The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

RETROFITTING PROGRAMS FOR COMPLETE SECURITY MEDIATION

A Dissertation in

Computer Science and Engineering

by

David Holliday King

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2009

The dissertation of David Holliday King was reviewed and approved* by the following:

Trent Jaeger

Associate Professor of Computer Science and Engineering

Dissertation Co-Advisor

Co-Chair of Commitee

John Hannan

Associate Professor of Computer Science and Engineering

Dissertation Co-Advisor

Co-Chair of Commitee

Patrick McDaniel

Associate Professor of Computer Science and Engineering

Martin Furer

Professor of Computer Science and Engineering

Stephen G. Simpson

Professor of Mathematics

Raj Acharya

Professor of Computer Science and Engineering

Head of the Department of Computer Science and Engineering

* Signatures are on file with the Graduate School.

Abstract

Application security is the cause of many vulnerabilities on a modern operating system, making it important for applications to enforce an external security policy. Recent work in language-based security has provided methods for verifying whether or not an application is compliant with a security policy. To use these methods, the programmer provides security labels to program objects and then inserts mediation statements to indicate positions where the security label of a program element might change. However, it is currently not possible to retrofit existing applications to use these guarantees for two reasons: the complexity of information flows and the amount of code that must be manually examined by the programmer to insert mediation statements. In this thesis we provide methods for resolving these problems: an algorithm for explaining information flow errors, an algorithm for automatically suggesting locations in code where mediation statements should be inserted, and a model that enables automatically adding security guarantees to legacy code.

Table of Contents

List of Tables

List of Figures

Acknowledgments

A thesis is the culmination of an individual's work over an entire graduate career, and I am indebted to my advisors and coworkers for their guidance over the last six years. Dr. Hannan and Dr. Jaeger have provided countless hours of assistance and direction for in molding this work into a completed state. Dr. McDaniel interested me in security and taught me to think critically about the research and writing process. The SIIS Laboratory has provided me with a constructive environment in which I could discuss ideas. Boniface Hicks first got me involved with security-typed language research and showing me that language ideas could be successfully applied to the building of secure systems. Coworkers at other institutions have been invaluable, especially the help that Stephen Chong provided in understanding the inner workings of the Jif compiler.

Of course, graduate school does not consist solely of research done towards the completion of the thesis. Jason Woolfrey, Ross Rosemark, Ben Franzen, and Dan Lee helped keep me sane during Penn State football season and provided good company on cold fall mornings. I am indebted to Patrick Traynor for showing me the best backroads and mountains for road cycling in Centre County. I spent many hours in the White Building with Jason Ranville and Matthew DeAngelis while learning how best to cause myself intense pain. The Penn State campus has been much emptier as these people have completed their schoolwork and moved on to their careers.

Throughout this process, my wife Bethany Bray has been extremely supportive and helpful. It would not have been possible to complete this thesis without her assistance.

Chapter 1

Introduction

Security is a major problem facing users and administrators of computer systems. A user expects that the personal data that he enters into a computer will not be released improperly, while an administrator expects that data maintained on his system cannot be improperly retrieved. Both user and administrator are required to trust that an application does not misuse the data it processes.

Trusting an application requires knowledge of the information flows that the application permits: under what circumstances will an application release data? Some information releases should be authorized as necessary for an application to function properly, others should be only be allowed under certain conditions (after a user has been authenticated), while others represent errors in the program.

Work in language-based security has built models to verify automatically that a program's information flows will not violate the security of data. The programmer first specifies a security policy that contains *labels*: unique identifiers corresponding with a security level and a *security lattice*, containing information as to how labels interact and transition with one another. Security labels represent different levels of security and the security lattice gives a semantics to how these labels are allowed to interact with each other.

A *security-enforcing* compiler verifies that a program satisfies that policy: specifically, the program's runtime behavior will not allow a flow between program elements that are associated with incomparable labels in the security lattice. To control the kinds of information flows that an application allows at runtime, we focus on enforcing *information-flow security*. To enforce this property, the programmer gives annotations to the code detailing the labels associated with program elements that correspond to security-enabled input and output channels. Afterwards the compiler verifies that, when run, information is not able to flow between program elements with incomparable security labels. We investigate information-flow security because it is capable of defining strong security guarantees, such as preventing all means of program data leakage.

To verify that a program does not cause a policy violation when run, a information-flow enforcing compiler checks each information flow that the program permits between program elements. If a flow is required for the program to function as normal, the programmer must explicitly allow it with a *mediation statement*. A mediation statement is a notification to the compiler that a certain information flow should be statically authorized. A mediation statement may also be associated with some runtime semantics; for example, a policy check that a user is allowed to access a security-sensitive object. Mediation statements include filters to upgrade low integrity data after sanitization, declassifications to release secret information, and runtime label checks to verify that security labels established at execution are compatible with one another.

It can be difficult to write programs that satisfy information-flow security. Any control-flow element can propagate an information flow, meaning the source of a security error is often quite subtle; compounding this, there is currently no good mechanism for

explaining violations detected during compilation. Even with perfect knowledge of what caused each information flow violation, the most applications contain a massive number of information flows, requiring a programmer to sort through each possible information flow to determine how best to resolve it. Automating the mediation process would allow programmers to easily verify that their programs enforced application security.

We claim that a natural candidate for placing mediation statements is a set of expressions that *cut* a program's information flows. This cut contains a set of expressions that, if removed, completely disconnects the information flows between incompatible security levels in the program. Cuts can be computed using known graph algorithms [24], and so a set of mediation statements can be automatically suggested. A programmer could then examine the set of suggestions and incorporate them into the source code of the application. In the event that a program enables an information flow that should never be allowed, the suggested set of mediation statements would contain a mediation statement for this flow, allowing the programmer to rewrite the program to correct it.

Even with an automatic method to assist the placement of mediation statements, the programmer must adapt this mechanism to the specific security policy and application being retrofitted. We define *placement requirements* as restrictions on the automatic suggestion of mediation statement locations, and give three primary categories for placement requirements. *Functional requirements* concern how an inserted mediation statements will affect the operation of the program, *security requirements* place restrictions on location because of the security policy, and *programmer requirements* are application-specific constraints specified by the programmer. The programmer can use these requirements to select locations for mediation statements in a program and assist

building programs that satisfy the reference monitor guarantees [3]: complete mediation, verifiability, and tamperproofness.

We make the following contributions in this thesis:

- A general method for explaining information-flow errors in applications (Chapter 4). A programmer can use this method to investigate the causes of an information flow that violates a static security specification. Resolving information flows in legacy programs is a difficult task, requiring the programmer to examine all of the information flows that exist in a original program. Due to the difficulty of perfoming this with existing tools most security-typed applications are written from scratch [48].

  To ensure that a program does not permit any illegal information flows, the programmer can identify each source-to-sink information flow and either insert a mediation statement, or modify the code so as to remove an illegal flow. Our solution is a general error explanation method for the Rehof-Mogensen constraint solver [81], which supports a fundamental class of first-order constraints.

- An automated method for placing mediation statements in programs (Chapter 5). We show how a security-enforcing compiler can transform a program into a set of constraints that satisfies *cut-mediation equivalence*, which states that a cut of the information-flow graph having a certain form has a 1-1 correspondence with a set of mediation statements. We show how this method can be implemented as an extension of Jif [71], an existing security-enforcing compiler.

- A demonstration as to how these techniques can be used on existing programs to enforce fine-grained security policies (Chapter 6). We present a model of how to use the techniques from Chapter 5 to build a security mechanism into an existing legacy program and present performance results from an implementation of our suggestion system. We elaborate this for an X Server implemented in Java, `WeirdX`, presented in Chapter 7.

The remainder of this thesis is structed as follows: Chapter 2 gives background to the problem. Chapter 3 surveys related work to the problem. Chapter 4 provides an error explanation algorithm to help programmers understand information flow errors. Chapter 5 introduces the property of cut-mediation equivalence as the key to inserting automating the insertion of mediation statements and shows how a security-enforcing compiler implement it. Chapter 6 demonstrates how the methods of Chapter 5 works on a number of different applications. Chapter 7 gives a detailed look at using our suggestion model on a Java Server. Chapter 8 concludes and suggests areas of future work.

Chapter 2

Background

In this chapter, we give an in-depth exploration of the challenges that programmers face when attempting to enforce security policy at the program level. Specifically, we show that the task of retrofitting program manually presently requires an undesirable amount of work. We formalize the definitions of security policy and how a program uses security policy. Through the use of example code, we highlight the difficulties that programmers currently face in ensuring that a security policy and the runtime behavior of a program can converge.

Due to the difficulty of manually placing security mediators, our goal is automate this process. Our contribution in this thesis is an automated system that will, when given a program and a security policy for that program, suggest statements that mediate each flow from label l_1 to l_2, for each $l_1 \not\leq l_2$. The programmer can use these suggestions to completely mediate the flows in the program and in this way add security guarantees to legacy code.

2.1 Defining Security Errors in Programs

Applications are relied on to enforce system security policy. Therefore, it is important that, when run, these programs respect the security of the data that they process.

Some applications are manually engineered to respect this data through inserting mediation statements or explicit label modification. However, most applications are given more permissions than required to perform their required tasks, leading to security vulnerabilities. Without a precise model for program security, it is difficult to reason about the possible behavior of the system. In this section we give formal definitions for defining and enforcing security policies in programs, and then show why resolving information flow errors in programs is difficult.

2.1.1 Security Lattices

In order to talk about enforcing security policies at the program level, we must first give some formal definitions. We are principally concerned with the enforcement of a specified security policy that consists of *security labels* that represent distinct security together with relationships between these labels. There are a variety of possible interpretations for a security label. For example, in the Decentralized Label Model (DLM) [70], a label contains information about which users can read and write data, while in type-based systems such as SELinux [93, 92], a label has no inherent semantics other than those assigned to it through its security policy.

The key relationship between labels is whether data of one label is allowed to flow to another label via a read or a write. Mathematically, we model a security policy as a lattice $\mathcal{L}$ that consists of labels $l_1, l_2, \ldots$. If one security label l_1 is allowed to flow to another in l_2, then we write $l_1 \leq l_2$ to indicate that this is an allowed flow in a security lattice; otherwise we write $l_1 \not\leq l_2$.

To verify that a program enforces application security, the programmer tags data with a security label describing the program element's security semantics. A security-enforcing compiler then checks the program's behavior and generates *label constraints* that describe a program's information flows. As an example, if a program contains a write of secret information into a public channel (for a lattice containing `Secret` and `Public` labels), then a security-enforcing compiler would generate the constraint `Secret <= Public` from that line of code. We give the exact language for the first-order constraints that we use in this thesis in Section 2.3.1.

Not every program element has a security semantics associated with it. Program elements with an unknown security behavior are assigned lattice variables, denoted $\alpha, \beta, \ldots$ (from a set of variables $\mathcal{V}$). For example, the security constraint $\alpha \leq l_1$ indicates that information at unknown α level flows to security level l_1. Once a set of label constraints representing the information flows of a program has been generated, a label constraint solver creates a map $\rho : \mathcal{V} \rightarrow \mathcal{L}$ such that for all label constraints $\tau_0 \leq \tau_1$, $\rho(\tau_0) \leq \rho(\tau_1)$ in the lattice $\mathcal{L}$. Such an assignment ρ is said to *satisfy* a set of label constraints. If no ρ exists, then the constraint set is said to be *unsatisfiable*. If the constraint generation process is *sound*, meaning that the information flows reflected by the set of lattice constraints accurately represents the information flows in the program, then an unsatisfiable constraint set implies that the program may perform an information flow when run.

2.1.2 Information-Flow Security

In this thesis we concern ourselves primarily with the security policy of *information-flow security*, or *noninterference* because information-flow security is a very strong security policy that precisely specifies the runtime information-flow behavior of a system. If a program satisfies noninterference then it is not possible to gain information about secret data from observing the program's public outputs.

Most programs do not satisfy noninterference "out of the box": either the program will leak a small amount of information about secret data, or will require mediation statements to statically prove the safety of flows between dynamic labels. Currently, adding this information to allow for static security verification must be performed manually.

However, it is difficult to insert security mediation points manually. Inserting a complete set of security mediation statement requires an understanding that the placed statements resolve every information-flow error in the program, meaning that each of these errors needs to be understood by the programmer. Another issue is determining what security policy that the statements should be checking: which label or labels require mediation at each point in the program? A third issue is knowing whether or not the mediation statements placed in the program are redundant; if so, it can be difficult to determine which mediation statements are required to enforce the policy, confusing programmers attempting a code review.

The main challenge in understanding information-flow errors is that programmers cannot tell which information flows are part of a program's regular operation and which are unintentional the program requirements. As information flow is a property of how

components in the program interact with one another, these errors might ultimately manifest in a different location from the component that originally caused the error. It can also be quite subtle to a programmer not comfortable with information-flow errors to diagnose the reason why an error has occurred.

Let l_1 and l_2 be security labels. If $l_1 \not\leq l_2$, then information-flow security prevents data at security label l_1 from influencing data at security label l_2. This prevents both explicit flows of information (for example, data at level l_1 is output on a channel at level l_2) and implicit flows of information (for example, if data at level l_1 satisfies a certain predicate then the program makes an output on a channel at level l_2). Many standard control-flow techniques cause and propagate implicit flows, thereby making it difficult to easily see these errors without a trained eye.

Once a programmer is aware of the information-flow errors in a program, there are many considerations that must be addressed while resolving them. First, most programs have a large number of information-flow errors relative to the number of mediation statements required to resolve them (Table 6.2 in Chapter 6). Therefore, sorting through error reports to determine the best resolution for each of them can be quite cumbersome. It is often not clear when two error reports can be resolved by inserting a single mediation statement. Secondly, there are a number of placement considerations that need to be taken into consideration when placing mediation statements. Some of these constraints are programmer-suggested: for example, the programmer might not want to place a mediation statement within a particular method or file not typically associated with security. There are other placement considerations. Access control checks that check whether a subject can access an object to form an operation: to mediate this, all of the

program elements associated with these entities must be in scope. Also, as querying the security policy has a runtime cost associated with it, a mediation statement placed in a loop can seriously affect system performance, especially in frequently-executed sections of code.

2.2 Challenges in Understanding and Resolving Information Flows

To show the difficulties that programmers face when attempting to convert legacy code into enforcing a security policy, we present code for MASTERMIND. MASTERMIND is a game played between two players, the codebreaker and the codemaker: the codebreaker guards a code and reveals information about it to the codebreaker. It is important that the codemaker not reveal any more information than is necessary to the codebreaker.

Before the game begins, the codemaker sets the code, a sequence of colors. During the game, the codebreaker repeatedly guesses a sequence of colors; to each guess, the codemaker responds with how close the guess is to the code. For example, if the code is *Red, Blue, Green, Red* and the codebreaker guesses *Blue, Blue, Yellow, Green*, then codebreaker gives the feedback: "one color is in the right position, and one color is correct but in the wrong position". Psuedocode implementing MASTERMIND, is given in Figure 2.1.

In Figure 2.1, data corresponding to program elements with a security semantics, such as the codemaker's `code` array, has had security labels added to it Data belonging to the codebreaker is labeled `Public` while data belonging to the codemaker is labeled `Secret`. This corresponds to an underlying information-flow security policy: information

that is `Secret` should not flow to `Public`. The security lattice used for this program is $\mathcal{L} = \{\texttt{Public}, \texttt{Secret}\}$ with $\leq_{\mathcal{L}} = \{(\texttt{Public}, \texttt{Secret})\}$.

The Mastermind program compares the `guess` array (provided by the user) against the `code` array (the secret to be protected). If the ith element of the array is the same as the ith element of the code, then the value `CORRECT_COLOR_AND_POS` is added to the returned array. Otherwise, the program searches through the rest of the array to determine whether or not the guessed color is occurs somewhere else in the code. If so, `CORRECT_COLOR` is added to the returned array. After the program finishes iterating over the `guess` array, it returns a pair containing the number of exact color and position matches.

Mastermind maintains a secret (the `code` array) that is released in a controlled way to a public user (comparison of the `guess` to the `code`). Information flows from the `code` result to the return array `returnArray` lines 25, but the `guessArray` variable is set at lines 14 and 20. There is one security-relevant source in this program: the array `code` is labeled as containing elements at security-level `Secret`. The security-relevant sink is the return statement at line 29. Information-flow errors occur when data flows from the security relevant source to a security relevant sink. Here there would be a security error when `returnArray` is returned from the method, as the method has been given the return label of `Public` but the array contains data that is `Secret`.

MASTERMIND contains examples of both explicit and implicit flows of information. There is one explicit flow of information: `returnArray` is initialized to be a new array with the same length as the secret `code` array. There are two implicit flows in the MASTERMIND code that correspond to assignments of `guessResult` at lines 14 and 14.

```
1  int[]{Secret} code;    // Bob's secret code
2  final int CORRECT_COLOR, CORRECT_COLOR_AND_POS;
3
4  public int[]{Public} guess(int[]{Public} guess) {
5     int[] returnArray = null;
6     if (guess.length == code.length) {
7        returnArray = new int[code.length];
8        int[] used = new int[guess.length];
9        int k = 0;
10       for(int j = 0; j < code.length; ++j) {
11          int guessResult = -1;
12          if (guess[j] == code[j]) {
13             used[j] = 1;
14             guessResult = CORRECT_COLOR_AND_POS;
15          }
16          else for(int i = 0; i < code.length; ++i)
17                if (guess[j] == code[i] &&
18                      used[i] != 1) {
19                   used[i] = 1;
20                   guessResult = CORRECT_COLOR;
21                   break;
22                }
23          }
24          if (guessResult != -1)
25             returnArray[k++] = guessResult;
26       }
27       sort(returnArray);
28    }
29    return returnArray;
30 }
```

Fig. 2.1. Mastermind Code Example

These are implicit flows because their assignment only occurs if conditional statements involving the `code` array are satisfied, at lines 12 and 17. There is a two additional implicit flows from `code` to `returnArray`, as the `returnArray` is modified only if the `guess` array has the same size as the `code` array: `returnArray` is only modified if the `guess` has the same length as `code`, and `returnArray` is modified any number of time up to `code.length` by the for loop at line 10. While several of these program elements leak the same secret information, at present the programmer must determine and mediate each of these information flows separately.

Complete mediation requires that each of these source-sink data flows is properly mediated. In Mastermind, It would be simple for the programmer to simply mediate the contents of `returnArray`. However, while this approach to mediator placement provides complete mediation, it provides no insight as to what is actually being mediated. This is because the values stored in the array might have been influenced by any statement on the execution path from the source and sink. A programmer placing mediation points in the Mastermind program would likely select lines of code close to where elements of `returnArray` have been assigned, thus making it easier for a casual inspection to reveal what information is actually being mediated.

