependency graph to find errors using the Jif solver. The only modification to the method described in Section 4 is that, when the Jif solver performs a backtracking step, a separate copy of the blame dependency graph is given to each recursive call to the backtracker. In the event that a backtracking call cannot satisfy a constraint without causing another constraint to fail, these copies can be used to provide an error explanation. Our experience has been that backtracking is performed very rarely, and so the overhead associated with this step will be minimal.

5.2 Interprocedural Label Checking

To focus on resolving security conflicts in Java code without annotating every formal argument and side effect bound to each method, we modified the Jif compiler to perform *interprocedural label checking*. Procedures whose arguments are tagged with security labels are checked normally. Otherwise, the constraints for a procedure are inferred using *method summaries*, a standard technique in static analysis [25]. During label checking, we assign each procedure a list of constraints that must be satisfied by the arguments to the procedure for the code to be information-flow secure. Unlabeled methods are assigned *summary variables*, which represent security labels that are not known. If a procedure p has summary variables $v_1, \ldots, v_k$ and summary constraints C_p, then the call of p with actual label arguments $a_1, \ldots, a_k$ generates the summary constraints $C_p[a_1/v_1, \ldots, a_k/v_k]$, where a_1/v_1 represents the substitution of a_1 for all instances of v_1. We can only label-check a procedure if its constraints have no unbound summary variables.

Because procedures may be recursive or mutually recursive, we generate summary constraints for each strongly-connected component of a program's call-graph. Summary constraints for recursive procedures are generated by taking the fixed point of each strongly-connected component. This fixed point will always exist because the number of recursive calls within a strongly-connected component is finite and the join operator is idempotent.

In our experiments, we found that a context-insensitive interprocedural approach worked well on a variety of Java programs. Our analysis reported several false positives arising from context-insensitivity in cases where a program variable was treated as having two different security levels. For example, if a method is used to return both high-security and low-security values, our tool will report an error. We found only a few false positives when examining the program code: we found 2 false positives in JES, 8 in tinySQL, and none in the other three applications that we evaluated.

When we detected a false positive that occurred because of the context insensitivity of our analysis, we were able to resolve it by adding a *label parameter*, a feature of the underlying Jif language, to the enclosing class. A label parameter allows for variables in a class to be given a label based on an immutable label assigned to the enclosing instance of the class; this allows the class to be treated in a context-sensitive fashion. For example, the Java Card Purse (described in Section 6) used a special utility `Decimal` class to perform arithmetic operations. Some instances of `Decimal` contained `Tainted` information (low-integrity), while some contained `Untainted` information (high-integrity)[4]. We parameterized `Decimal` with a label parameter, changing the definition to `Decimal`[**label** `L`], and then annotated each field of `Decimal` as having security level `L`. We then annotated instances of `Decimal` that contained `Untainted` data as `Decimal`[{`Untainted`}]. Our analysis then automatically inferred the labels required for every other instance of `Decimal`. To aid programmers in the process of annotating programs, we are investigating ways to automatically determine which classes and methods may cause context-insensitive false positives.

5.3 Program Counter Variables

The dependency graph can trace the interactions between variables to explain why a label constraint of the form $\tau \leq L$ failed, assuming that such a constraint is satisfiable in the first place. However, under the standard implementation of the program counter, illegal implicit flows can generate unsatisfiable constraints that cannot be explained using the dependency graph.

For example, if the program counter is {`Bob`} before an assignment to an {`Alice`} variable (where `Alice` and `Bob` represent incomparable security levels), then the assignment statement will cause the constraint {`Bob`} $\sqcup \beta_{data} \leq$ {`Alice`} to be generated; here β_{data} is a label variable for expression on the right hand side of an assignment statement. This constraint cannot be satisfied as {`Bob`} and {`Alice`} security levels are incomparable, and so the value of β_{data} is irrelevant. Without a technique to explain why the program counter was set to {`Bob`}, we cannot explain error in a satisfactory way.