In a large program, finding exactly what sources have influenced data being sent out on a sink can be difficult. As information-flow security is not typically a program property that programmers are aware of, the sheer number of ways that information can be propagated between variables makes this is a difficult task. Even if a programmer can quickly understand what causes program flows, any runtime path from source to sink

might have contributed to an information-flow violation, meaning that a programmer will have to investigate a large percentage of the program to identify these paths.

2.3 Security Enforcement

In this section we overview how current security-enforcing compilers operate. We survey how programmers specify security policies and how security-enforcing compilers determine information flow in a program.

2.3.1 Label Constraint Formalism

We now give the formal definition for the first-order label constraints that we use through this thesis. Let P be a partially-ordered set (poset) and F be a finite set of monotone functions $f : P^{a_f} \rightarrow P$ 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 (security labels) 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 denumerable infinite set, be $\mathcal{V}$. We use lower-case 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:

Φ-SAT: Given a constraint set C over Φ, determine whether C is satisfiable.

For the rest of the thesis, 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* A 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 right-hand size of the constraint. The label constraints generated by the static analyses considered in this thesis will all be definite label constraints. The Φ-SAT problem for sets of definite constraints has a polynomial-time solution [81].

If a satisfying mapping ρ exists, then there is definitely no illegal information flow when the program is run. Otherwise, there might be an illegal information flow in the program. We will study how to explain and resolve information-flow errors in Chapter 4.

2.3.2 Static Analysis

We use *static analysis* to enforce security properties on programs. A static analysis determines facts about the runtime behavior of a program from the source code. The major advantage of using static analysis is that facts about every possible run of the program can be determined. In comparison, a *dynamic analysis* that monitors the program can only determine facts related to the current run of the program. When performing a security analysis it is beneficial to know that all execution paths of a program are safe before a program is granted special trusted status by an operating system. We mention dynamic analyses for enforcing program security in the related work.

Static analyses used for information flow enforcement read the program's source code and generate label constraints that represent the information-flow behavior of the program. A *constraint solver* then attempts to generate a mapping ρ that satisfies the label constraints. The static analyses that we work with in this thesis are *sound* analyses, meaning that the analyses will not produce a false negative where the analysis does not produce a warning about a flow that actually could exist when the program is run.

2.3.3 Limitations of Static Analysis

We use a type-based static analysis because type-based analyses can be implemented in a fast and efficient manner, at the cost of simplifying the analysis. The main drawback to static analysis is the presence of false positives. Whether a particular data flow happens when a program is run is undecidable in the general case, and so static analyses for information-flow must only approximate a program's runtime behavior. A

false positive for a security-enforcing compiler occurs when this approximation contains an illegal flow that cannot occur when the program is executed.

As an example, one of the key factors in the behavior of an analysis is how it handles interprocedural calls. Because most programs contain many such calls, treating each call with a separate security semantics can require a large amount of memory. For this reason, it is common for programs to restrict procedure calls. A basic method for interprocedural analysis is to treat procedure calls as *context-insensitive*: all calls to a method m as having the same security behavior. This can cause erroneous reports should a method process data of differing security levels: for example when a program is run, one instance of a call to m might return high security data, while another might return low security data. If all calls to m are treated as having the same security semantics, a sound security analysis would treat m as always returning high security data. This could cause a false positive at the procedure call that returned low-security data if the value returned from m was passed to a public sink.

To reduce the number of false positives, programmers can change the context-sensitivity (procedure calls), path-sensitivity (branches), and object-sensitivity (objects) of their analysis. These analyses reduce the number of false positives reported, but come with a corresponding increase in performance cost and rely on algorithms with high computational and memory overheads. Many algorithms for label flow [79, 96, 98] exist that can produce a more faithful set of label constraints by adding various degrees of sensitivity to methods, objects, and classes.

Certain classes of information flow errors are more susceptible to false positives than others. A study of several programs used for security reveals that over 90% of all

information flow alarms resulted from program flows that occurred because of runtime exceptions, and of those, 79% of them were false positives: the violating flows could not occur at runtime [57]. Many of these information flows correspond to the imprecision of null-pointer analysis (53% of total exceptions, 95% false positive rate). While the methods developed in this thesis still apply to resolve information-flow errors arising from exceptions, without a better method for filtering out false positives we focus on flows that do not arise from exceptions.

2.3.4 Implementing Mediation Statements

Once the programmer has identified the program sites that contribute to an information flow error, it is still necessary to implement a mediation statement that resolves this security violation. However, implementation of mediation statements is usually often policy-specific for reasons that we will describe. If two policies have identical *placement constraints*, then the same automatically selected positions can be used for both policies.

An implemented mediation statement is a piece of code that can perform a policy action: for example, a secrecy policy with `Secret` and `Public` labels might require a declassification statement to mediate a flow from `Secret` to `Public`.

We expect that for most nontrivial policies, mediation statement implementation will be policy specific. The implementation of a declassification statement in one policy may be to encrypt the data, while another might apply a content filter to remove sensitive information. Implementations of an endorsement statement for a flow of information between integrity levels will differ based on how that data is used. For example, a

filter for query strings passed to a database will differ based on the format of the query language, as each database language has a different syntax and set of "unsafe" symbols.

However, we also expect that for similar classes of policy, mediation statement placement will remain the same. Let a *placement constraint* be a restriction or mandate on where a mediation statement can be placed in the code. For example, the policy may forbid a mediation statement from being placed in code meant to handle tring parsing. Given a mediation placement that satisfies the placement constraints for an application, the programmer can insert a different implementation at each placement point. Although different declassification policies might have different reclassification semantics, a single point in the program can serve to mediate a violating secrecy flow no matter what the policy specifics. This is possible because the information-flow checker treats a mediation statement, irrespective of implementation, as a policy action that modifies the security label on data.

The only mediation placements that require knowledge of the implementation are those policies whose security semantics depend on the placement of mediation statements. For example, consider a policy that checks whether a security sensitive operation is allowed to be executed: mediation statements here should only be placed at program points which are uniquely identified with one operation. For policies with identical mediation placement requirements (*placement constraints*), the same mediation point locations can be used.

The focus of this thesis is on *mediation placement*. This is because we believe that most policies will have only a few different kinds of mediation statements, most of which will not contain placement-specific requirements. If a security policy satisfies

these requirements, the programmer will have to only manually write a few different types of *templates* for implementation statements. These templates can be instantiated at various points in the program based on which kind of label transition is required to allow the information-flow checker to accept the program. For example, there may be several different implementation methods that a compiler is allowed to authorize a label flow $L_1 \rightarrow L_2$: a programmer can choose one of these during the security checker process.

In Chapter 5 and Chapter 6 of this thesis, we present a system for automatically selecting the points at which mediation points should be placed subject to certain placement constraints, but it is still be necessary for the programmer to manually implement a mediation statement that conforms to the policy.

Chapter 3

Related Work

In this chapter we survey related work in information-flow security. Our work aims to build tools for retrofitting legeacy code to enforce complete mediation guarantees. We survey work in two main areas: information-flow security and in operating-system security.

3.1 Information-Flow Security

Denning first investigates the problem of *noninterference* and gives an algorithm for determining whether or not a program contains an illegal flow of information [28, 29]. Smith, Volpano, and Irvine framed noninterference as a type problem [101] and presented a type system that enforces noninterference on a simple imperative language. Followup work expanded the features of the type system, allowing more advanced languages features such as interprocess communication and procedures [66, 85, 94, 102]. This work relies on a definition of noninterference that presumes a *low observer* (an observer that can view all of the low-security information in the system) is monitoring the execution of the program. If the low observer can see a difference between any two execution traces with different assignment of high variables, then the program contains an illegal leak.

JFlow is a type system for Java that enforces information-flow guarantees by adding security types to the static type system [69]. JFlow was extended to provide an

implementation of the Jif security-enforcing compiler [71], a compiler platform to explore extensions and implementations of security-typed languages. Jif combines a static label checking approach with some dynamic label checking to verify certain security properties that cannot be checked at compile time. Since its initial release, Jif has added features for dynamic labels and principals (users). Dynamic labels and principals [100] allow the same source code to operate on security labels that might be unknown at compile time; as this corresponds to most security applications, their inclusion is mandatory for any security-enforcing compiler.

Security-typed languages focus on enforcing all illegal flows of information, both implicit and explicit. Implicit flows are related to the problem of ensuring that a program has no covert channels [61]. In certain execution environments, covert channels can leak a large amount of information, going so far as to reveal enough information to recover public keys [30, 18]. Modern security-enforcing compilers are concerned mostly with programmatic flows and do not attempt to remove timing channels (revealing secret information based on program execution time), termination channels (revealing secret information based on whether the program terminates or not), or power channels (revealing secret information based on how much power is used by the system). Additionally, lightweight static analyses used for security enforcement cannot handle all of the potential covert channels possible in legacy programs, as nearly any expression in a modern programming language might terminate abnormally, possibly because of a runtime exception or a bad pointer dereference [57].

Noninterference is a very restrictive security policy. Because of this, there is work in allowing explicit declassification statements to release information. This work is separated into different dimensions of declassification [86]: "who" can do the declassification, "what" information can be declassified, "when" the declassification can be performed, or "where" the declassification statements can occur in code. We survey some recent work in the field. Delimited information release [85] prevents laundering secure information through declassification statements: this security property can be enforced by not allowing updates to variables that occur in a declassification statement. Gradual release [6] furthers the security guarantee of delimited release: a program satisfies gradual release if at runtime, a low observer learns only the information released by the declassification statement. A conservative approximation of gradual release can be enforced by a type system that restricts declassifications to occur within a low context.

3.1.1 Building Information-Flow Secure Systems

There has been a variety of recent work in integrating security-typed languages with secure operating systems. The main difficulty in this is instantiating application-level labels with system-level labels. Hicks et al. guarantee security policy compliance between SELinux and an application running in Jif is guaranteed by a mapping between the two security lattices [52]. Follow-up work explores these issues more deeply, highlighting the input and output of a system as the point to enforce compatibility through a runtime system for allocating dynamic labels [51]. A separate line of work suggests a general method for policy construction [84], wherein the integrity of a trusted program's data is always considered higher integrity than that of the system it is running on.

A number of papers integrate software engineering themes in order to build systems using security-typed languages. Because of the strictness of information-flow enforcement, programmers deal with many more restrictions when dealing with security-enforcing compilers, leading to many difficulties. Aaskarov and Sabelfeld implemented a distributed cryptographic protocol in the Jif language, Mental Poker, and suggest a number of patterns for programming in security-typed languages [5]. Hicks et al. developed JPMail, a mail client written in Jif, and highlighted the main challenges in using the language from a security perspective [48]. This work identified several principles for compiler and tool development, leading to the development of Jifclipse [49], a platform for program development in security-typed languages. To separate policy from its implementation, Hicks et al. giving each declassification a lexical name [50] so that users can easily specify which declassification statements are allowed to modify the security of their data. Though its main focus is on system design, the Civitas system is the largest program written in a security-typed language, implementing a secure remote voting machine in a version of Jif extended with information erasure policies [22]. Erasure policies ensure that certain sensitive information entering into a system is not stored [19].

3.1.2 Quantifying Information Leaks

One problem with traditional definitions of noninterference is that there is no measure of an information leak's *severity*: a program that infrequently leaks a small amount of information is classified as just as bad as a program that routinely leaks a large amount. Typically information leakage is classified as the number of secret *bits* that are leaked to an observer over a given amount of time. Statically determining

how much a program will leak is a hard problem, and depends on understanding the distribution of the input and the behavior of the program's loops [21]. Recent work approaches this from a probability perspective [64, 65], providing a way to give an upper bound to the number of bits leaked by a program using Shannon entropy [89]. Dynamic instrumentations [67] has been used to bound the number of bits leaked by the program. This approach currently requires variables that can be modified by an implicit flow to annotated as part of an *enclosure region*; this is presently done manually, though it can be automated. When the instrumented program runs, a graph is generated containing the flows that occurred during the run of the program. The maximum-flow of this graph is an upper bound on the amount of information that leaked during the program's run.

3.2 Language-Based Security

There has been a large amount of work in enforcing general security properties of a program through static and dynamic analysis. In this section, we survey some main works.

3.2.1 SQL/XSS Attacks

A casual inspection of recent US-CERT Cyber Security bulletins [*] reveals that SQL injections and cross-site scripting (XSS) are a major class of security vulnerabilities. An *SQL injection* [44] occurs when a program, frequently a PHP script running on a website, performs an SQL query containing unsanitized user input, allowing attackers to retrieve or delete arbitrary information from the database. Cross-site scripting occurs

[*] `http://www.us-cert.gov/cas/bulletins/`

when a call to run a malicious script is inserted into trusted content. Both of these problems are a problem of *integrity*, a security property that can be enforced by performing an information-flow analysis. These taint analyses ignore implicit flows as most SQL injections and XSS attacks are caused by explicit flows (untrusted user input being treated as trusted). A variety of taint analyses exist to determine if a program is vulnerable to these integrity attacks [97, 106, 107]. As both of these problems can be enforced by information-flow analyses, the techniques described in this paper are applicable to building tools to ensure that scripts are free of SQL injections and XSS attacks.

3.2.2 Dynamic Monitoring and Java Security

Many applications support integrity and secrecy requirements through a runtime system. A unifying concept is that of *security automata* [88], wherein a program is monitored by an external agent that makes policy decisions. This requires the program to be instrumented with *hooks* to the external policy monitor. Other approaches generalize this concept [62] and allow runtime policies to be more easily composed [13]. Our work aims to place these policy enforcement hooks into programs such that the program properly enforces the required policy, satisfies complete mediation of illegal data flows, and complies with programmer expectations as to hook placement and system performance.

The main runtime system for policy enforcement in use now is provided with the Java Runtime Environment (JRE). Java "sandboxes" untrusted applications and allows the user to control their interactions with the outside system. Administrators can specify which resources a Java application (or applet) is permitted to have access to [26]. This enforcement is performed at runtime and can cover a wide variety of attacks on the

system. However, it cannot enforce finer grained security policies such as those provided by an information-flow analysis: the JRE does not track data-level information flow, so applications are still vulnerable to taint-based attacks such as SQL injections.

To determine whether or not an applet is permitted to perform an operation, most Java implementations implement some form of *stack inspection* [103]. When attempting to perform a restricted action, the runtime environment checks the current call stack to verify that the runtime stack has not been called by an untrusted applet. Stack inspection does not stop all attacks: for example, it is possible for an untrusted applet to change data such that the application's behavior is modified during a time when the untrusted code is no longer on the stack. Another approach to runtime policy enforcement is to consider elements from the entire history of the program's execution [2, 33, 34, 38], known as *history-based access control* (HBAC). History-based access control can prevent many attacks that stack inspection-based access control would allow, but also restricts the use of untrusted code before an unrelated security-sensitive operation. Information-Based Access Control [77] combines these approaches to ensure that only the code actually required to perform a runtime enforcement is checked for integrity.

Dynamic monitoring of programs for information-flow errors has several problems. If an information-flow leak is detected, then the monitor has two options: to either not allow or allow the execution of the leak. If the leak is allowed but logged, then it is the task of the system administrator to identify whether this leak is a false positive or not. A main problem in systems such as this, like intrusion detection systems, is determining which alarms are true information leaks given the high incidence of false alarms compared to true information-flow errors [8].

3.2.3 Enforcing Other Code Safety Properties

Many other research projects leverage programming language theory to enforce application safety properties. CQual [37] extends C's base type system with *type qualifiers*: the CQUAL compiler can then be used to enforce explicit flow properties between type qualifiers. In one work, researchers annotated kernel pointers with a type qualifier `$kernel`. CQUAL's type system can certify that no user pointer is set equal to a kernel pointer, verifying that the application contains no user-kernel pointer bugs [39]. CCured [73] is a compiler analysis that adds sanity checks to C programs to ensure that the program satisfies memory safety properties. Deputy [23] extends CCured by enforcing a dependent type system [7, 110] on the C language. First-order dependent type systems such as those included in the Deputy compiler add richer types by allowing quantifiers over natural numbers. This allows programmers to verify that their programs satisfy more complicated safety properties; for example, that the program does not perform any out-of-bounds array accesses.

3.2.4 Proof Carrying Code

We focus on verifying the safety of a program at compile time. However, in distributed systems, the primary time of enforcement is at run time. Proof-carrying code [4, 72, 74, 75] allows programs to carry with them a proof of their correctness; however, this requires a certain amount of work from the programmer in specifying verification conditions (VC) that can be checked by a theorem prover. Ideally a program would be paired with a proof of security correctness, certifying that it is safe to run the program on the system. Recent work has outlined the design of an information-flow

certifying compiler [11, 12, 10]: a compiler that generates, with bytecode, a proof of the bytecode's information flow security. The construction of such a compiler would require several additions to exist information-flow compilers.

3.2.5 Aiding Security Annotation

Most security analyses on existing code require some manual annotation by the programmer to associate portions of the program with a security policy. This relies on the programmer to know the code well enough to accurately provide these annotations. However, it is still possible for programmers to provide incorrect labels or forget to label certain parts of the program. The Merlin system recently developed by Livshits et al. [63] infers an information flow specification from an data-flow graph produced by an analysis of the program. At present this analysis is presumed to be done on a graph of the program's explicit information flows.

3.3 Reference Monitors

The reference monitor is a fundamental concept in computer security [3]. A reference monitor is a security policy integrated into a system through a series of access queries. A reference monitor guarantees a program satisfies a security policy by ensuring *complete mediation*, *tamperproofness*, and *verifiability*. It is important that trusted programs satisfy each of these properties. Most security-enforcing compilers guarantee complete mediation. Tamperproofness of a program can be achieved by enforcing integrity guarantees on system resources utilized by the program. Ensuring verifiability of a reference monitor is more difficult, and relies on a proof that the security-enforcing

compiler is actual enforcing the expected security policy. At present this is achieved by proving facts about the underlying type system of the security-enforcing compiler.

Many systems have had reference monitors added to them after their initial implementation. One example of this is the Linux kernel: kernel developers decided, instead of enforcing one security policy, to build the Linux Security Modules (LSM) framework, where policy queries were implemented through *mediation hooks*. Mediation hooks are locations in the program where a security check statement are inserted by the compiler. These hooks were discovered to contain numerous omissions, violating their complete mediation [112]. When selecting security hooks in legacy applications, developers have been primarily concerned with correctness and in limiting the performance impact of security code [56].

Ganapathy et al. explored a line of work in verifying that security hooks were placed properly. The main motivation for this work is in observing that certain security operations always were associated with a given operations performed in a code. To determine whether or not a program is correctly mediating its security sensitive operations, the programmer specifies which data structures are protected. Observing that security-sensitive operations are often correlated with reads and writes of these data structures (called *structure member accesses*), program actions that use these data structures are grouped together into *security-sensitive fingerprints*. Ganapathy et al. showed how these fingerprints, if given by the programmer, could be used to statically verify access hook placement [40]. Further investigations focused on ways to determine for each high-level security-sensitive operation the security-sensitive fingerprint associated with it. One approach observes runtime traces to determine which structure member accesses are

associated with which security-sensitive operation [41]. A second approach uses static analysis to determine which procedures perform structure member accesses, and then uses concept analysis to cluster these together into candidate fingerprints [42]. Candidate fingerprints can then be manually associated with security-sensitive operations.

The line of work investigated by Ganapathy et al. differs from the work presented in this thesis in several ways. First, it is primarily concerned with what kind of operation should be mediated in a mandatory access-control system, whereas the work in this thesis focuses on a more general information-flow policy. Secondly, this work does not provide a solution to the general question of hook *placement*, instead focusing on what policy should be enforced through attempting to discover security-sensitive fingerprints.

3.3.1 Decentralized Information-Flow Control Systems

Several operating systems that support decentralized information-flow control [70] have been proposed [60, 111, 31, 59]. These systems can prevent illegal flows of information through dynamic monitoring; the information-flow control is *decentralized* because each process can create new labels and assign permissions to it. As retrofitting code for security is presently difficult, label checking in DIFC systems is generally done dynamically, meaning that existing programs might be terminated. Our work aims to make it easier to retrofit applications with security mediators; such applications can be then used in DIFC-based operating systems to provide end-to-end system security. However, our work does not yet support all DIFC security operations such as tag creation and delegation.

Chapter 4

Finding the Cause of Information-Flow Errors

4.1 Introduction

As discussed in the previous chapter, security-typed languages add strong guarantees to trusted programs. However, although secure systems have been built using security-typed languages, at present these have been not been built from existing systems [5, 20, 48, 22]. As our ultimate goal is to add these guarantees to existing programs, it is important to understand why retrofitting has not been done.

Most programs are not written with information-flow security requirements in mind. Therefore, applying a security analysis to a program will likely reveal a large number of security violations, some of which correspond to expected system behavior (releasing the result of a password check), and some of which do not (the unintended release of a patient's social security number). Because any runtime program path can cause an error, these errors can be quite complex; 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 [16, 46], typing errors [105], and incorrect value computation [53, 99, 108]. However, current methods for determining causes of information-flow errors are either incomplete and non-minimal [27], or require computationally expensive auxiliary solvers for high precision [45]. Although our ultimate goal is on automatic resolution of errors, we require an understanding of how to solve the information-flow blame problem to assist programmers should an automatic resolution not be suitable.

In this chapter, 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 [71] with an inter-procedural 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 chapter 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.

4.2 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*, we investigate the security-typed language Jif [71], an extension of JFlow [69]. 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 [28]. 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

*Most implementations of information-flow checkers behave in the way described in this chapter, though their underlying type systems may vary.

`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.

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 [81] (described in Section 4.3.1), 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*.

4.2.1 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 of label L impose a security requirement on flows through the program: 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*. 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). 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 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

*`http://www.ericdaugherty.com/java/mailserver/`

```
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 }
```

Fig. 4.1. Fragment of code from the POP3 processer in the JES Email Server.

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 [45, 95], 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 [45] relies on *path conditions*, which are boolean conditions for determining when exactly a given program path is executed [83]. 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 [37], do not operate on a full programming language [27], or require an advanced solver to determine the conditions under which a leakage occurs [45].

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

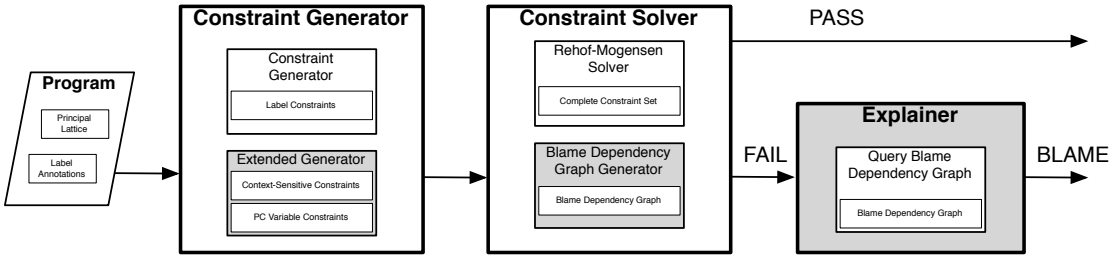

Fig. 4.2. **Modified Label Checking Process**: (1) a partially-labeled program is input to a *constraint generator* that generates the label constraints from the code using an interprocedural analysis; (2) a *constraint solver* determines if there is a satisfying assignment for the label variables, building 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.

argument and the side effects the method may release. This can lead to labeling conflicts rather than true security errors [20, 48].

4.2.2 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 4.2 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.

4.3 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.

4.3.1 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 [81]. 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.

The algorithm for Φ-SAT where C is a definite set of constraints is shown in Figure 4.3. A detailed explanation for the algorithm, including proofs of its tractability

and completeness, is given by Rehof and Mogensen [81]. 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.

All the Φ-terms τ that we consider in the thesis have the form $\tau \equiv \beta_0 \sqcup \cdots \sqcup \beta_k \sqcup L_1 \sqcup \cdots \sqcup L_j$. This assumption simplifies the presentation of the blame algorithm given in the upcoming section.

```
RMSolve(C)
 1  ρ(β) ← ⊥ for all β ∈ V
 2  W ← {τ ≤ β | τ ≤ β ∈ C such that L, ρ ⊭ τ ≤ β}
 3  while W is non-empty
 4      τ ≤ β ← Pop(W)
 5      if L, ρ ⊭ τ ≤ β
 6              ρ(β) ← ρ(β) ⊔ ρ(τ)
 7              for each τ' ≤ α ∈ C with β ∈ Vars(τ')
 8                  W ← Push(W, τ' ≤ α)
 9  for each τ ≤ L ∈ C
10      if L, ρ ⊭ τ ≤ L
11              raise exception
12  return ρ
```

Fig. 4.3. The Rehof-Mogensen constraint solver

4.3.2 Constraint Solving Example

Let $C = \{L_1 \leq \beta_0, L_2 \sqcup \beta_0 \leq \beta_1, \beta_0 \sqcup \beta_1 \leq \beta_2\}$, 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

$L_1 \leq \beta_0$. Because β_0 is currently mapped to $\bot$ under ρ, the solver raises $\rho(\beta_0)$ to L_1. Next, the solver considers $L_2 \sqcup \beta_0 \leq \beta_1$. 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 $\beta_0 \sqcup \beta_1 \leq \beta_2$; 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 $\tau \leq L \in C$, the solver succeeds, returning ρ.

Suppose instead the constraint $\beta_0 \sqcup \beta_1 \leq \beta_2$ 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, $\beta_0 \sqcup \beta_1 \leq \beta_2$ 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 $C' = C \cup \{\beta_2 \leq L_2\}$, the solver initially performs as described above. However, after each constraint of the form $\tau \leq \beta$ is considered, the solver attempts to check that $\beta_2 \leq L_2$ holds under ρ. However, as $\rho(\beta_2) = L_1 \sqcup L_2$ and $L_1 \sqcup L_2 \not\leq L_2$, the solver reports an error, specifically that it has failed on $\beta_2 \leq L_2$.

From the constraint set, it is not obvious why β_2 was raised above L_2. We know that $\beta_0 \sqcup \beta_1 \leq \beta_2$ 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.

4.4 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 4.2. 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*).

4.4.1 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 $\tau \leq \beta$ raise the level of a variable β. If the constraint $\beta \leq L$ fails, meaning that β was raised too high, the solver can use this information and determine whether or not the particular constraint $\tau \leq \beta$ was responsible for this.

4.4.1.1 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 4.5 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}.

Definition 4.4.1 (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 $\tau \leq \alpha$, then for each $\beta \in \text{Vars}(\tau)$ 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 $\tau \leq \alpha$.

A visualization of the dependency graph for the solver run over constraint set C from Section 4.3.2 is given in Figure 4.4. Initially each of the variables $\beta_0, \beta_1, \beta_2$ begin at $\bot$. The dependency graph indicates that the constraint $L_1 \leq \beta_0$ is responsible for raising β_0 from $\bot$ to L_1. Next, it shows that β_1 was raised because of the constraint $L_2 \sqcup \beta_0 \leq \beta_1$. Finally, it identifies that the constraint $\beta_0 \sqcup \beta_1 \leq \beta_2$ was responsible for raising β_2 to $L_1 \sqcup L_2$.

4.4.1.2 Using the Blame Dependency Graph

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

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

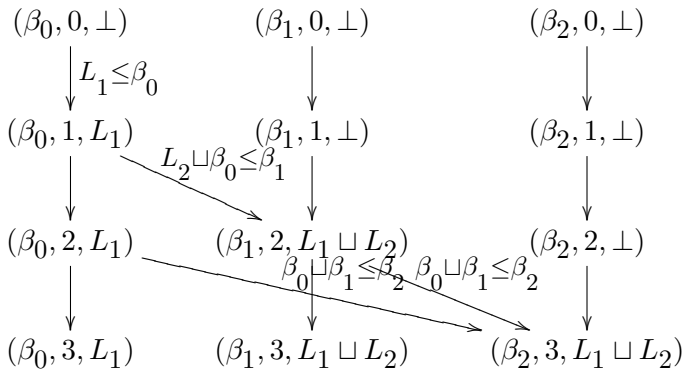

Fig. 4.4. The dependency graph after a solver run

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

4.4.1.3 Generating the Blame Dependency Graph

To populate the dependency graph during the run of the 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 $\tau \leq \beta$ and increment the current time. The modified solver algorithm is presented as RMSolveDependencyGraph, shown in Figure 4.5.

4.4.2 Reporting the Cause of an Error

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

The algorithm RecursiveExplain, given in Figure 4.5, recursively explains why a constraint $\tau \leq L$ failed, returning every constraint that contributed to its failure. When $\tau \leq L$ fails, RecursiveExplain consults the dependency graph to see why $\tau \leq L$ has failed, receiving the answer $\tau' \leq \beta$. Since $\tau \leq L$ 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 $\tau' \leq L$ to fail. This continues until the algorithm is called on an unsatisfiable constraint. In Section 4.4.3, 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 4.3.2 by examining the blame set returned by our explanation algorithm for the unsatisfiable set C', which contains the extra constraint $\beta_2 \leq L$. When the solver is run on C',

```
RMSolve-DependencyGraph(C)
 1  ρ(β) ← ⊥ for all β ∈ 𝒱
 2  W ← {τ ≤ β | τ ≤ β ∈ C such that 𝓛, ρ ⊭ τ ≤ β}
 3  t ← 0
 4  while W is non-empty
 5      τ ≤ β ← Pop(W)
 6      if 𝓛, ρ ⊭ τ ≤ β
 7          Record(t, β, ρ(β) ⊔ ρ(τ), τ ≤ β)
 8          ρ(β) ← ρ(β) ⊔ ρ(τ)
 9          t ← t + 1
10          for each τ' ≤ α ∈ C with β ∈ Vars(τ')
11              W ← Push(W, τ' ≤ α)
12  for each τ ≤ L ∈ C
13      if 𝓛, ρ ⊭ τ ≤ L
14          X ← RecursiveExplain(ρ(τ) ≤ L)
15          raise exception "τ ⋠ L because X"
16  return ρ

RecursiveExplain(τ ≤ L)
1  if τ ≤ L is unsatisfiable
2      return ∅
3  τ' ≤ β ← FailureCause(τ ≤ L)
4  return {τ' ≤ β} ∪ RecursiveExplain(τ' ≤ L)
```

Fig. 4.5. The Rehof-Mogensen Solver extended with error explanation

the solver fails attempting to verify $\rho(\beta_2) \leq L_2$. Instead of reporting this constraint as the sole cause of the violation, RecursiveExplain searches for the constraint that first caused $\beta_2 \leq L_2$ 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.

4.4.2.1 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.

4.4.3 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.

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 4.4.2 (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 4.4.3. 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 $\beta \in \mathrm{Vars}(\tau)$ (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 4.4.4. 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 4.4.5 (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 4.4.6 (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 4.4.8.

Lemma 4.4.7. *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 4.4.7, we can show that RecursiveExplain($\tau \leq L$) returns an error set for $\tau \leq L$.

Theorem 4.4.8 (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 4.4.7, $X_{\tau' \leq L} \cup \{\tau' \leq \beta\}$ 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 $\tau \leq L$ 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($\tau \leq L$).

Lemma 4.4.9. *If* RecursiveExplain$(\tau \leq L) = X$*, then* X *can be ordered as* $\langle \tau_0 \leq \alpha_0, \tau_1 \leq \alpha_1, \ldots, \tau_n \leq \alpha_n \rangle$*, where:*

1. *for* $i < j$, $\mathsf{fail}(\tau_i \leq L) < \mathsf{fail}(\tau_j \leq L)$
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 $\tau_i \leq \alpha_i$ according to their failure time; by Lemma 4.4.3, each recursive call has a distinct failure time.

To see that each α_i is distinct, observe that from the previous proof, $\mathsf{fail}(\alpha_i \leq L) = j = \mathsf{fail}(\tau_{i+1} \leq L)$, and so as the failure time $\mathsf{fail}(\tau_j \leq L)$ 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 $\mathsf{fail}(\tau_j \leq L) \leq \mathsf{fail}(\alpha_i \leq L) = \mathsf{fail}(\tau_{i+1} \leq L)$; therefore, each of the α_i must not occur in any later τ_j. The full statement follows from $\mathsf{fail}(\alpha_n \leq L) = \mathsf{fail}(\tau \leq L)$. □

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 4.4.2, 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 4.4.10 (Minimality)**.** *There is no* $X' \subset X_{\tau \leq L}$ *such that* X' *is an error set for* $\tau \leq L$.

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

By Lemma 4.4.9, the error set X can be ordered by failure time as $\langle \tau_0 \leq \alpha_0, \tau_1 \leq \alpha_1, \ldots, \tau_n \leq \beta \rangle$, with each of the variables occurring on the right-hand side of the equation is distinct. Let k be the first $\tau_k \leq \alpha_k \in X_{\tau \leq L}$ such that $\tau_k \leq \alpha_k \notin X'$. For $m > k$, $\mathrm{Vars}(\tau_m) \cap \{\alpha_0, \ldots, \alpha_{k-1}\} = \emptyset$. Therefore, we can satisfy $X' \cup \{\tau \leq L\}$ by running the Rehof-Mogensen solver over $\{\tau_0 \leq \alpha_0, \ldots, \tau_{k-1} \leq \alpha_{k-1}\}$ and leaving $\alpha_{k+1}, \ldots, \alpha_n$ at $\bot$. As τ does not share variables with $\alpha_0, \ldots, \alpha_{n-1}$ and X' is nonempty (so $\tau \leq L$ must be satisfiable), the valuation ρ produced from this run will satisfy $\rho(\tau) \leq L$. This is a contradiction. □

We now provide a proof that finding the *minimum* error set, as opposed to a *minimal* error set returned by the algorithm in Section 4.4, is NP-complete.

Definition 4.4.11 (Vertex-cover)**.** Given a graph $G = (V, E)$ and a positive integer $k \leq |V|$. Is there a *vertex cover* of size k or less for G, that is $V' \subseteq V$ such that $|V'| \leq k$ and for each edge $\{u, v\} \in E$, atleast one of u and v belongs to V'?

Definition 4.4.12 (MFP-unsatisfiable-core)**.** Let $\Phi = (L, F)$ be a *monotone function problem (MFP)*, where L is a lattice and F is a finite set of monotone functions. Let C be a set of definite constraints that is unsatisfiable. Is there a *unsatisfiable core* of C of size $\leq k$, i.e., a set of constraints $C' \subseteq C$ such that C' is unsatisfiable and $\mid C' \mid \leq k$?

Lemma 4.4.13. The MFP-unsatisfiable-core problem is NP-complete.

Proof. It is clear that the MFP-unsatisfiable-core problem is in NP. We can simply guess a set of constraints $C' \subseteq C$ and check that it is unsatisfiable.

Let $G = (V, E)$ be a graph. Now we describe the system of constraints C_G correspond to the graph $G = (V, E)$. C_G will be unsatisfiable. Moreover, C_G will have an unsatisfiable core of size $\leq k+ \mid E \mid +1$ iff G has a vertex cover of size $\leq k$.

For each vertex $v \in V$, we have a variable α_v. Similarly we have a variable α_e for each edge $e \in E$. We will work in the two-point Boolean lattice, i.e., $L = \{0, 1\}$ and $x \sqcap y = x \wedge y$ and $x \sqcup y = x \vee y$. For each edge $e = (u, v)$ in E we have the following two constraints:

$$\alpha_u \leq \alpha_e$$

$$\alpha_v \leq \alpha_e$$

For each vertex $v \in V$, we have the following constraint:

$$1 \leq \alpha_v$$

Finally, we have the following constraint:

$$\sqcap_{e \in E} \alpha_e \leq 0$$

First, note that each constraint in C_G is definite. Moreover, the set of constraints C_G is unsatisfiable. Let ρ be the minimal solution computed by the Rehof-Mogensen solver. The constraint $1 \leq \alpha_v$ forces $\rho(\alpha_v) = 1$. The constraints corresponds to the edges forces $\rho(\alpha_e) = 1$ for all $e \in E$. This means that the following constraint is violated by

ρ:

$$\sqcap_{e \in E} \alpha_e \leq 0$$

Assume that $G = (V, E)$ has a vertex cover of V' size k. Then there exists an unsatisfiable core C' of C_G of size $k + |E| + 1$. For each, $v \in V'$ we put the constraint $1 \leq \alpha_v$ in C'. For each edge $e = (u, v)$, we put a constraint $\alpha_u \leq \alpha_e$ in C' such that $u \in V'$ (note that such a u exists because V' is a vertex cover). Finally, we put the constraint $\sqcap_{e \in E} \alpha_e \leq 0$ in C'.