To better explain implicit flows with the dependency graph, we introduce temporary variables indicating when the label associated with the program counter has changed,

[4] `Tainted` and `Untainted` form an integrity-dual lattice to the traditional `Secret` and `Public` lattice: `Untainted` information can flow to `Tainted`, but not vice versa.

Application	LOC	Constraint Set Size	Time (s)
Java Card Wallet	296	237	0.60
Mental Poker	1499	6021	5.12
JES Email Server	2595	9539	6.99
Java Card Purse	5581	20924	36.63
tinySQL	8240	23518	102.71

Table 1: A table summarizing the runtime behavior of our analysis. Column 1 contains the name of the application. Column 2 contains the size of the application in source-lines-of-code, (the lines of code without whitespace). Column 3 contains the total number of constraints generated by the program, while Column 4 contains the time for constraint generation.

called *program counter variables.* These variables are a layer of indirection that allow the dependency graph to trace errors across program counter changes. Whenever the program counter is changed from pc to L, a fresh variable α_{pc} is introduced, a definitional equality constraint $\alpha_{pc} == pc \sqcup L$ is created, and the program counter is set to α_{pc}. If a constraint generated by code checked under this program counter fails because α_{pc} is raised too high, there is then a clear path back to where the program counter was set.

This process is similar to converting a program to SSA form [7]. We found that using program counter variables greatly improved our error explanations at the cost of increasing their length.

6. EVALUATION

We evaluated our blame algorithm on several codebases, and show that it aids in the process of finding and resolving errors in Java programs. We compared the source-to-sink error traces returned from our tool to a backwards slice from that sink (comparing with previous work using slicing to discover witnesses for information flow [13]). We used the WALA libraries for program analysis [1] to create program slices. Our experiments demonstrate the following:

- Each of the error sets returned contained a reasonable target site to resolve the illegal information flow.
- The errors returned by our information-flow blame framework were much smaller than backwards slices on the violating sink, corresponding to past use of program slicing for information flow [13].

Our analysis ran on an Intel Core 2 Duo running at 2.20 GHz with 2 GB of memory. Though we ran our tool on a multiprocessor system, our analysis is presently single-threaded. Statistics about the total number of generated constraints and the running time of our analysis are given in Table 1.

The goal of our experiments was to demonstrate that the error sets returned by our tool contained an expression that caused the error (completeness) and were small enough that we could easily find this expression. A complete catalogue of the source-to-sink violations found along with candidate resolutions is available at our website `http://www.cse.psu.edu/~dhking/jlift/`.

6.1 Java Card Wallet

The Java Card Wallet is a small example designed to teach Java programmers how to program in Java Card [5]. It represents a Java Card program that contains a balance that can be credited, debited, or retrieved. To protect the card from unauthorized tampering, the application stores the wallet's PIN in a member field `OwnerPIN pin`. We labeled field `OwnerPIN pin` as having secret data and labeled the `APDU`, a Java Card data structure representing input and output, as having public data. We also labeled the security data revealed by termination of the main `process` method as `Public`.

We found eight total information flows from source to sink, six of which had the same underlying cause. We found three violations: the `APDU` would be written to only if the PIN was validated (two instances), and failing to verify the PIN would throw an exception that was visible to the user. These errors were resolved by adding declassifiers around the calls to checking and verifying the PIN of the Wallet. In many of our other experiments, we often found that one declassifier would resolve multiple errors.

The backwards slices computed by Wallet were particularly small. The Wallet source code represents a best-case scenario for backwards slices: there were only three paths through the program, and each represented an information-flow violation.

6.2 Mental Poker

Mental Poker was one of the original implementations of a non-trivial application in a security-typed language [2]. We ran our analysis tool on a prototype Java implementation provided by the authors. Most classes in the implementation were designed to store either `Secret` data or `Public` data, making manual annotation of the field data in the classes simple. We found 8 separate resolution points in the program, each corresponding with a cryptographic operation associated with the Mental Poker cryptographic protocol.