Assume that C_G has an unsatisfiable core C' of size less that or equal to $k+|E|+1$. It is clear that the constraint $\sqcap_{e \in E} \alpha_e \leq 0$ has to be in C' (otherwise there is no constraint to violate). Moreover, for each $e = (u, v)$ one of the following constraints has to be in C'.

$$\alpha_u \leq \alpha_e$$

$$\alpha_v \leq \alpha_e$$

If none of the constraints given above is in C', then $\rho(\alpha_e) = 0$, where ρ is the solution returned by algorithm shown in Figure 4.3 on the set of constraints C'. This means that the constraint $\sqcap_{e \in E} \alpha_e \leq 0$ is not violated. Let $V(C')$ be the set of vertices in V such that the constraint $1 \leq \alpha_v$ is in C'. It is easy to see that $V(C')$ is a vertex cover for G and the size of $V(C')$ is $\leq k$. □

4.4.4 Error Traces From Code

Figure 4.6 and Figure 4.7 show the error sets returned by our blame algorithm for two of the errors as described in Figure 4.1 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 4.5.3. Definitional constraints, specified by contraints of the form $v =_{def} l$, are syntactic sugar for a constraint $l \leq v$, where v first appears in this constraint.

For both figures, the failed constraint is the same: however, the reason is different. Our tool first returned the error set associated with Figure 4.6, 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 4.6, 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.

```
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: }
```

Fig. 4.6. Error set and error trace for first error from the code in Figure 4.1. 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: }
```

Fig. 4.7. The error set for second error from the code in Figure 4.1.

4.5 Blame in Program Code

The Rehof-Mogensen solver together with the dependency graph can give explanations for why constraints failed, independent 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 4.6.

4.5.1 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 [69]. 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 4.4 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.

4.5.2 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 [91]. 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 4.6) 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)*. 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.

4.5.3 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,

*`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.

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, 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 [25]. We found that using program counter variables greatly improved our error explanations at the cost of increasing their length.

4.6 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 [45]). 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 [45].

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 4.1.

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/`.

4.6.1 Java Card Wallet

The Java Card Wallet is a small example designed to teach Java programmers how to program in Java Card [*]. 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

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

Application	LOC	*ConstraintSet 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 4.1. 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.

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.

4.6.2 Mental Poker

Mental Poker was one of the original implementations of a non-trivial application in a security-typed language [5]. 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.

4.6.3 Java Card Purse

The PACAP Purse is a prototype designed to secure information flow in Java Card applications [15]. 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).

4.6.4 JES Email Server

As described in Section 4.2, JES is an open-source Java email server with full POP3 and SMTP functionality. Its functionality is a superset of JPMail [48], 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 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.

4.6.5 tinySQL

tinySQL is a minimal implementation of an SQL client and server on DBF and text files*. 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.

4.6.6 Comparison to Program Slicing

Using slicing to explain information-flow errors has not been well investigated. Previous work by Hammer et al. [45] 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* [83] to precisely specify the conditions under which such a leakage occur. However, they do not specify

*`http://www.jepstone.net/tinySQL/`

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 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.

4.6.7 Discussion

Table 4.2 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 [82].

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.

4.6.8 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

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 4.2. 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.

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*. 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 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.

4.7 Related Work

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

*`http://www.cse.psu.edu/~dhking/siggen`.

4.7.1 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 [105]. 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 [32]. Finding root-causes of counterexample traces has been used in software verification for abstraction refinement [9].

4.7.2 Explaining Information-Flow Errors

Hammer *et al.* [45] 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 [27]. 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.

4.7.3 Type Qualifiers

Our tool has a similar goal to that of CQual, a tool that refines the C type system with qualifiers on types [37], allowing programmers to find inconsistencies between program points. CQual has been used to find bugs in the Linux kernel [54, 112]. 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 [54] 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.

Chapter 5

Automatically Resolving Information-Flow Errors

Understanding error messages allows programmers to trace information-flow errors from source to sink. However, the technology in Chapter 4 still requires the programmer to examine and resolve each error message individually. For many non-trivially-sized programs, this represents an unreasonable burden.

We now focus on *automating* the resolution of information-flow errors.

Security-typed languages [69, 78] use type systems that augment the types of data with security labels to statically verify that a program satisfies a security property based on a relationship among those labels. However, many programs exhibit behavior that is not compatible with a static type system. For example, we do not know whether a user accessing patient data is assigned a doctor label or another label until runtime, requiring an authorization check. Also, secret data may need to be released under controlled conditions (e.g., patient records to a new doctor), requiring a declassification that changes the data's security label.

To resolve these conflicts within the type system, programmers insert *mediation statements*, such as declassifiers or authorization checks, that ensure that the runtime behavior of the program remains consistent with the security labels expressed by the type system. Currently, the addition of mediation statements is a manual task that requires examining a large amount of code and careful consideration to avoid errors.

Automatic tools can identify missing mediation statements [39, 69, 112]. However, even once the errors have been identified, reaching consensus on manual placement often takes a long time (e.g., for the X Window Server [104]). Given a set of candidate mediation statements, they may not actually resolve all labeling conflicts, they may contain redundant statements, may significantly degrade performance, and they may violate the program's coding style.

As a result, there is a need for a more automated approach that helps programmers determine a placement of mediation statements that ensures that the program correctly enforces a security policy. Finding legal and reasonable placements for non-trivial programs is complex, as there may be several type system conflicts, each with several possible resolutions that may interact. Recent work constructed an approach that enumerates complete and minimal sets of causes for a type system conflict [58]. However, the programmer must still correctly choose placements that resolve these causes, accounting for interaction among conflicts.

In this chapter, we present a method for automatic identification of mediation points in legacy programs that is based on a minimum cut graph approach. A mediation point is a location where a mediation statement can be placed. We were inspired to investigate mediation point placement as a minimum cut problem due to recent work assigning a quantitative measure of leaked information in a program by solving a maximum-flow graph problem [67]. By solving a minimum-cut problem, the dual of maximum-flow, we present the programmer with options biased toward lower programmer effort (i.e., programmers only have to write the fewest number of authorization checks and declassifiers to enforce their security property). In the case where these points are not satisfactory to

the programmer, using the minimum-cut problem also enables the programmer to guide the solver, such as by disallowing placement of mediation points in certain sections of code or by giving a preference to certain types of mediation points that imply security enforcement.

Our method outputs a set of *suggestions*: each suggestion is a set of locations for placing mediation statements that resolve a program's type system conflicts. We outline the properties required of the type system so that mediation points can be determined by finding the minimum cut of a graph. Once the minimum cut has been determined, we use existing graph algorithms to output each minimum cut of the graph, thereby providing the user with a set of legal placement suggestions to assess, reducing their effort significantly. Our evaluation shows that these suggestions are located in similar places to those manually selected by human programmers in the same codebases.

We make the following contributions in this chapter:

- We recall the standard transformation of a set of information-flow constraints for a program into an information-flow graph. We show how to modify the constraint generation process so that the corresponding information-flow graph has the property that every source-sink cut of the graph corresponds to a set of mediation points that completely resolve the program's illegal information flows.

- We develop an automated technique to output *suggestions* for mediation points based on minimum cuts of the information-flow graph. We describe how we modified the security-typed language Jif to output label constraints that could be converted into an information-flow graph. We also applied a method for clustering

expressions to prevent many redundant suggestions from being output to the programmer.

- In our experiments, our tool reduces the number of locations that would be required for a programmer to examine given current tools for finding security-typed language errors by 85%, and in programs originally written in a security-typed language, more than 95% of the selected mediation points were classified as similar to those placed by human programmers.

The minimum-cut approach presented in this chapter provides a framework for programmers to solve practical placement concerns, such as ensuring solutions: resolve all conflicts, contain no redundant mediators, and can account for performance and style considerations.

5.1 Overview

In this section we introduce some of the challenges with placing mediation statements in program code. In security-typed languages, programmers specify security properties in code by annotating various security-relevant sources and sinks in the program with security labels from a lattice $\mathcal{L}$. These languages enforce *noninterference* [43]: a program satisfies noninterference if at runtime, the computation of data with security label l is independent of data with security label l' if $l \not\leq l'$ in $\mathcal{L}$. Noninterference can be used to model both secrecy and integrity requirements, depending on the semantics of the labels in $\mathcal{L}$. A program satisfying noninterference is also said to satisfy *information-flow security*. Statically checking noninterference has two problems: (1) without a notion

of declassification [86], programs can never violate $\mathcal{L}$, even when properly releasing data (e.g., releasing patient records to new doctors), and (2) without runtime authorization checks, we have no way to enforce $\mathcal{L}$ over labels whose security values may be instantiated at runtime, causing the program to unnecessarily not satisfy noninterference. *Mediation statements* allow programs to execute flows between label l and incomparable labels l'.

To investigate issues in placing mediation statements in code, we introduce the example of `logrotate`, a program that rotates system logs into backup files. `logrotate` is trusted to maintain the security properties of the operating system: if the user configuring `logrotate` is not allowed to perform an action, then `logrotate` should not be allowed to perform that same action. In recent work [52], a version of `logrotate` has been written in the security-typed language Jif [69], a variant of Java. The Jif version of `logrotate` guarantees that the program satisfies information flow security. However, it also requires that the programmer insert mediation statements to allow information to flow from the `logrotate` configuration files to the logs being rotated. Without a mediation check, `logrotate` could be used to violate the secrecy and integrity guarantees of the system: for example, by revealing configuration data through viewing the results of log rotation or by a user using `logrotate` to modify log file data that she does not have access to.

The code on the left of Figure 5.1 shows the rotation code from `logrotate` without any statements to mediate security. Information from the configuration file is given the security level {`config`} and data in the log files is given the security level {`log_lbl`}. These labels are *runtime labels*: when the application runs, these labels will be instantiated with the actual security labels of the configuration and log files respectively. In the given code,

for a given `filename`, the code is told to rotate `rotateCount` logs. If `filename` is `"messages"` and `rotateCount` is 5, then the code renames each log name of the form `messages.i` to `messages.i`+1 (for `i` between 1 and 5) and finally deletes `messages.6`. In the above code, the `getFile` function returns a handle to a file, while `renameTo` renames a file.

However, the code on the left contains an illegal flow of information from {`config`}, a runtime label representing the security of the `logrotate` configuration files, to {`log_lbl`}, a runtime label representing the security of the current log being rotated. This is a violation of both the secrecy and integrity guarantees of `logrotate`. To see that the program permits an integrity violation, observe that modifying {`config`}-level data affects {`log_lbl`}-level data. If an attacker reduces the number of logs maintained by `logrotate` and can influence when `logrotate` is run, the attacker could hide attacks on the system by attacking and then rotating the logs containing evidence of these attacks off the system. The program also permits a secrecy violation: an observer can gain information about the {`config`}-level file by observing its effects on {`log_lbl`}-level logs. By inserting mediation statements, a programmer using a security-typed language will be able to ensure that `logrotate` respects the security policies of the system, while still permitting it to operate as required to rotate the system logs.

We highlight three individual flows from the configuration file to the rotated logs in Figure 5.1. Each flow requires mediation.

- The number of logs to rotate before deleting the final log (`rotateCount`) is equal to the number of file rename operations performed.

- The filename specified by the configuration file is used to get a handle to the system file that `logrotate` renames through `oldName`. If an attacker can control this variable, then she can determine which logs to be rotated.

- The filename specified by the configuration file is used to create the new name that a log file is renamed to, `newName`. If an attacker can control this variable, then she could cause a file to be overwritten.

The code on the right of Figure 5.1 shows the logrotate code with mediation statements inserted. Each of the above flows has been mediated by adding a `mediate(e,lbl)` expression: if the label on the expression `e` is allowed to flow to `lbl`, then the expression has the value of `e` with the security label of `lbl`. Otherwise, the program throws a security exception and terminates. The placement given in the right half of the figure is not the only possible placement: for example, it would have also been possible to mediate the loop guard `i` $>= 0$, which would have disconnected the number of times the loop was executed from data labeled as {`config`}.

To place mediation statements that resolve these information flows, a programmer must first annotate the sources and sinks in the program with their security semantics. Next, the programmer must examine each line of code contributing to these errors. She can use automated methods to identify possible causes of information-flow errors [58]. Resolving these errors is currently a manual process and requires the error explanation analysis to be run multiple times to resolve each of the possible causes of an information-flow error. An automated solution would free the programmer from having to examine all the error explanations, requiring them only to determine whether the selected mediation

```
1 label config, log_lbl, LogInfo[{config}]{config} log;
2 String{config} filename = log.getFilename(logNum);
3 int{config} rotateCount = log.getRotateCount();
4 File[{log_lbl}]{log_lbl} disposeFile =
5   Runtime.getFile(filename+"."+(rotateCount+1),log_lbl);
6 File[{log_lbl}]{log_lbl} newlogfile, oldlogfile = null;
7 // rename messages.n to messages.n + 1
8 for (int i = rotateCount ; i >= 0; i--) {
9    String newName = filename + "." + i;
10   String oldName = filename + "." + (i-1);
11   newlogfile = Runtime.getFile(newName,log_lbl);
12   oldlogfile = Runtime.getFile(oldName,log_lbl);
13   if (oldlogfile != null)
14      oldlogfile.renameTo(newlogfile);
15 }
```

```
1 label config, log_lbl, LogInfo[{config}]{config} log;
2 String{config} filename = log.getFilename(logNum);
3 int{config} rotateCount = log.getRotateCount();
4 File[{log_lbl}]{log_lbl} disposeFile =
5   Runtime.getFile(filename+"."+(rotateCount+1),log_lbl);
6 File[{log_lbl}]{log_lbl} newlogfile, oldlogfile = null;
7 // rename messages.n to messages.n + 1
8 for (int i = mediate(rotateCount,log_lbl); i >= 0; i--) {
9    String newName = filename + "." + i;
10   String oldName = filename + "." + (i-1);
11   newlogfile = Runtime.getFile(mediate(newName,log_lbl),
12                                log_lbl);
13   oldlogfile = Runtime.getFile(mediate(oldName,log_lbl),
14                                log_lbl);
15   if (oldlogfile != null)
16      oldlogfile.renameTo(newlogfile);
17 }
```

Fig. 5.1. **Top:** example from `logrotate` that performs rotation of log files. **Bottom:** the same code, with mediation statements inserted. The expression `mediate(e,lbl)` checks that the current label of the expression `e` is allowed to flow to the runtime label `lbl`. If so, the resulting expression is given label `lbl`. Otherwise, an exception is thrown and program execution is aborted. Here, three mediation statements are inserted to relabel data from the configuration file (`config` level) to the level of the logs (`log_lbl` level). Without this relabeling, the program cannot guarantee that high-integrity logs will not be affected by lower-integrity data and will not be certified by a security-typed language compiler.

points were suitable or not. Our method uses the results of a whole-program information-flow analysis to suggest a set of *mediation points*, locations in code where mediation statements can be inserted to resolve a program's labeling conflicts. The particular mediation mechanism required is application-specific, and so ultimately the programmer must decide for each selected mediation point what type of mediation statement should be inserted.

Often programmers have certain placement constraints with regards to where mediation statements should not be placed. For example, class A is used for string formatting, while class B implements cryptographic operations on the contents of a string. Programmers might therefore prefer to perform a mediation statement in class B rather than class A so that security operations are performed in classes already used for security. Any automated system should be *customizable*, as requirements of this type for declassifier placement differ across applications and programmers.

5.2 Background

In this section we review how information-flow checking is performed by security-typed language compilers. Rather than adapt our program to any specific security-typed language, we operate on a set of *type constraints*, an abstraction that many static type inference systems can be adapted to use [76]. As the type systems we consider in this work are information-flow based, their associated contraints are referred to as *information-flow constraints*.

5.2.1 Information-Flow Checking

Security-typed languages augment traditional compilers to allow programmers to specify the security properties of program data. Generally, these languages enforce noninterference in code by augmenting the type system with security types. There are two different categories of illegal information flow that noninterference disallows. *Explicit information flows* occur when high security data is written to a low output, such as writing a secret key to a socket. *Implicit information flows* occur when high security data otherwise affects a low observable result. For example, a password check that compares the hash of a guess against the hash of a password and reveals that information is an implicit flow of information. If `h` and `l` are high and low variables respectively, then the assignment `l` := `h` is an explicit flow of information, while the conditional **if** `h` **then** `l` := 1 is an implicit flow of information.

To prevent the programs from releasing secure information through an explicit information flow, types τ are annotated with labels l, and the type system forbids subtyping of the form $\tau\{l\} \preceq \tau\{l'\}$ if $l \not\preceq l'$. To prevent information from leaking through implicit flows, the type system maintains a label containing the security level of the program counter. This security label is equal to the join of all of the security labels that the execution of the current expression depends on. When an assignment is performed, the type system verifies that the variable being assigned to is less than or equal to the program counter.

For example, the program **if** `h` **then** `l` := 1 modifies low security data based on the value of high security data, and so contains an information flow from high to low, a

violation of noninterference. When the security-typed language compiler processes the condition `h`, the program counter is set to `high`. Next, it examines the assignment `l` := 1. Because the assignment is to a variable with security level `low` and the label program counter is not at a label $\leq$ the level of the variable being assigned, this statement is forbidden and the program is rejected as insecure.

To enforce these security guarantees, type systems generate *information-flow constraints* from the program. Information-flow constraints contain both security labels l from the lattice $\mathcal{L}$ as well as label variables α representing the security level of program elements that have not been explicitly labeled. A security type system generates a set C of information-flow constraints corresponding to the information flows that a program permits. If there exists a mapping ρ from label variables to security labels such that for each constraint $\xi \in C$, the substituted $\rho(\xi)$ holds, then C is *satisfiable*. A program with a satisfiable information-flow constraint set satisfies noninterference.

Below, we introduce a method for generating information-flow constraints C such that we can determine sets of mediation points in a way that resolves all of a program's illegal information flows.

5.3 Constraint Methodology

In this section, we show how to generate information-flow constraints so that finding a minimum set of mediation statements can be solved as a graph-cut problem. We first introduce sIMP, a constraint-based type system for IMP, a simple imperative language [109]. The IMP language contains conditionals, variable assignment, and while loops, and is presented as a simple foundational language. We then describe how to

transform the information-flow constraints generated by the sIMP analysis into a graph that has the property that a vertex cut is equivalent to a set of mediation points that resolve the illegal information flows in the program. We will describe in Section 5.3.6 how we modified the constraints generated by the Jif compiler to satisfy this property.

The main technical distinction between the constraint-based type system presented here and standard type systems for information-flow security, such as the one presented by Volpano *et al.* [101], is that sIMP does not assume a total mapping from each variable to its security level. In the case where every variable is assigned a security level, there is no ambiguity as to where to place a mediation statement. In legacy code, it is unreasonable for the programmer to assign security semantics to each variable, meaning that the security level of an expression e is equal to the security level of every expression affecting e. A constraint-based type system models language expressions that have an undetermined security semantics: in sIMP, the security label of an expression e is associated with a unique label variable α_e.

In sIMP, a command c is information-flow secure if the set of information-flow constraints C that the type system assigns to c is satisfiable. Using a standard technique from the literature [35, 36, 47, 90], we view the information-flow constraints C as a directed graph $\mathcal{G}_C$, which we refer to as *information-flow graph*. If C contains the constraint $\tau \leq A$ (where A is an atom: either a lattice variable or lattice element and τ is a join of atoms), then there is an edge from each atom in τ to A. The information-flow graph therefore contains a path between two nodes $n_1, n_2 \in \mathcal{G}_C$ if the value of the program element associated with n_1 can affect the value of the program element associated with n_2. We first show that for a two-point lattice consisting of $\top$ and $\bot$,

the constraints generated by sIMP have the *cut-mediation equivalence* property, meaning that a set of mediation statements that resolve the illegal $(\top, \bot)$ flows in a sIMP program is equivalent to a $(\top, \bot)$ cut of the information-flow graph. We show how to generalize this approach for arbitrary lattices in Section 5.3.4.

5.3.1 A Constraint-Based Type System For Secure Information Flow

We now introduce sIMP, a constraint-based type system for enforcing secure information flow in IMP. We begin by introducing the IMP language. IMP contains two distinct syntactic elements: commands and expressions. A command c can modify a global program state σ, while an expression e evaluates to an integer value n using variable bindings from σ. An example of an IMP command is $x := x + 1$: this updates the variable x to be equal to the current value of x added to 1. Commands c and expressions e in IMP have the following grammar*:

$$
\begin{array}{llll}
\text{Integers} & n & ::= & 0, 1, \ldots \\
\text{Variables} & v & ::= & x, y, \ldots \\
\text{Expressions} & e & ::= & n \mid v \mid e_1 + e_2 \\
\text{Commands} & c & ::= & \mathsf{skip} \mid c_1 \ ; \ c_2 \mid v := e \mid \\
 & & & \quad \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c
\end{array}
$$

Let σ be a memory, mapping variables to integer values. Evaluation in IMP has the judgment $\langle \sigma, c \rangle \rightarrow \sigma'$: under memory σ, command c produces memory σ'. Evaluating

*For simplicity, we omit presenting the semantics for handling Boolean values. This modification does not affect the security properties of sIMP.

the above command under a memory that maps x to the integer 4 returns a memory mapping x to the integer 5. This is written $\langle \{x \mapsto 4\}, x := x + 1\rangle \rightarrow \{x \mapsto 5\}$. The evaluation semantics for IMP are standard big-step semantics: as our focus is on static checking of the security properties of IMP commands, we omit the presentation of these.

5.3.1.1 Label Constraints

To enforce information-flow security on IMP, we define a *constraint-based type system* that determines *label constraints* from a command c that describes the flows that c enables in a security lattice $\mathcal{L}$. If a command c has a set of label constraints that is *satisfiable*, then for all flows that c enables from l_1 to l_2, $l_1 \leq l_2$ in the lattice $\mathcal{L}$. If $l_1 \not\leq l_2$, then this flow will require mediation before the program can be used as a component of a secure system. We now give the syntax of label expressions and constraints.

Label Variables	α	::=	$\alpha, \beta, \ldots \in \mathcal{V}$	Security Labels	l	::=	$l \in \mathcal{L}$
Atoms	a	::=	$\alpha \mid l$	Label Joins	τ	::=	$a \mid a \sqcup \tau$
Constraints	ξ	::=	$\tau \leq A$				

An atom a_i is a label expression that is either a label variable α or a label $l \in \mathcal{L}$. Label joins have the form $a_1 \sqcup \cdots \sqcup a_n \leq a_0$. A label variable α represents the security level of an expression that has not been explicitly been labeled by the programmer. A label l represents a value that has been given a security semantics ahead of time: for example, a key used for encryption that has been read from a file would be given a `Secret` security value.

We now give a security type system for IMP (sIMP) that enforces noninterference of high and low security data. Let Γ be a context assigning a security level to seed variables, which is a subset of the set of all program variables, and Δ be a context assigning to each program variable x a unique security variable α_x. To track implicit flows, the type system also keeps track of the current label of the program counter with the *pc* label. The constraint generation rules are given in Figure 5.3. If the generated constraint set C for a command c is satisfiable, then when run, c will not cause any high-security data to affect low-security data. The type judgments presented in this figure are for both expressions e and commands c. An expression is assigned information-flow constraints C and a security variable α with the judgment $\Gamma; \Delta; pc \vdash e : \alpha, C$, while a command c is assigned information-flow constraints C with the judgment $\Gamma; \Delta; pc \vdash c : C$. We associate expressions e with a unique security variable α_e so that the vertices corresponding to a cut of the graph are uniquely identified with mediation points. In the type checking rules, we add a unique position tag p to refer to expressions e, allowing us to uniquely refer to subexpressions. We write $(e)_p$ to indicate that expression e has the position p (assumed to be taken from a unique set of positions). This is similar to converting a program to SSA form [25]. We refer to the label variable $\alpha_{e,p}$ as the *expression variable* for the expression-position pair (e, p). In the case where there is no loss of ambiguity, we refer to α_e as the expression variable for e.

The constraints generated by sIMP satisfy the standard guarantee of non-interference for an information-flow type system: if the constraints C generated by the type system for a commadn c that contains no mediation statements are satisfiable, then c does not permit an unmediated flow of information between lattice labels. This is

formulated as the condition that the run of a command under a memory μ is observationally equivalent to a run of that command on a memory ν that agrees with μ on all low-security values. Specifically, μ and ν may contain different high-security information, but if c is noninterfering, there is no way to tell how they differ by observing the memory during a run of c*

Definition 5.3.1 (Low-Equivalence)**.** Let μ, ν be program memories and Γ be a security context. If for all $x \in \text{dom}(\Gamma)$ such that $\Gamma(x) = \bot$, $\mu(x) = \nu(x)$, we say μ and ν are low-equivalent, written $\Gamma \vdash \mu \simeq_l \nu$.

Theorem 5.3.2 (Noninterference)**.** *Let* $\mu \simeq_l \nu$ *if* $\text{dom}(\mu) = \text{dom}(\nu)$ *and for all* $x \in \text{dom}(\mu)$ *such that* $\Gamma(x) = \bot$, $\mu(x) = \nu(x)$*). If* $\Gamma; \Delta; \bot \vdash c : C$ *such that* c *contains no mediation expressions,* C *is satisfiable,* Δ *is one-to-one, and for all* $x \in \Gamma$ *such that* $\Gamma(x) = \bot$, $\mu(x) = \nu(x)$*, then for* $\langle \mu, c \rangle \rightarrow \mu'$ *and* $\langle \nu, c \rangle \rightarrow \nu'$, $\Gamma \vdash \mu' \simeq_l \nu'$.

Proof. Proof proceeds by induction on the typing derivation. **Work through this – will need an auxiliary lemma for expressions**

□

5.3.1.2 Constraint Example

We now investigate the information-flow constraints associated with the main loop in Figure 5.1 by building the information-flow constraint set C. The constraints

*Our type system does not prevent termination leaks. For example, the program **while** `h` == 0 **do** `skip` only terminates if `h` is non-zero, allowing an attacker to gain information about the run of a program through means other than observing the low-security variables in a memory. Many systems [6] that consider termination channels as low-observable output, and it would be possible to enforce this condition in sIMP by disallowing **while** statements from being based on $\top$-level information.

Expressions

$$\langle \mu, n \rangle \to n$$

$$\langle \mu, v \rangle \to \mu(v)$$

$$\frac{\langle \mu, e_1 \rangle \to n_1 \qquad \langle \mu, e_2 \rangle \to n_2 \qquad n_1 + n_2 = n}{\langle \mu, e_1 + e_2 \rangle \to n}$$

$$\frac{\langle \mu, e \rangle \to n}{\langle \mu, \mathsf{mediate}(e) \rangle \to n}$$

Commands

$$\langle \mu, \mathsf{skip} \rangle \to \mu$$

$$\frac{\langle \mu, c_1 \rangle \to \mu_0 \qquad \langle \mu_0, c_2 \rangle \to \mu'}{\langle \mu, c_1 \ ; \ c_2 \rangle \to \mu'}$$

$$\frac{\langle \mu, e \rangle \to n}{\langle \mu, v := e \rangle \to \mu[v \mapsto n]}$$

$$\frac{\langle \mu, e \rangle \to n \qquad n \neq 0 \qquad \langle \mu, c_1 \rangle \to \mu'}{\langle \mu, \mathsf{if} \ v \ \mathsf{then} \ c_1 \ \mathsf{else} \ c_2 \rangle \to \mu'}$$

$$\frac{\langle \mu, e \rangle \to 0 \qquad \langle \mu, c_2 \rangle \to \mu'}{\langle \mu, \mathsf{if} \ v \ \mathsf{then} \ c_1 \ \mathsf{else} \ c_2 \rangle \to \mu'}$$

$$\frac{\langle \mu, \mathsf{if} \ v \ \mathsf{then} \ (c \ ; \ \mathsf{while} \ v \ \mathsf{do} \ c) \ \mathsf{else} \ \mathsf{skip} \rangle \to \mu'}{\langle \mu, \mathsf{while} \ v \ \mathsf{do} \ c \rangle \to \mu'}$$

Fig. 5.2. Operational Semantics for sIMP

Expressions

$$\frac{\alpha_{n,p} \text{ fresh}}{\Gamma; \Delta \ \vdash (n)_p : \alpha_{n,p}, \emptyset}$$

$$\frac{x \in \mathrm{dom}(\Gamma) \qquad \alpha_{x,p} \text{ fresh}}{\Gamma; \Delta \ \vdash (x)_p : \alpha_{x,p}, \{ \quad \begin{array}{l} \alpha_{x,p} \leq \Delta(x), \Delta(x) \leq \alpha_{x,p}, \\ \Gamma(x) \leq \Delta(x), \Delta(x) \leq \Gamma(x) \quad \} \end{array}}$$

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \alpha_{x,p} \text{ fresh}}{\Gamma; \Delta \ \vdash (x)_p : \alpha_{x,p}, \{\alpha_{x,p} \leq \Delta(x), \Delta(x) \leq \alpha_{x,p}\}}$$

$$\frac{\Gamma; \Delta \ \vdash e_1 : \alpha_1, C_1 \qquad \Gamma; \Delta \ \vdash e_2 : \alpha_2, C_2 \qquad \alpha_{3,p} \text{ fresh}}{\Gamma; \Delta \ \vdash (e_1 + e_2)_p : \alpha_{3,p}, C_1 \cup C_2 \cup \{\alpha_1 \sqcup \alpha_2 \leq \alpha_{3,p}\}}$$

$$\frac{\Gamma; \Delta \ \vdash e : \alpha_0, C \qquad \alpha_{1,p} \text{ fresh}}{\Gamma; \Delta \ \vdash (\mathsf{mediate}(e))_p : \alpha_{1,p}, C}$$

Commands

$$\Gamma; \Delta; pc \ \vdash \mathsf{skip} : \emptyset$$

$$\frac{\Gamma; \Delta; pc \ \vdash c_1 : C_1 \qquad \Gamma; \Delta; pc \ \vdash c_2 : C_2}{\Gamma; \Delta; pc \ \vdash c_1 \ ; \ c_2 : C_1 \cup C_2}$$

$$\frac{\Gamma; \Delta \ \vdash v : \alpha_0, C_0 \qquad \Gamma; \Delta \ \vdash e : \alpha_1, C_1}{\Gamma; \Delta; pc \ \vdash v := e : C_0 \cup C_1 \cup \{\alpha_1 \sqcup pc \leq \Delta(v)\}}$$

$$\frac{\begin{array}{cc} \Gamma; \Delta \ \vdash e : \alpha_0, C_0 & \Gamma; \Delta; \alpha_{pc} \ \vdash c_1 : C_1 \\ \alpha_{pc} \text{ fresh} & \Gamma; \alpha_{pc} \ \vdash c_2 : C_2 \end{array}}{\Gamma; \Delta; pc \ \vdash \mathsf{if} \ e \ \mathsf{then} \ c_1 \ \mathsf{else} \ c_2 : \begin{array}{l} C_0 \cup C_1 \cup C_2 \cup \\ \{pc \sqcup \alpha_0 \leq \alpha_{pc}\} \end{array}}$$

$$\frac{\Gamma; \Delta \ \vdash e : \alpha_0, C_0 \qquad \Gamma; \Delta; \alpha_{pc} \ \vdash c_1 : C_1 \qquad \alpha_{pc} \text{ fresh}}{\Gamma; \Delta; pc \ \vdash \mathsf{while} \ e \ \mathsf{do} \ c : C_0 \cup C_1 \cup \{pc \sqcup \alpha_0 \leq \alpha_{pc}\}}$$

Fig. 5.3. Type Inference Rules for sIMP

generated by this program represent the information flows through the program. Later in the section, we will show how these constraints induce an information-flow graph on the label variables and lattice elements.

Let α_{rc}, α_{fn}, α_{nn}, $\alpha_{on}, \alpha_i, \alpha_{nlf}, \alpha_{olf}$ be the expression variables associated with the variables `rotateCount`, `filename`, `newName`, `oldName`, `i`, `newlogfile`, and `oldlogfile` respectively. For a variable x, the label variable $\alpha_{x,n}$ represents the occurrence of x on line n. For all n such that x appears on line n, the constraint set contains the constraints $\alpha_x \leq \alpha_{x,n}$ and $\alpha_{x,n} \leq \alpha_x$ (the expression x on line n has the same security level as the variable α_x).

From the definitions at the beginning of the code, the constraint set C contains the constraints `config` $\leq \alpha_{rc}$ and `config` $\leq \alpha_{fn}$. The **for** loop introduces a new program counter variable α_{pc1} and the constraints $\alpha_{rc,8} \leq \alpha_i$ (from **int** `i` = `rotateCount`), $\alpha_{i,8} \leq \alpha_i$ (from the `i--` statement), and $\alpha_i \leq \alpha_{pc1}$ (from the loop being executed until a condition on `i` is satisfied). The next two statements generate the constraints $\alpha_{i,9} \sqcup \alpha_{fn,9} \leq \alpha_{nn}$ and $\alpha_{i,10} \sqcup \alpha_{fn,10} \leq \alpha_{on}$. The call to `Runtime.getFile` requires that both the first argument passed and the value returned have the label of the second argument passed in. Therefore, the two calls to `getFile` generate the constraint set

$$\begin{aligned} \{ \quad & \alpha_{nn,11} \leq \texttt{log\_lbl}, \quad \texttt{log\_lbl} \leq \alpha_{nlf}, \\ & \alpha_{on,12} \leq \texttt{log\_lbl}, \quad \texttt{log\_lbl} \leq \alpha_{olf} \quad \} \end{aligned}$$

The **if** statement comparing `oldlogfile` to **null** creates a new program counter variable α_{pc2}, the constraint $\alpha_{pc1} \sqcup \alpha_{olf} \leq \alpha_{pc2}$. Finally, the call to `renameTo` generates

the constraints $\alpha_{olf} \leq \alpha_{nlf}$, as the old log file must be able to flow to the new log file, and $\alpha_{pc2} \leq \alpha_{olf}$, $\alpha_{pc2} \leq \alpha_{nlf}$, as observing if one file has been renamed to another is an observable action that reveals information about the program counter.

5.3.2 The Information-Flow Graph

We now define the information-flow graph as an alternative representation of an information-flow constraint set and show that a cut of the information-flow graph formed from a set of sIMP constraints C corresponds to a set of mediation points that make C satisfiable.

For the rest of this section, we assume that the lattice $\mathcal{L}$ has only two labels: $\top$ and $\bot$ with $\bot \leq \top$. We describe how to extend the cut-based approach to place declassifiers in a general security lattice in Section 5.3.4.

We now define a translation of an information-flow constraint set C into an information flow constraint graph $\mathcal{G}_C$. This $\mathcal{G}_C$ contains dependency information for the label variables and labels that are described by C. Every label variable and lattice element that occurs in C is a vertex in $\mathcal{G}_C$. There is an edge between two vertices in $\mathcal{G}$ if the program permits a flow of information between the program elements that those vertices represent in the graph. For example, if $\alpha \leq \beta \in C$, there are vertices for α and β in $\mathcal{G}_C$ and an edge between them, as the security level of α is constrained to be less than or equal to that of β.

Definition 5.3.3 (Information Flow Graph)**.** Let C be an information-flow constraint set. Let $\mathcal{G}_C$ be the graph with vertex set $V(\mathcal{G}_C) = \mathcal{V} \cup \mathcal{L}$ and, for atoms a, a', $(a, a') \in E(\mathcal{G}_C)$ if $\tau_0 \sqcup \cdots \sqcup a \sqcup \cdots \tau_n \leq a' \in C$.

5.3.2.1 Information-Flow Graph Example

Figure 5.4 shows the information-flow constraints from `logrotate` code as a graph as described in Section 5.3.1.2. The program permits several flows between lattice labels `config` and `log_lbl`. To determine a set of mediation points from a cut of the graph, we allow vertices that correspond to the security values of expressions to be part of the cut. These variables are shaded in grey. Every vertex cut of the graph that separates `config` from `log_lbl` and contains only vertices that correspond to expressions induces a set of mediation points placed in the code. For example, the set of mediation points `rotateCount_8`, `oldName_12`, and `newName_11` separates `config` from `log_lbl`, and corresponds to placing mediation statements at `rotateCount` on line 11, `newName` on line 14, and `oldName` on line 15.

5.3.3 Correspondence of Graph Cuts and Mediation Points

We now show that a vertex cut of the information-flow graph containing only expression variables corresponds with a set of expressions that needs to be mediated. We will show that sIMP constraints have the property that a $(\top, \bot)$ cut of the information-flow graph $\mathcal{G}_C$ corresponds to a placement of mediation statements that fully resolves the flows in the command c.

$(\top, \bot)$ paths in the information-flow graph witness the unsatisfiability of the constraint set C, as they describe a flow between $\top$ and $\bot$. Specifically, C is satisfiable if and only if there is no $(\top, \bot)$-path through the information-flow graph.

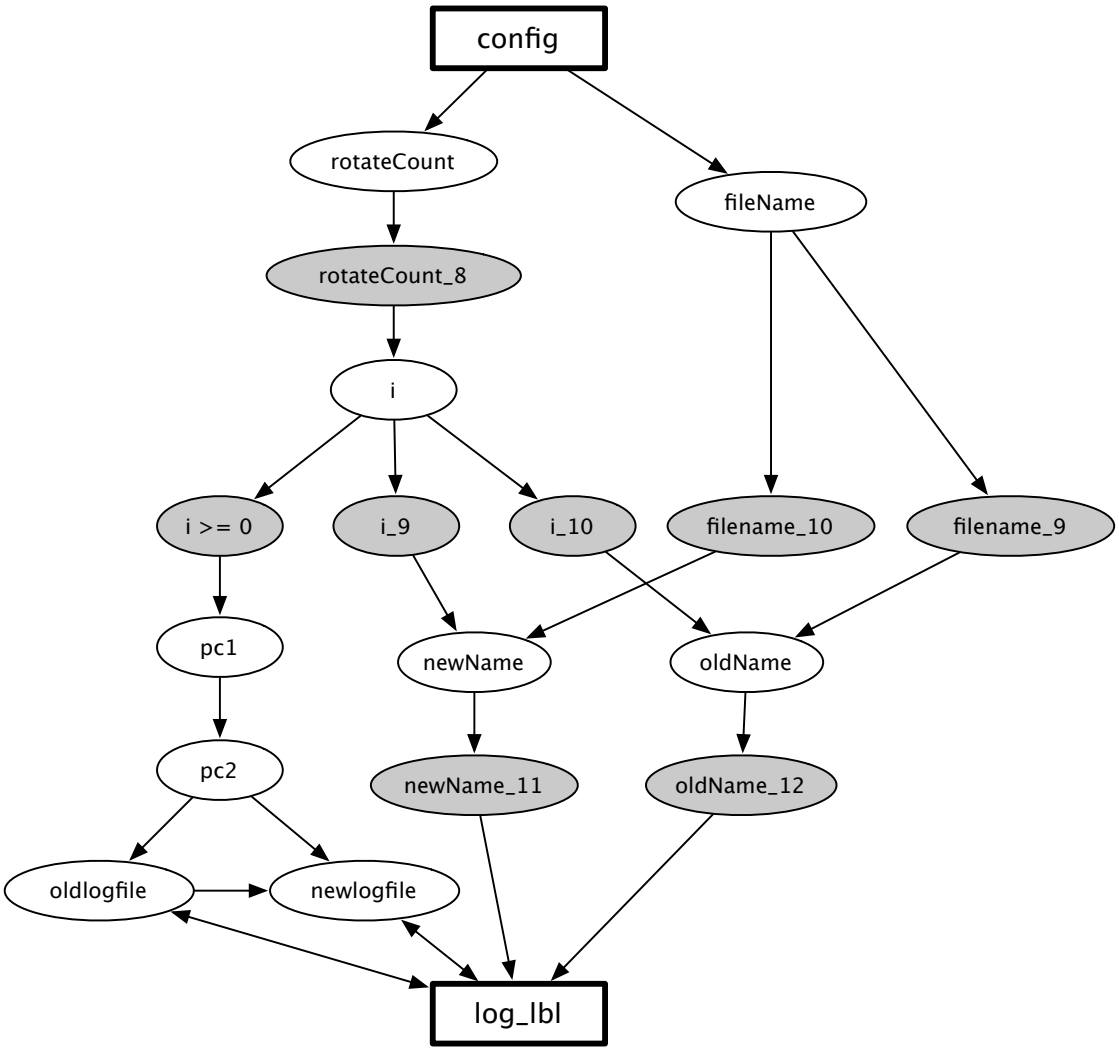

Fig. 5.4. Constraints for `logrotate` code from Figure 5.1 as an information-flow graph. The program permits a flow from `config` to `log_lbl`. Variables that correspond to expressions that can be mediated are shaded in grey. A possible set of mediation points that has minimum size is {`rotateCount_8`, `oldName_12`, `newName_11`}.

Lemma 5.3.4. *Let* $\Gamma; \Delta \vdash c : C$. *The set* C *is satisfiable if and only if there is no* $(\top, \bot)$*-path in* $\mathcal{G}_C$.

Proof. We first show that if there is no $(\top, \bot)$-path in $\mathcal{G}_C$, then C is satisfiable by proving the contrapositive. Let C be unsatisfiable. Therefore, there is a $\tau \leq \bot \in C$ such that the valuation ρ produced by a run of the Rehof-Mogensen solver does not satisfy $\rho(\tau) \leq \bot$. We show by induction on the size of C that there is a path from $\top$ to $\bot$ that includes at least one atom in τ in $\mathcal{G}_C$. The desired result follows.

Induction Hypothesis: Let $\tau = a_0 \sqcup \cdots \sqcup a_n$. If $C = X \cup \{\tau \leq \bot\}$ is unsatisfiable, then there is a path $\top \to \cdots \to a_i \to \bot$ in $\mathcal{G}_C$.

Base Case: $X = \emptyset$: if $\{\tau \leq \bot\}$ is unsatisfiable, then one of the $a_i = \top$ and so there is a $(\top, \bot)$ edge in $\mathcal{G}_C$ by Definition 5.3.3.

Inductive Hypothesis: $X \neq \emptyset$: because $X \cup \{\tau \leq \top\}$ is unsatisfiable, then let ρ_X be the valuation produced by running the Rehof-Mogensen constraint solver on X. This valuation satisfies $\rho(\tau) = \top$. During the run of the Rehof-Mogensen solver, let k be the time at which $\rho^k(\tau) = \bot$ and $\rho^{k+1}(\tau) = \top$, and let $\tau' \leq \beta$ be the constraint considered at time k. This constraint satisfies that $\beta = a_i$ for some i, and $\rho^k(\tau') = \top$. Because $\tau' \leq \beta$ modified the valuation at time k, $\rho^k(\beta) = \bot$. Therefore, the set $C' = (X \setminus \{\tau' \leq \beta\}) \cup \{\tau' \leq \bot\}$ is unsatisfiable.

Let $\tau' = a'_0 \sqcup \cdots \sqcup a'_m$. By the induction hypothesis, there is a path $\top \to \hat{a}_0 \to \cdots \to \hat{a}_n \to a'_i \to \bot$ in $\mathcal{G}_{C'}$ that includes at least a'_i. Because constraints of the form $\tau_0 \leq \beta_0$ that occur in C' are a subset of the constraints in C, each of the edges $(\hat{a}_i, \hat{a}_{i+1}) \in E(\mathcal{G}_C)$. Therefore in $\mathcal{G}_C$, there is the path $\top \to \hat{a}_0 \to \cdots \to \hat{a}_n \to a'_i$.

Because $\tau' \leq \beta \in C$, $\beta = a_i$, a_i occurs in τ, and $\tau \leq \bot \in C$, there is the path $\top \to \hat{a}_0 \to \cdots \to \hat{a}_n \to a_i' \to \beta \to \bot$ in $\mathcal{G}_C$. This proves the induction hypothesis.

We now prove the reverse direction. If C is satisfiable, assume there is a $(\top, \bot)$-path $\top \to \alpha_0 \to \cdots \to \alpha_n \to \bot$. Therefore there is a set of constraints X' that can be given the ordering $\langle \tau_0 \leq \alpha_0, \ldots, \tau_n \leq \alpha_n \rangle$, where $\top$ occurs as an atom in τ_0, C contains a constraint $\tau \leq \bot$, and for $0 \leq i < n$, $\alpha_i \in \text{Vars}(\tau_{i+1})$. However, $X' \cup \{\tau \leq \bot\}$ is an unsatisfiable subset of C: running the Rehof-Mogensen solver over $X' \cup \{\tau \leq \bot\}$ will consider $\tau_i \leq \alpha_i$ at time i and so $\rho^{i+1}(\alpha) = \top$ for all i. Finally, when the algorithm checks $\tau \leq \bot$, the produced valuation ρ has $\rho(\tau) = \top$ as $\alpha_n \in \text{Vars}(\tau)$. This contradicts the satisfiability of C. □

We define an *expression cut* as a $(\top, \bot)$ vertex cut of the information flow graph that only includes label variables of the form $\alpha_{e,p}$.

Definition 5.3.5. Suppose $\Gamma; pc \vdash c : C$. An expression cut of (c, Γ) is a set of expression-position pairs $T = \{(e_0, p_0), \ldots, (e_n, p_n)\}$ such that the set $\{\alpha_{e_0,p_0}, \ldots, \alpha_{e_n,p_n}\}$ is a vertex $(\top, \bot)$ cut set of the graph $\mathcal{G}_C$.

We now define the command $T(c)$, which is the command c with each expression e in the expression cut T replaced by $\mathsf{mediate}(e)$.

Definition 5.3.6. Let T be a set of expression-position pairs (e_i, p_i). Let $T(c)$ represent the command with each e_i at position p_i replaced with $\mathsf{mediate}(e_i)$ at position p_i.

We now show that expression cuts are exactly those sets of expressions which, when mediated, make the generated set of information-flow constraints C satisfiable.

Theorem 5.3.7 (Cut-Mediation Equivalence)**.** *Let T be a set of expression-position pairs, $\Gamma \vdash c : C$, and $\Gamma \vdash T(c) : C'$. Suppose also that C is unsatisfiable. Then T is an expression cut of (c, Γ) if and only if C' is satisfiable.*

Proof. Let $X = \{\tau_1 \leq \tau_2 \in C | (e_i, p_i) \in T \wedge \alpha_{e_i,p_i} \in \text{Vars}(\tau_1)\}$ and let the expression variable α'_{e_i,p_i} be the expression variable for the mediation expression introduced for the expression associated with the label variable α_{e_i,p_i}. Then the constraint set C' generated by the rules from Figure 5.3 on $T(c)$ has the form

$$C' = (C \setminus X) \cup X[\alpha'_{e_i,p_i}/\alpha_{e_i,p_i}]$$

Here $X[\alpha'_{e_i,p_i}/\alpha_{e_i,p_i}]$ represents set resulting from substituting each occurrence of α'_{e_i,p_i} for α_{e_i,p_i} in X. (C' is C with every use of α_{e_i,p_i} on the RHS of a constraint replaced by α'_{e_i,p_i}).

Let the vertex $\alpha_{e_i,p_i} \in \mathcal{G}_C$. For each edge $(\alpha_{e_i,p_i}, A) \in \mathcal{G}_C$, the graph $\mathcal{G}_{C'}$ contains no edge (α_{e_i,p_i}, A), but does contain the edge (α'_{e_i,p_i}, A). Therefore, if there is a path through $\mathcal{G}_C$ that contains the variable α_{e_i,p_i}, then the path on the same variables does not exist in $\mathcal{G}_{C'}$.

We now show that $\{(e_0, p_0), \ldots (e_n, p_n)\}$ is an expression cut of C if and only if C' is satisfiable. We first assume that C' is satisfiable. Because C' is satisfiable, there is no $(\top, \bot)$ path in $\mathcal{G}_{C'}$. By Lemma 5.3.4, because C is unsatisfiable, there is a $(\top, \bot)$-path P in $\mathcal{G}_C$. This path P passes through at least one α_{e_i,p_i} in $\mathcal{G}_C$: otherwise, a path in $\mathcal{G}_{C'}$ from $\top$ to $\bot$ on the vertices with the same labels as those in P would exist. Therefore, T is an expression cut of $\mathcal{G}_C$.

Conversely, if T is an expression cut of $\mathcal{G}_C$, then there is no $(\top, \bot)$-path in $\mathcal{G}_{C'}$: such a path would be a $(\top, \bot)$-path that did not include any of the vertices for expressions in T, which would contradict T being an expression cut. □

5.3.4 Finding Mediation Points For General Lattices

We now describe the more general problem of finding a set of mediation points for an arbitrary lattice. We will show that this problem is an instance to the graph problem of *cut-conjunction*, which has a currently unknown complexity. We thus give an algorithm to solve this problem based on a known NP-complete problem the *hitting set* problem.

5.3.4.1 Complexity of Placement

We first introduce the notion of cut-conjunction for directed graphs [55]. The problem of placing mediation points for a generalized security lattice is an instance of the cut-conjunction problem.

Let $G = (V, E)$ be a directed graph on vertex set V and edge set E. Let $\mathcal{P} \subseteq V \times V$ be an arbitrary family of pairs of vertices in G. A set of edges $E' \subseteq E$ is called a $\mathcal{P}$-cut iff none of the pairs of vertices in $\mathcal{P}$ are connected in $G' = (V, E \setminus E')$. The *cut-conjunction (CC)* problem [55] is the following: given a graph $G = (V, E)$ and $\mathcal{P} \subseteq V \times V$ find a subset of vertices $E' \subseteq E$ that is a minimal $\mathcal{P}$-cut. The cut-conjunction enumeration problem is to enumerate all minimal $\mathcal{P}$-cuts in a graph $G = (V, E)$. The *weighted cut-conjunction (WCC)* problem is the cut conjunction problem, except that a function $f : E \to \mathbb{N}$

specifies edge weights, and the enumerated $\mathcal{P}$-cuts in G are required to have minimum weight.

The following lemma generalizes Lemma 5.3.4 to $\mathcal{P}$-cuts.

Lemma 5.3.8. Let C be an unsatisfiable constraint set over a lattice $\mathcal{L}$ and $\mathcal{G}_C = (V, E)$ be the information-flow graph for C. Let $E' \subseteq E$ be a $\mathcal{P}_{\mathcal{L}}$-cut for $\mathcal{G}_C$. Let $C_{E'}$ be the constraint set generated by the program where the expressions corresponding to the edges in E' have been mediated to $\perp$. The constraint set $C_{E'}$ is satisfiable.

The problem of finding all minimum sized mediation sets for a program in a generalized security lattice is reducible to the weighted cut-conjunction problem. Given an arbitary security lattice $\mathcal{L}$, let $\mathcal{P}$ be the set of all pairs of labels (l_1, l_2) such that $l_1 \not\leq \mathcal{L}$. The solution to the cut conjunction problem for our constructed information-flow graph then corresponds to the expressions that mediate all illegal flows through the program associated with the information-flow graph. However, the complexity of enumerating $\mathcal{P}$-cuts for directed graphs without taking into account edge weights is unknown [55].

5.3.4.2 Generalized Placement Algorithm

We now present a strategy an algorithm to place generalized security mediators that relies on a solution to the generalized hitting set problem.

An instance of the *generalized hitting set (GHS)* problem consists of a set of collections $\{C_1, \cdots, C_n\}$ where each C_i is a collection of subsets of T (i.e., each $C_i = \{S_{i,1}, \cdots, S_{i,k_i}\}$ where $S_{i,j}$ is a subset of T) and a positive integer $k \leq |T|$. The problem is to determine whether there is a subset H of T such that $|H| \leq k$ and for all i such

that $1 \leq i \leq n$ there exists a j such that $S_{i,j} \subseteq H$ (a set in the collection C_i is a subset of H).

We show that *hitting set* is reducible to GHC. As hitting set is an NP-complete problem, this will show that the GHS problem is NP-complete.

An instance of the *hitting set (HS)* problem consists of a collection $\{S_1, S_2, \ldots, S_n\}$, where each S_i is a subset of T, and a positive integer $k \leq |T|$. The problem is to determine whether there is some subset H of T such that

$$|H| \leq k \ \wedge \ \forall i \ (H \cap S_i) \neq \emptyset$$

We claim that GHS is NP-complete by reducing GHS to hitting set. Recall that an instance of HS has three components: T, $\{S_1, \cdots, S_n\}$, and a positive integer $k \leq |T|$ (each S_i is a subset of T). GHS can be easily seen to be in NP: nondeterministically guess an H with $|H| = k$ and verify that H contains an element from each of S_i. Next we will prove that GHS is NP-complete. The reduction will be from HS. Assume that we are given an instance $\mathcal{I}_{HS}$ of HS. We convert $\mathcal{I}_{HS}$ instance of HS to an instance $\mathcal{I}_{GHS}$ of GHS.

- For each $S_i = \{s_{i,1}, \cdots, s_{i,n_i}\}$ define a collection $C_i = \{\{s_{i,1}\}, \cdots, \{s_{i,n_i}\}\}$ (essentially for each element in S_i, C_i has a singleton set)
- $H \subseteq T$ and k are the same as in $\mathcal{I}_{HS}$.

It easy to see that H is a solution to $\mathcal{I}_{GHS}$ if and only if it is also a solution to $\mathcal{I}_{HS}$. Therefore the GHS problem is NP-complete.