6.3 Java Card Purse

The PACAP Purse is a prototype designed to secure information flow in Java Card applications [4]. We added integrity labels to its source code: input from the user was marked as `Tainted`, while state kept in the Purse was marked as `Untainted`. Our analysis tool returned 110 errors, as each member of the `Purse` class was marked as being an `Untainted` container, meaning that each code location that modified a field of `Purse` was treated as a separate sink. However, we found that all of the errors could be resolved by adding six declassifiers at places in the code that corresponded to an existing card verification mechanism. These resolution sites were either a call to verify a PIN or a call to an access control table with one of five different arguments, corresponding to the operation that was about to be performed (initialize debit, initialize credit, initialize exchange, initialize PIN verification, or initialize administrative mode).

6.4 JES Email Server

As described in Section 2, JES is an open-source Java email server with full POP3 and SMTP functionality. Its functionality is a superset of JPMail [15], an information-flow secure email server that was manually developed in Jif. To determine how the data stored by the JES server interacted with the user's inputs and outputs, we labeled user data as `Secret` (confidential and high integrity) and labeled

[5] `http://developers.sun.com/mobility/javacard/articles/intro/`

the input and output sockets as containing `Tainted` (public and low integrity) data. `Tainted` data should not be allowed to influence the user configuration file without being sanitized, and `Secret` data should not be released outside the system without being declassified. The security lattice for this example had three labels: `Tainted` and `Secret` (two incomparable levels) and `Public` (a level below both `Tainted` and `Secret`).

We found that five distinct program points created errors in JES, and that these error sites were exclusively either integrity violations or confidentiality violations; no site caused both kinds of violations. In total, we found that we needed to insert 66 (51 confidentiality violations, 15 integrity violations) different resolutions to remove the information-flow errors in the mailbox. Most of the confidentiality violations corresponded to expected behavior of the mail server. For example, the POP implementation of the mail server sent information about the size of a user's mailbox in response to a STAT command. With respect to integrity violations, we found that the application already contained functions to parse and sanitize input; once these were marked appropriately, there were only a few other points in the program that caused an integrity violation. The process of manually inserting these fixes took a few hours.

6.5 tinySQL

tinySQL is a minimal implementation of an SQL client and server on DBF and text files[6]. We marked `System.out` as a `Public` output stream and marked the contents of a SQL table as `Secret`. We found that we needed to manually resolve 30 separate information-flow violations. The major difference between tinySQL and JES was that tinySQL did not separate logging output from normal system output: a debugging flag being enabled caused secret data to be sent to the screen, which we had labeled as `Public`. Using an automatic find/replace, we changed over 60 calls to `System.out` behind debugging conditionals to calls to a logger, and then manually resolved the remainder. We found that the errors corresponded to expected system behavior, with the exception of several unnecessary `System.out` calls in the code revealing data about unnecessary table structure during a query.

6.6 Comparison to Program Slicing

Using slicing to explain information-flow errors has not been well investigated. Previous work by Hammer et al. [13] uses program slicing in order to avoid false positives common to many type-based analyses from such sources as flow and object insensitivity. Hammer et al. determine violating program paths by taking, for each sink at a label L, the set of all sources in the backwards slice of that sink that would cause a violation by flowing to L. To gain greater precision, they use *path conditions* [24] to precisely specify the conditions under which such a leakage occur. However, they do not specify how long path conditions generated by a real program take to solve, and do not provide small witnesses, beyond augmenting a backwards slice with path conditions. The Wallet and Purse Java Card applications that we use were also used by Hammer et al. in their work. The principal difference between our experimental results is that the Jif program analysis is relative imprecise, while Hammer et al. build flow-sensitive and object-sensitive program dependence graphs for greater precision.