```
MediationPoints(G_C, P)
1  Labels ← {l | (l, l') ∈ P}, X ← ∅
2  for each l ∈ Labels
3      T_l ← {l' | (l, l') ∈ P}
4      X_l ← AllMinimumCuts(G_C, l, T_l)
5  S ← MinGHS(X_{l_0}, ..., X_{l_|Labels|})
6  return ExpressionsFromEdgeCut(S)
```

Fig. 5.5. An algorithm for choosing a set of mediation points for a general lattice based on the generalized hitting set problem. The procedure AllMinimumCuts takes an information flow graph $\mathcal{G}_C$ and a set of labels $(l_1, l_2) \in \mathcal{P}$ such that $l_1 \not\leq l_2$ and returns all expressions associated with edge cuts of minimum size in a graph between a source vertex and a set of sink vertices in polynomial time per cutset [80]. The procedure ExpressionsFromEdgeCut converts a set of graph edges into their associated mediation points.

Let MinGHS$(C_1, \ldots, C_n)$ be a procedure that solves the generalized hitting set problem. Figure 5.5 contains an algorithm for placing security mediators for a generalized lattice that relies on an external procedure to solve the generalized hitting set problem. The complexity of this algorithm is therefore dependent on both the number of cuts returned by AllMinimumCuts and the complexity of MinGHS.

It is easy to see that if MediationPoints$(\mathcal{G}_C, \mathcal{P}) = \mathcal{X}$, for all $(l, l') \in \mathcal{P}$, there is no path from l to l' in $\mathcal{G}_C \setminus \mathcal{X}$. Assume there is such a path from l to l': by the definition of a minimum vertex cut, this path intersects at least one vertex in S_l chosen from $\mathcal{X}_l$. This path cannot exist as each vertex in S_l was removed from $\mathcal{G}_C$. By Lemma 5.3.8, mediating the expressions specified in a P-cut results in a satisfiable constraint set.

5.3.5 Lattices in Practice

Our personal experience is that most lattices used to enforce security on applications have a very simple structure. If dynamic labels used to enforce label isolation (our envisioned usage scenario), then the corresponding security lattice will not have a rich security semantics: for example, *client* data will be incompatible with *user* data to prevent data from a user being released to a client. While the actual security policy that these labels represent may be complicated (for example, users allowing discretionary access control to other users as in the decentralized label model [68]), the analysis will not use any assumptions about these in order when using dynamic labels associated with different parts of the program. This necessitates that a mediator be required at each flow from one dynamic label to another.

5.3.6 Modifying the Jif Compiler

To generalize the results presented in this section to the full Java language, we investigated using the constraints generated by the Jif compiler, as Jif is a superset of Java. The main feature of sIMP constraints is that mediating an expression e at position p removes any of the security information affecting the expression label variable $\alpha_{e,p}$. However, the unmodified constraints generated by the Jif compiler do not satisfy cut-mediation equivalence because the security values that the compiler associates with an expression e are affected by both explicit and implicit security information.

The key feature of cut-mediation equivalence in our model is that mediating an expression e at position p removes any the security information affecting the expression label variable $\alpha_{e,p}$. Jif assigns each expression e a *path map* X, which maps symbols

representing security information about e to labels. In the Jif compiler, the value $X(\underline{\mathsf{nv}})$ represents the security label of an expression: this is the security level of observing the value of e. In order to choose a minimum cut that corresponds to a set of mediators, we must choose certain nodes as potential parts of a cutset. However, the $X(\underline{\mathsf{nv}})$ label is also affected by implicit flow information. For example, in the expression **if** `h` **then** `l` $:= 0$, the pathmap $X(\underline{\mathsf{nv}})$ for the expression 0 is set to `high`, although the constant integer 0 itself is not assigned a security label. Therefore, Jif constraints do not have a one-to-one mapping with sIMP constraints.

As an example of the differences between Jif constraint generation and sIMP constraint generation, consider the assignment statement $v := e$. In sIMP, this expression generates two constraints: one ensuring that the flow is allowed explicitly and one ensuring the flow is allowed implicitly. For Jif, let l_v be the declared label of v: Jif will generate the single constraint $\alpha_e \leq L_v$. Because information about the program counter when evaluating e also affects the α_e label, Jif only needs to generate one constraint.

To continue the above example, we kept the previous constraint $\underline{\mathsf{nv}}_e \leq L_v$, but added an additional constraint $pc \leq L_v$, where pc is the label of the program counter at the assignment statement.

5.4 Suggestion Methodology

The information-flow graph construction in Section 5.3 constructs, from program code, a graph for which a cut is equivalent to a placement of mediation points. In this section, we describe how we output sets of mediation points to a programmer using this graph. This section contains a discussion of how we use the framework presented earlier

in this chapter to provide suitable suggestions, as well as how we remove some candidate mediation points to improve the quality of mediators returned. The performance of the methods described in this section is given in Chapter 6.

5.4.1 Minimum Cuts

In our experiments, we used minimum graph cuts to select mediation points. The minimum cut of a graph corresponds to be minimum number of mediation points that need to be inserted into the program. While a minimum sized set of mediation points may not necessarily agree with programmer intent, we believe that a set of minimum size provides a good starting point for understanding how best to mediate the illegal flows in a program. A minimum cut has the advantage of having the smallest possible size among any complete set that a suggestion method could return. This reduces the amount of work a programmer must do when correcting a set of returned suggestions to correspond with their position preferences.

If the programmer wishes to give incentive or decentive to select certain mediation points, then this can be accomplished in the graph framework by modifying the graph cut framework: for example, to make it more costly to select a mediator, the programmer could specify an option that would cause the graph to assign a higher weight to that particular mediator. We will describe the design of such a system in the future work later in this chapter.

5.4.2 Clustering Mediation Points

One difficulty in outputting a set of mediation points is that sets of complete mediation points contain expressions that can be ambiguous in the value being mediated. For example, suppose an expression `l` := `h` + 1 requires mediation (where `l` is a low variable and `h` is a high variable). In this case, both `h` and `h` + 1 are valid expressions to mediate, but selecting the occurrence of `h` as a mediation point obscures the high-security value actually flowing to a low-security sink.

To provide more helpful mediation points, we cluster mediation points by removing expressions which are postdominated in the information-flow graph. That is, if the value of expression e always flows to the value of another expression e', then we do not consider e as a candidate mediation point. In the assignment above, the expression `h` always flows to the expression `h` + 1 and so is postdominated. We therefore do not consider `h` as a possible location for a mediation point.

Because the definition of postdomination relies on the exit node, this must be done for each $l \in \mathcal{L}$. The process of removing postdominated expressions from the set of possible declassifiers is done before computing the maximum network flow between l and its associated super-sink T_l (Figure 5.5). Expressions that are not selected as candidate mediators are given a weight of ∞ in the graph, meaning that they cannot be selected as part of a minimum cut.

To cluster the information-flow graph, we first compute the immediate dominators for each candidate mediation point; this can be performed in $O(n \lg^* n)$ time. However, computing the full dominator relationship from this information can be expensive. Let

I be the immediate domination relationship. To see if a vertex v in the information flow graph is postdominated by another v', we must perform a full reachability check from v to v' along the edges in the immediate dominator relationship (requiring $O(n^2)$ time). However, in most cases where an expression node is dominated, a full reachability check is not necessary: it is sufficient to follow one or two edges. For performance reasons, our implementation only checks the first 10 nodes along the immediate domination relationship to check if a node is postdominated.

5.4.3 Clustering Example

The graph in Figure 5.4 shows the information-flow graph for the code from Figure 5.1. From inspection it is easy to see that there are 8 possible mediation sets. One of `rotateCount_8` and `i` <= 0 must be chosen, one of `filename_10` and `oldName_12` must be chosen, and one of `filename_9` and `newName_11` must be chosen, making $2 \cdot 2 \cdot 2 = 8$ in all.

Observe that `newName_11` and `oldName_12` postdominate `filename_9` and `filename_10` respectively (for example, there is no path through `filename_10` from `config` to `log_lbl` that does not also include `oldName_12`). Because of this, we eliminate `filename_9` and `filename_10` from consideration as mediation points. Note that there is a path from `rotateCount` to `log_lbl` that does not include `i` >= 0, so `rotateCount` is considered as a potential mediation point. There are thus two possible minimum cuts that mediate the flows in this program: the first would be { `rotateCount_8`, `oldName_12`, `newName_11` }, while the second would be { `i` >= 0, `oldName_12`, `newName_11` }. Our suggestion engine would output one of these: later we discuss how existing graph algorithm could be used to make all complete minimum cuts available to the programmer.

5.5 Conclusion

In this chapter we have discussed the design and implementation of a system for automatically selecting mediation points. Programmers can use this technique to help place mediation statements in their program. In the following chapter we investigate in detail the behavior of the mediation placement system on programs and detail the amount of work required by programmers once a set of mediation points has been selected.

Chapter 6

Completing the Reference Monitor

In this chapter we demonstrate the use of the tool as developed in Chapter 5. Our goal is to show how our algorithm for mediator placement can be used to aid in building a reference monitor into existing programs. We show that the suggestion algorithm described in Chapter 5 can be used to select mediation points in legacy programs that satisfy security, function, and programmer requirements. This leads to an effective placement of mediation points with good performance; it reduces the number of program sites that a programmer is required to examine compared to the blame algorithm in 4, while the mediation points selected in programs written in the security-typed language Jif are similar to those placed by expert human programmers.

We first give a model for apply the mediation placement system of Chapter 5 to legacy code. To place mediation statements, the programmer specifies a security policy (the security label on sources and sinks) and placement policy. The programmer can specify whether program locations must or must not appear in a minimum cut, allowing placement policy to be made flexible to fit the application.

We then give metrics regarding our system's performance on several applications programmed both in Jif and Java. In examining Java programs it was our goal to show the performance of our tool and its ability to reduce the amount of lines of code that the programmer would have to examine, compared with the error explanation algorithm

from Chapter 4. In examining Jif programs it is our goal to measure the quality of the automatically placed mediators as compared with those placed manually by expert programmers. This chapter shows the following results:

- For each of our experiments, computing a set of suggestions took under 90 seconds. Our largest code example was an X Server written in Java that contained over 22,000 lines of code, corresponding with over 230,000 information flow constraints. It took 83 seconds for our suggestion method to complete when run on these constraints, returning 176 mediation points. The time required for graph computations were generally dominated by the cost of computing the minimum cut. A pattern in all of our experiments was the small size of the minimum cut relative to the size of the overall graph (for our largest code example, a flow value of 176 vs a graph containing 92,000 nodes and 150,000 edges). This behavior suggests that our suggestion algorithm should scale well on even larger programs.

- For Java programs, we found that automatic selection of mediation points greatly reduced the number of expressions that would have to be examined to resolve a program's information-flow errors. The total number of unique mediation points selected across all suggestions was at 95% less than the individual lines of code that a programmer would have to examine using the algorithm for information-flow blame from Chapter 4. These results are presented in Figure 6.2.

- For Jif programs, we found that our method could be used to place declassifiers in similar locations to those manually selected by expert programmers. We define a set of similarity metrics for placements and show that, over all of our applications,

more than 80% of the automatically placed mediation points were classified as similar to those placed by expert programmers. These results are presented in Figure 6.3. The remaining 20% of mediation points that were placed by our tool generally were selected in a way to reduce the total number of mediation points, whereas the programmer had chose to insert more expressive mediation statements.

6.1 A Model For Specifying Placement Policy

We now expand the minimum-cut based approach of Chapter 5 to take into account more general placement policy when selecting a set of program points that completely mediate the information flows in a legacy program. There are three kinds of placement constraints:

- *Functional requirements* constrain how inserted mediation statements will affect the program's operation. In inserting a mediation statement into a legacy program, functional requirements restrict the selection of mediation points to be in locations where inserting a mediation statements will not alter the program's behavior for authorized users.

- *Security requirements* constrain where mediation statements can be placed based on the information required by the security policy. For example, when placing a reference monitor hook into the program, the user, object being mediated, and the appropriate security-sensitive operation must all be known: a mediation statement cannot be placed in a program location where the object is not in scope. This

security policy sets a security requirement that mediation statements only be placed when an object that can be mediated is in scope.

- *Programmer requirements* constrain where mediation points can be placed based on application-specific placement requirements that the programmer has. For example, a programmer may prefer not to put a mediation statement inside of a procedure that does not have a security semantics.

Programmer Placement Policy Model: Figure 6.1 contains an example of how an application's programmer requirements are specified. The model associates *code elements* with a *placement constraint*. Code elements are specified at one of the following granularities: *file*, *class*, *declaration*, *method*, *block*, *statement*, or *expression*. We group the placement constraint for a code element into five categories: *must*, *should* (with a preference), *don't care*, *should not* (with a preference), and *must not*. The placement constraints *must* and *must not* are a hard limitation on where a mediation statement must or must not be placed. In contrast, the placement categories *should* and *should not* only express a placement preference. If some preferences are stronger than others, the programmer can specify different levels of *should* to indicate that, for example, two classes are both good places for a mediation statement, but he would prefer the first.

Generate Functional and Security Placement Requirements: Both functional and security requirements for placement can be automatically generated from the security policy that is being enforced on the program. Algorithms for generating these placement requirements will be specific to the type of security policy being enforced. For

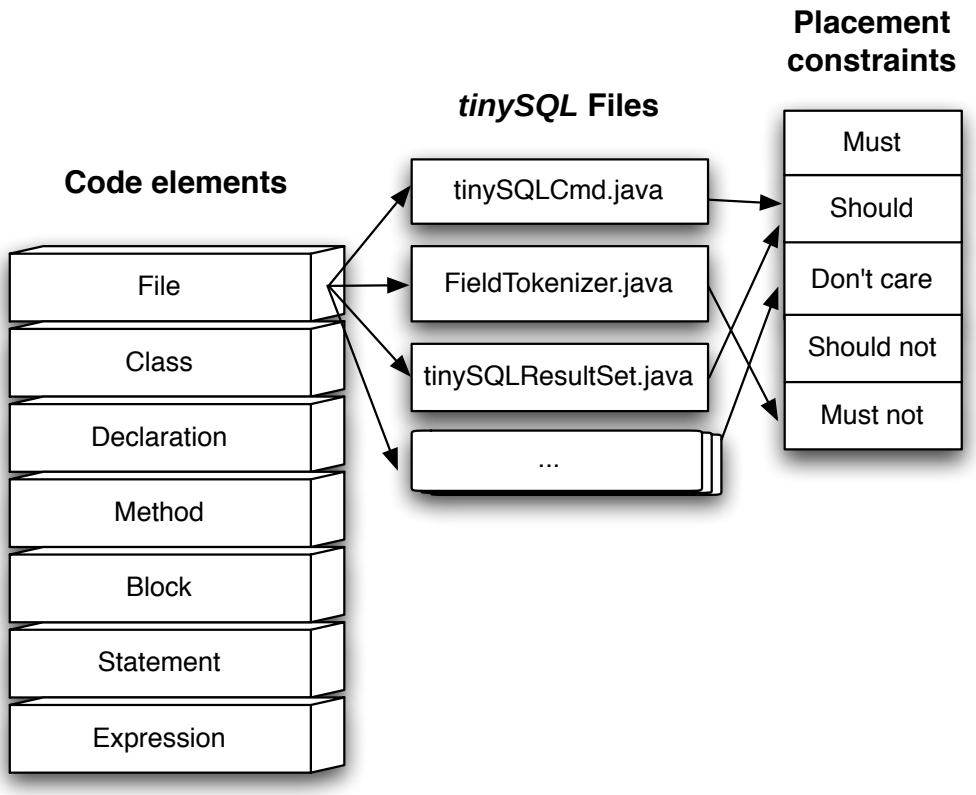

Fig. 6.1. The model for placement policy. Code elements are associated with placement constraints, which indicate hard limitations and preferences on where mediation points should be placed.

example, a security requirement for an integrity policy might be: "untrusted data should have an integrity filter immediately applied to it". The implementation of this security requirement would determine the points at which data from a low-integrity stream enters the system, and mandate mediation points to occur there.

For functional requirements, certain security policies require richer placement policies to avoid causing the program to cease normal legal operation. Consider an authorization policy language and the following code:

```
1 op = getOperationFromUser();
2 if (op == FIRST_OP)
3    performFirstOperation();
4 else if (op == SECOND_OP)
5    performSecondOperation();
```

If a location in the program can eventually take two different paths that have a different set of security-sensitive operations, then this location should not be a candidate mediation point for either set of operations. In this case, line 1 is an example of a program location where during two different runs of the program, two different security sensitive operations may be performed. By inserting a mediation statement for the first security-sensitive operation after line 1, then the runtime behavior of the original program is modified: a user authorized to perform the second security-sensitive operation but not the first will be unable to use the program as written. It is unreasonable to expect a programmer to manually categorize code elements to satisfy this relationship, and so security policies with richer placement policies will require auxiliary analyses to aid placement.

Programmer-Specified Placement Requirements: The programmer can then specify the *programmer placement requirements* for the program, which defines a constraint (possibly with a preference) as to where mediation statements should be placed. For example, a programmer can specify that a function performing encryption should have a declassifier somewhere in it using a *must* constraint.

6.1.1 Modifying the Information-Flow Graph for Placement Requirements

Incorporating placement policy into the suggestion engine requires modifying the information-flow graph before the minimum cut problem is solved. Auxiliary analyses to determine functional and security analysis may be required to eliminate certain locations as candidate mediation points. The "must" and "must not" preferences can be implemented by respectively always including an edge in a cut or making edges associated with the specified program elements uncuttable. The "should" and "should not" preferences are implemented by changing the edge weight: lower preference expressions should be given a higher weight. The exact weights assigned to various levels of "should" and "should not" are not specified by the model and can be defined by during implementation.

To implement "must", "must not", "should" and "should not", we change the edge weights on the minimum cut problem. Let a normal edge ("don't care) is given weight n. A "must" edge is given weight 0 (it is removed from the graph), and a "must not" is given weight ∞, meaning that it cannot be selected during the minimum cut algorithm. A "should" edge is given any weight between 1 and $n - 1$ inclusive, and a

"should not" edge is given any weight above $n+1$ inclusive. The exact weights assigned to a "should" or a "should not" are implementation specific.

Run Placement Mechanism: To run the automated mediation placement tool, we need to assign security semantics to program elements. Many program locations can be automatically assigned a security value: for example, for an integrity policy the program object associated with user input would be classified as `tainted`, and for a secrecy policy the program object associated with program output would be classified as `public`. As described earlier in this thesis, the programmer must manually assign a security label to certain program locations associated with security-relevant sources and sinks.

Once properly annotated, the programmer next uses the automated tool described in Chapter 5 to select mediation points, with constraints and mandates on their selection as specified in the last two sections. After being run, the tool returns for each incomparable security label a suggestion, which consists of points in the program that, when a mediation statement for the expression associated with each position, properly mediates all the information flows in the program. If the selected points are not satisfactory, the programmer can iterate this procedure by modifying the placement policy and re-running the placement tool.

Complete Authorization System: Finally, the programmer must write code to implement the mediation statements with the guidance of the selected mediation points. These mediation statements must satisfy the program's policy. Mediation statements have three principal kinds of semantics, corresponding to three policy actions: declassifier, endorser, or runtime label check.

For example, for an integrity policy the programmer must specify the filtering mechanism that transforms low-integrity data to high-integrity data. It is also possible to check policy at a higher program level: instead of mediating a single expression, we may mediate a block of expressions together. We refer to this process as *hoisting*. For example, if several fields of a protected data structure are always accessed sequentially, then accessing these fields may correspond to a higher-level policy operation [42]. Our tool presently only places mediation points at the level of expressions and statements; any hoisting performed must be done manually by the programmer. Hoisting is also dependent on the program elements that the language allows to be mediated.

6.2 Implementation Required to Enforce a Security Policy

In this section we describe how to implement the model discussed in Section 6.1. There are four main required tasks for the implementation. Two tasks rely on generating the constraints and the runtime system: generating security constraints from source code, making security policy queries accessible from the program. These tasks are security-policy specific and do not need to be repeated when implementing the same category of security policy on a different program.

Next, the programmer must do the work required to make the placements comply with the policy model: implementing any functional and security placement requirements that are specific to a security policy, and transforming the set of suggested mediation points into mediation statements. Only this final task is program-specific.

One of the largest tasks for implementation is writing an analysis that converts source code into information-flow constraints. How information-flow constraints are generated depends on the security policy: for example, a security policy involving only explicit flows does not need to keep track of the program counter as described in Chapter 4. Higher granularity analyses will contain lower false positives but at a higher computational cost.

To write code for mediation statements, there needs to be an interface for the system security policy system accessible at the programming language level. This can take the form of a library call that is implemented as a system call. Flows arising from these queries should be considered as atomic; the security-enforcing compiler is not required to mediate these flows. Certain policy interfaces will mandate security placement requirements: for example, if a policy interface requires an operation, a user, and an object, each of these must be in lexical scope in order to insert a mediation statement at a program point.

To enforce certain functional and security placement requirements, the programmer must write policy-specific source code analyses that perform a source-code analysis and then modify the information-flow graph by making certain edges ineligible or mandatory as part of a cut. These extensions to the suggestion system work as a transformation of the information-flow graph before it is passed to the minimum cut algorithm.

As an example of what is required, we consider the previous example of a policy interface with queries taking a triple of operation, user, and object. A program analysis that prunes the candidate mediation points in the information-flow graph to legal sites must be aware of which variables are in scope, which variables correspond to a user and

object, and which procedures correspond with which operations. This analysis will be run once for each operation. Only in procedures where variables corresponding to a user and object are in scope and the containing procedure performs the correct operation can a mediation statement be placed.

6.2.1 Mediation Statement Implementation

The main work required for a programmer to complete building the reference monitor is the implementation of mediation statements. To convert the suggested set of mediation points into mediation statements, the programmer must determine, for each position in the code chosen as a mediation point, the type of mediation statement (as permitted by the policy interface) that properly resolves the information flow at runtime. This choice is entirely the programmer's decision: mediation points selected are a property of the ways that information flows between security-relevant sources and sinks interacts in the program and do not otherwise have a security semantics. For example, for a mediation point placed in the location of an encryption statement, the programmer could choose a declassifier to indicate that the label on the information must be changed.

Mediation statements corresponding to either a secrecy or an integrity policy will generally have the most interesting semantics. Consider a mediation point associated with a flow from a `Secret` secrecy label to a `Public` secrecy label. The semantics of the declassification statement placed at this point depends on the context of this flow and in what information the program is allowed to release. If the information is not allowed to

be released as-is, the mediation statement will have to sanitize the information as part of a declassification statement.

Programmers have more difficult choices when faced with more general security policies. Consider a program that permits a flow between two runtime labels L_1 and L_2. Depending on the context of the flow, this flow can be resolved through a declassification statement (representing a relabeling of the data from L_1 to L_2), an integrity endorsement (requiring a sanitization filter), or a runtime label check (requiring a runtime check that the security label that instantiates L_1 can flow to the label instantiating L_2). The key difference between which mediation statement is chosen is the runtime semantics of the statement. A declassification will sometimes modify the data being released, an integrity endorsement statement usually applies a filter, while a runtime label check statement usually has no security semantics beyond terminating the execution if the check fails. If the wrong mediation statement is chosen, then the program might not function: if a runtime check is chosen to resolve a flow where L_1 and L_2 are always instantiated with incomparable labels, then the check will always fail and the program will no longer work correctly.

6.3 Experimental Results

In this section, we present the results of running our mediation point placement tool on program code from a variety of Java and Jif applications.

Figure 6.1 gives the number and size of the minimum cut problems for each application, along with running time. We separate programs originally written in Java

(top) from programs originally written in Jif (bottom). For each application, we report the lines of code in the files analyzed, give the number of constraints solved, the number of minimum cut problems that our tool needed to solve, the average size of the information-flow graph for each label l, and the average number of mediation points from the minimum cut. We give the performance of our algorithm by reporting the two factors that had the most effect on running time: total time required to cluster the graphs before performing a minimum cut, and total time required to solve minimum cut problems. Finally, we give the total running time of the analysis. Table 6.3 gives the statistics for each application with respect to the metrics introduced in the previous section. Our results show that, for Jif programs, 80% of the mediation points placed by our tool are done in a location similar to those placed by a human programmer.

6.3.1 Experiment Setup

Our mediation placement algorithm is written in 1,001 lines of C++ code, and our experiments were run on a machine with a 2.3 GHz AMD Operton processor with 3 GB of memory. We used the `Lemon` graph libraries developed for scientific computing to calculate the minimum cut of a graph, but implemented our own dominator computation.

To compare the similarity of mediation points selected by our tool with those manually placed by security-typed language programmers, we ran our analysis on four separate Jif applications: logrotate [52], JPMail (a mail server) [48], Mental Poker (an implementation of a cryptographic protocol) [5], and Civitas (a distributed voting system) [22]. We analyzed the two main functionalities of JPMail (sending and receiving mail). For Civitas, we only examined the main voter loop as many declassification

Application	Lines Analyzed	# Constraints	Min Cut Problems	Avg. Graph Size (vertices)	Avg. Vertices per Minimum Cut	Clustering Time (s)	Cut Time (s)	Total Time (s)
JES	2,407	22,151	1	6,021.00	3.00	0.57	1.06	4.30
Java Card Purse	13,981	48,728	1	8312.00	8.00	0.64	0.50	6.46
tinySQL	12,632	60,909	1	20,683.00	10.00	1.50	2.16	11.83
weirdx	22,308	239,521	2	92802.00	73.00	15.54	21.61	83.74
logrotate	911	6,063	2	1654	3.50	0.11	0.006	1.34
JPMail (reader)	3,934	8,438	59	3151.29	3.31	3.88	0.46	13.37
JPMail (sender)	3,932	14,495	32	3844.69	4.28	4.95	0.12	14.84
Mental Poker	1,578	13,344	1	3553.00	4.00	0.25	0.24	2.21
Civitas (voter)	13,828	67,135	5	17658.00	1.4	7.11	0.62	28.71

Table 6.1. Runtime performance of our mediation placement algorithm.

statements placed throughout the rest of the system enforced information erasure semantics [19], a feature our analysis does not support.

To generate information-flow constraints from each program, we used a context insensitive interprocedural label analysis. The mediation placement technique described in this paper is independent of the specific kind of label analysis, so long as that analysis has the cut-mediation equivalence in Theorem 5.3.7. Our current analysis was sufficient for our examples; we encountered some false positives, but these were easily detected and removed. However, an improved analysis will be necessary in general.

6.3.2 Performance

Table 6.1 contains metrics about the performance of our system. For each application, we report the number of minimum cut problems solved during the analysis, the average size of the information-flow graph among all of the labels, and the amount of time it took to output a minimum cut. Table 6.1 shows the total size of the files required for each of the analyses (column 2). However, as this number also contains whitespace and comments, a more accurate metric for the difficulty of the graph problem is the total number of constraints generated by the analysis (column 3).

There were two major factors that affected the running time of our tool. The first was the number of minimum-cut problems (column 4) that needed to be solved per program; this was equal to the number of labels in its security lattice. The second was the number of mediation points returned as a solution to each minimum cut problem (column 6). The number of minimum cut problems is a multiplicative factor: graph operations such as dominator computation and minimum cut needs to be performed once for each minimum cut problem. The number of vertices returned affects the Minimum cut algorithms are dependent on the number of edges involved in the cut, Programs with a higher number of vertices returned by the minimum cut spent more time performing the minimum cut algorithm.

Automatic mediation placement resolves information flows based on how these flows are represented in the constraint set. The experiments performed here are based off of a context insensitive type-based label analysis to determine the information flows in a program with expressions being the locations for potential mediation points. Our approach uses algorithms with fast running times: domination computation takes essentially linear time, and minimum cut algorithms have a running time that is bounded by the size of the minimum cut. The programs that we analyzed have a small number of mediation points relative to the number of total constraints (less than 1%), allowing the minimum cut algorithm to finish quickly.

6.3.3 Comparison To Previous Work

To evaluate how well our approach reduces the space of placement options, we compared our mediation placement algorithm to an existing mechanism for resolving

Application	Mediation Points		
	Candidate	Error Trace	Min-Cut
JES	5,492	89	3
Java Card Purse	11,540	62	8
tinySQL	14,735	553	10
WeirdX	133,356	1868	176

Table 6.2. Comparison of selected mediation points to information-flow errors for each Java application. The second column gives the total number of candidate mediation points after clustering. The third column gives the number of mediation points highlighted by an information-flow error analysis, while the fourth column gives the number of mediation points selected by our tool. We also report the number of total suggestions output.

Application	Candidate Mediation Points	Total Mediation Points Suggested	Similarity			
			Exact	Same Block	Same Data	Not Similar
logrotate	1,540	9	1	7	1	0
Mental Poker	3,569	7	3	0	1	3
JPMail (reader)	2,434	37	1	15	14	7
JPMail (sender)	3,976	74	2	52	19	1
Civitas (voter)	19,977	9	6	0	2	1

Table 6.3. Similarity Results. For each application, we give the number of mediation points that occur in at least one suggestion and the classification of these mediation points into one of four similarity categories.

information-flow errors. Our results, given in Figure 6.2, show that while the blame algorithm significantly reduces the number of locations that need to be inspected, our tool selects a much smaller set of mediation points.

Initially, a programmer can place a mediation statement at any expression; however, there are too many expressions to determine which of them would be valid security mediators. Recent work [58] proposed a tool to display the reasons that an information-flow constraint becomes unsatisfiable, allowing a programmer to examine *error traces* to find suitable mediation sites.

While this approach narrows down the points in the program that need to be examined, it only reports one error trace per failed information-flow constraint, requiring the programmer run the analysis multiple times to resolve all of the errors per constraint.

6.3.4 Quality of Placed Mediators

To investigate the quality of placed mediation points, we ran our tool on a number of applications originally written in the security-typed language Jif. We define a similarity metric to compare automatically placed mediation points with the mediation points placed by the original application programmers. Our results in Figure 6.3 show that over all Jif applications, over 80% of the selected mediation points were placed in locations that matched one of our similarity metrics. The remaining 20% of mediation points that were placed by our tool generally were selected in a way to reduce the total number of mediation points, whereas the programmer had chose to insert more expressive mediation statements. We examine a specific example of this in Chapter 7.

We classified each selected mediation point as either being *similar* or *not similar*. Those that were classified as similar belonged to one of the following three categories:

- **Exact:** Mediation point that mediates the exact same data in the exact same location as the original application.
- **SameBlock:** Placed location is in the same block of code as the original mediation point.
- **SameData:** Mediates the exact same value as the original mediation point.

Our results show that most mediators are placed in similar locations to those placed by expert programmers. Still, there are many factors used by expert programmers that a minimum cut based approach does not take into account. Should a placed mediator not agree with the intuition of the programmer, the graph-based framework makes it simple to restrict the search through a programmer-specific placement policy.

6.4 False Positives From Static Analysis

Our model is dependent on the security analysis that generates the constraints that are transformed into the information-flow graph. A more powerful constraint generating analysis will result in lower rate of false positives and allow the programmer to automatically suggestion different classes of mediation point; for example, an object-sensitive analysis would allow the suggestion method to suggest mediation at the object level.

As false positives are a limitation of every static analysis, the information-flow constraint set generated by the static analysis may contain flows that cannot be realized at runtime. The suggestion process will place mediation statements that resolve these false positives, meaning that the returned set of mediation points will have to be examined by the programmer to determine whether or not each statement is resolving an actual information leak or simply a false positive. To assist in this, we could run a graph reachability algorithm to generate a path from source to sink that includes any given mediation point.

Chapter 7

Case Study: WeirdX

It took many years to add SELinux mediation statements to the X Server [56]. Without these statements, X implementations were not compatible with the security policy being run on an SELinux system, meaning that the application could not guarantee any policies beyond that incorporated in the program. An X Server retrofitted with mediation statements can enforce security policies at the granularity of the SELinux server. However, because of the size of the X Server, it has taken a long time to insert mediation statements into the system. The claim of this thesis is that with better, automated tools the workload of the programmer could have been lessened. In this section we show how we place mediation statements to enable construction of a reference monitor for an X server using the minimum cut suggestion method.

`weirdx` is an X Server written in Java that implements the full X Window Protocol [87]. The server receives messages from a `Client` class and modifies `Window` objects according to the request. In this chapter we discuss the results of running our analysis on `weirdx`. We examine some of the practical problems involved with finishing the implementation of a reference monitor based on the results of our minimum-cut analysis. Recalling the framework in Section 6.1, we must first specify a security policy. Once we have set a security policy, we identify security and functional requirements, specify any

programmer-specific placement requirements, run the placement mechanism, and then finish implementing the reference monitor.

The main purpose of this chapter is to show how the tool presented in chapters 5 and 6 functions when operated on a large pre-existing codebase. The main result from this chapter is that placing mediation statements in the program can be done quickly and efficiently. We also show some examples of how the tool's minimum cut-based approach influences the returned mediation points. By presenting this chapter we hope to outline the main issues that programmers will encounter when using this automatic placement method to retrofit existing code.

7.0.1 Security Policy

We imposed a simple security policy consisting of a lattice containing two incomparable labels `Client` and `Window`. This signifies that information should not flow between `Client` and `Window` objects, except when explicitly allowed by a mediation statement. This security policy being in place would strictly separate information from the two classes. When data with one label flows to data with another, a mediation statement is necessary to allow this flow. The most common kind of mediation statement for this security policy is a runtime label check: this is used in situations where the information flow should only be allowed in the case where the source label is allowed to flow to the destination label under the underlying system policy.

Declassifications and endorsements would be used in the event that a flow violating system policy is allowed by the program. Under our specified policy, `weirdx` does not contain any of these kinds of flows: each flow highlighted as illegal by the static analysis

is an instance where a flow has occurred between a window of one label and a client of another during a legitimate request. This means that to implement a mediation statement at each mediation point, the programmer can insert a runtime label check. A runtime label check would perform a check between the two labels and, if unsuccessful, terminate the request. Other mediation statements specific to a particular code pattern are also possible: we discuss this below.

While the policy "do not allow information to flow unmediated between windows and their clients" appears simple, its implementation as a runtime label check provides a great deal of flexibility. A runtime label check represents a call to an external policy mechanism, which means that a wide variety of security policies can be enforced at runtime without modifying the program's mediation statements.

A Chinese Wall policy [17] can be enforced at runtime. The Chinese Wall is an integrity policy that restricts a consultant from gathering information from two conflicting sources. Suppose `Bob` is a consultant, and both `Alice` and `Charlie` own windows, but `Bob` should not be allowed to consult with both `Alice` and `Charlie` at the same time to prevent conflicts-of-interest. When the runtime system authorizes a flow between `Bob` (client) and `Alice` (window), it then can block future flows from occurring between `Bob` and `Charlie`. Because all flows from client to window are guarded by mediation points, we can guarantee compliance with the Chinese Wall policy.

Other security policies may require additional modifications to the program. For example, to enforce a Bell-LaPadula policy [14], we can set the structure and flows on runtime labels, but we may need to insert declassification statements in order to allow

data to flow between incomparable labels. This would require the security policy to specify what program elements are allowed to perform a declassification.

7.0.2 Identify Functional and Security Placement Requirements

Simple security policies such as runtime information flow do not require any special security placement requirements, and as such `weirdx` has no functional or security requirements for mediator placement.

7.0.3 Labeling the Code

In order to select mediation points, it is important for the programmer to determine the security-sensitive sources and sinks in a program. Both the client and the window in `weirdx` were identified by unique classes in the code. The main `run` loop receives requests from an instance of the `Client` class, parses them and then invokes the appropriate method on an instance of the `Window` class.

As mentioned above, for richer security policies or more complicated codebases, labeling the code may require more work.

7.0.4 Client to Window Flows

It is important to mediate flows between clients and windows to ensure that a window is not getting any more information about a client than is necessary to complete a request. For example, to enforce integrity of the window, we want to make sure all information entering the window from the client must pass through a mediation statement to enforces an integrity filter.

Our suggestion engine selected 69 mediators to resolve flows originating at the `Client` and flowing to the `Window`. These fell into 3 broad categories, with only a few mediators not involved in one of these categories.

Of these, 22 of them involved a commonly used style for reporting an error: on error, the code checks the `errorReason` field in the `Client` object (set when execution encounters an error). If this field is not equal to zero, the request terminates. This corresponds to an implicit flow: if the request terminates, then the client can observe that there has been no side effect in the window, and so can determine whether or not there was an error. There were two other main classes of selected mediation points: 19 instances of the client's request length were mediated, while 18 instances of the client's request data were mediated. Because of three primary categories of flow from `Client` to `Window`, this case the programmer could write three special mediation procedures to correspond with each of these commonly-occurring flow types.

Figure 7.1 shows a common example of these mediators as selected by our suggestion method. The expressions enclosed by the `mediate` function are the expressions that the minimum cut method selected as a mediation point; the programmer will have to do extra work (as described in Chapter 6) to convert this mediation point into a mediation statement.

The two selected mediation points correspond to the information that necessarily travels from the client to the window in order to complete this request: the window requires `c.data` in order to perform the colormap creation, while returning if `c.errorReason` is not equal to zero is an observable side effect as discussed in the previous paragraph.