Both slicing and type-based analyses (such as those in security-typed language compilers) can be used to find information-flow errors. The principal advantage of our framework is that it provides *complete* and *minimal* explanations of information-flow errors. Another benefit of the approach in this paper is that it requires few modifications to the Rehof-Mogensen solver, meaning that it can be easily integrated with existing verification engines that use the same constraint solver. We believe that slicing technology could be adapted to provide similar explanations.

6.7 Discussion

Table 2 compares the average size of a context insensitive backwards slice (computed using the WALA program analysis libraries), meant to simulate the behavior of past slicing work without path conditions, to the average size of an error trace returned by our system. The statistics given for error set size are the statistics that our tool gave during an inital run of our program, rather than during subsequent runs after fixes were applied. The largest error set sizes encountered during the error resolution process were: Wallet: 13, Purse: 18, JES: 57, Mental Poker: 14, tinySQL: 35.

In most cases there was a high degree of overlap between most error traces, meaning that often the whole error trace did not need to be inspected due to familiarity with past error traces. A common situation was that each error for a `Public` sink included a different reason (often within the same Java class) for `Secret` information tainting that sink, and then a common path back to a `Secret` sink. This corresponds with the expected use of program slices in practice [23].

Due to our recursive approach of blaming the *first* modification that caused a constraint to become unsatisfiable, our tool only reports one error for each constraint associated with a sink. Once that error is resolved, it may report another error for that sink, meaning that there was a different program path that caused a violation. Sometimes a resolution inserted to fix one error fixed many others; this corresponded to the situation where many sinks were hidden behind a common *authorization hook*, such as encryption (Mental Poker) or PIN/password verification (Java Card Purse). To free programmers from having to manually evaluate each error trace, we are working on a system for automatically suggesting candidate fixes.

6.8 Limitations

Our source code analysis has a few limitations; most are common to all static analysis techniques. We must have the source code to analyze a program. In particular, to analyze calls to a library, we must either have the source code for the library or make a simplifying assumption about the security behavior of a library. The Jif language handles library calls by relying on *class signatures*, which give an explicit security labeling for each call to an external method (these can be thought of as a security annotation on header files). To automatically assign a conservative security policy to external library calls, we developed a tool for automatically generating Jif class signatures from Java source[7]. By default, external libraries are treated as containing security data at some immutable level L: all data that enters and leaves the library must be of security level L. This conservative labeling prevents programs from laundering data through mutable data

[6] `http://www.jepstone.net/tinySQL/`

[7] `http://www.cse.psu.edu/~dhking/siggen`.

Application	# Failed Constraints	# Fixes Required	Avg. Size of Error Set (# constraints)	Avg. Size of Error Set (# lines)	Avg. Backwards Slice Size (# bytecode instructions)
Java Card Wallet	8	3	10	9.88	31
Mental Poker	65	8	9.86	6.91	397.69
JES Email Server	5	66	19	13.40	356.20
Java Card Purse	110	6	16.36	15.49	1416.93
tinySQL	120	30	28.61	19.37	312.56

Table 2: A table containing information about how our error traces compare to other tools for determining information-flow errors. Column 1 contains the number of failed constraints in each program (the number that would be reported by Jif). Column 2 contains the number of resolutions, specific to each application, required for an application to be information-flow secure (performed by hand based on our analysis). Column 3 and 4 contains the average size of an error set returned by our tool on the initial run over each program, first in the number of constraints and second in the number of lines of program code. Column 5 contains the average size of a backwards program slice as computed by the WALA library.

structures in libraries. However, these files may still require manual annotation for libraries with system-specific security behavior (for example, all Sockets in an application might be treated as outputting public data). The number and size of the automatically generated signature files scales with the size of application being analyzed. The Wallet Java Card signature files consisted of 150 lines of code (disregarding comments), while the tinySQL signature files required 712 lines of code.

At present, our analysis framework does not handle dataflows as caused by threads or reflection. For the applications that we analyzed that used threads (most notably the JES email server), we replaced calls of the form `new Thread(ThreadClass.class).run()` with explicit calls to `new ThreadClass().run()`: for the purposes of our label analysis, these two calls have identical security behavior. The applications that we surveyed used Java's reflection capacities in a very limited way; the most common method was using the Apache logger `log4j`, which is initialized with a `java.lang.Class` object.

7. RELATED WORK

In this section, we describe some related work in more depth.

7.1 Determining Error Causes

There is a large amount of work towards explaining the cause of type inference errors in ML-like languages. Our work follows a common theme of adding information to the compiler in order to produce more helpful error messages [28]. The major difference between error messages in ML and our work is that a type error in ML may have a syntactic fix, while resolving an information-flow error requires a semantic fix.

Efficient generation of counterexamples and finding the root cause of a counterexample has also been studied by the model checking community. Explaining counterexamples by keeping auxiliary information during fixed-point computations has been investigated in model checking [11]. Finding root-causes of counterexample traces has been used in software verification for abstraction refinement [3].

7.2 Explaining Information-Flow Errors

Hammer *et al.* [13] view detecting illegal information flow as a path traversal problem in the program dependency graph. Errors detected using this method will have clear error messages by observing the path from high information to low information. However, their analysis takes much longer to complete (up to 40 times longer when computing context-sensitive slices) because of a need to compute path-sensitive conditions for conditionals, while their computed relevant slices are much larger.

Deng and Smith have presented a method for security error explanations in a simple language featuring `while` loops and arrays [8]. Instead of generating constraints for information-flows, they use a customized solver algorithm that records the histories of which variables influenced the modification of another variable. When a program fails to type, the history of all of the variables that were involved in determining the type of the broken expression is recursively reported. While this gives good error messages in practice, such explanations may not be minimal. Additionally, as their approach is language-specific, providing error reports for different languages will require expanding and modifying their solver algorithm.

7.3 Type Qualifiers

Our tool has a similar goal to that of CQual, a tool that refines the C type system with qualifiers on types [12], allowing programmers to find inconsistencies between program points. CQual has been used to find bugs in the Linux kernel [17, 30]. A user of CQual annotates code with additional annotations, such as `kernel_ptr` or `user_ptr`, and applies a constraint-based analysis to the source code to determine inconsistencies between the annotations, where, for example, a kernel pointer is inadvertently passed to user code. However, CQual does not handle implicit flows and so can only be used to find explicit information flows; extending CQual to handle implicit flows may be difficult given that type qualifiers are a property of a value, rather than the program counter. In the event of an error, CQual can provide information as to the cause of a type error, which are similar to our error traces. Johnson and Wagner [17] provide heuristics for sorting and pruning error traces before displaying them to the user: we believe our error reporting system could benefit from this technique.

8. CONCLUSION

In this paper, we presented a blame model for finding the cause of information-flow errors in program code. The model extends the Jif compiler by adding an explanation procedure that returns a *complete* and *minimal* error set for a failing constraint. The explanation procedure uses a *blame dependency graph* that tracks relationships between constraints during the run of a solver. This is a general utility for explaining errors generated by the Rehof-Mogensen solver and

can be extended to explain errors in other security-typed languages that also use a similar constraint solver.

Acknowledgements

Lines-of-code data was generated by David A. Wheeler's 'SLOCCount' program. The fourth author was partially supported by NSF grant CNS-0627734. We thank Stephen Chong for answering questions about the internal workings of the Jif compiler, Bill Harris, Kevin Butler, and Swarat Chaudhuri for providing helpful feedback about earlier versions of this document, and the anonymous reviewers for numerous detailed and helpful comments.

9. REFERENCES

[1] T.J. Watson Libraries for Analysis. `http://wala.sourceforge.net`.

[2] Askarov, A., and Sabelfeld, A. Secure implementation of cryptographic protocols: A case study of mutual distrust. In *ESORICS '05*.

[3] Ball, T., Naik, M., and Rajamani, S. K. From symptom to cause: localizing errors in counterexample traces. In *POPL '03*, pp. 97–105.

[4] Bieber, P., Cazin, J., Marouani, A. E., Girard, P., Lanet, J.-L., Wiels, V., and Zanon, G. The PACAP prototype: A tool for detecting Java Card illegal flow. *Java Card Workshop* (2000), 25–37.

[5] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. A static analyzer for large safety-critical software. In *PLDI* (2003).

[6] Chong, S., Vikram, K., and Myers, A. C. Sif: Enforcing confidentiality and integrity in web applications. In *USENIX Security* (2007).

[7] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS 13*, 4 (Oct 1991), 451–490.

[8] Deng, Z., and Smith, G. Type inference and informative error reporting for secure information flow. In *ACM-SE 44* (New York, NY, USA, 2006).

[9] Denning, D. E. A lattice model of secure information flow. *Commun. ACM 19*, 5 (1976), 236–243.

[10] Denning, D. E., and Denning, P. J. Certification of programs for secure information flow. *Commun. ACM 20*, 7 (1977), 504–513.

[11] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *DAC '95* (San Francisco, CA, USA, 1995), pp. 427–432.

[12] Foster, J. S., Fähndrich, M., and Aiken, A. A theory of type qualifiers. In *PLDI* (1999).

[13] Hammer, C., Krinke, J., and Snelting, G. Information flow control for Java based on path conditions in dependence graphs. In *Proceedings of the IEEE International Symposium on Secure Software Engineering* (2006).

[14] Hangal, S., and Lam, M. S. Tracking down software bugs using automatic anomaly detection. In *ICSE* (2002).

[15] Hicks, B., Ahmadizadeh, K., and McDaniel, P. Understanding practical application development in security-typed languages. In *ACSAC '06*, IEEE Computer Society.

[16] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst. 12*, 1 (1990), 26–60.

[17] Johnson, R., and Wagner, D. Finding user/kernel pointer bugs with type inference. In *USENIX* (2004).

[18] King, D., Jaeger, T., Jha, S., and Seshia, S. A. Effective blame for information flow violations. Tech. Rep. NAS-TR-0069-2007 (Updated March 2008), The Pennsylvania State University, 2008.

[19] Myers, A. C. JFlow: Practical mostly-static information flow control. In *POPL '99*, pp. 228–241.

[20] Myers, A. C., Nystrom, N., Zheng, L., and Zdancewic, S. Jif: Java + Information Flow. `http://www.cs.cornell.edu/jif`.

[21] Pottier, F., and Simonet, V. Information flow inference for ML. In *POPL '02*, pp. 319–330.

[22] Rehof, J., and Mogensen, T. A. Tractable constraints in finite semilattices. *Science of Computer Programming 35*, 2–3 (1999), 191–221.

[23] Renieris, M., and Reiss, S. P. Fault localization with nearest neighbor queries. In *ASE* (2003).

[24] Robschink, T., and Snelting, G. Efficient path conditions in dependence graphs. In *ICSE* (2002), pp. 478–488.

[25] Sharir, M., and Pnueli, A. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications* (1981), Prentice Hall, pp. 189–234.

[26] Smith, S. F., and Thober, M. Refactoring programs to secure information flows. In *PLAS* (2006).

[27] Tip, F. A survey of program slicing techniques. *Journal of programming languages 3* (1995), 121–189.

[28] Wand, M. Finding the source of type errors. In *POPL '86* (1986), ACM Press, pp. 38–43.

[29] Weiser, M. Program slicing. In *ICSE* (1981).

[30] Zhang, X., Edwards, A., and Jaeger, T. Using CQUAL for static analysis of authorization hook placement. In *USENIX* (2002).