```
1     static void reqCreateColormap(Client c) throws IOException {
2         int foo;
3         int n;
4         int mid;
5         int alloc;
6         IO io = c.client;
7         alloc = mediate(c.data);
8         if (alloc != AllocNone && alloc != AllocAll) {
9             c.errorValue = alloc;
10            c.errorReason = 2;
11        }
12        mid = io.readInt();
13        foo = io.readInt();
14        Window w = c.lookupWindow(foo);
15        if (c.errorReason == 0 && w == null) {
16            c.errorValue = foo;
17            c.errorReason = 3;
18        }
19        foo = io.readInt();
20        c.length -= 4;
21        if (mediate(c.errorReason) != 0) { return; }
22    ...
23 }
```

Fig. 7.1. A typical pair of mediation points from `Client` to `Window`.

There were only a few other types of mediators selected by the placement system, but these did not correspond with any broader type of pattern. For example, 6 mediators were involved with setting and retrieving a cached instance of a `Drawable` object. By combining mediation statements for the three main categories of flow, the programmer would have to write 28 different mediation statements: 3 corresponding to the common flows and 25 corresponding to other program elements that were required to be mediated.

7.0.5 Window to Client Flows

In the X Server, it is important to mediate flows from windows to clients in order that clients do not receive information about windows that they are not privileged to view. This corresponds to a standard secrecy policy: a user of the server should not be authorized to view someone else's windows unless they have been granted specific access to them in the security policy.

There were 77 expressions selected as mediation points from the `Window` to `Client`. The most common expression mediated was the `attr` field of a `Window` class, a bit array represented as an integer that contained facts about the window: this occurred 22 times. Two other common expressions occur throughout the set of suggestions: 10 occurrences of a `Window`'s `borderWidth` field, and 7 occurrences of a `Window`'s `eventMask`, an integer mask representing a sent event. The remaining suggested mediation points contained unrelated expressions, primarily related to display attributes (for example, character widths, graphics objects, window alpha settings). The reason that these data flows were less uniform than those from `Client` to `Window` is that different requests from the `Window` revealed different things to the `Client`, whereas almost every flow from `Client` to `Window`

only required the fields of the client's request (request type, request length, and request data).

We highlight a few different selected mediation points here to give a feeling for the kinds of program elements that were chosen as requiring mediation. Figure 7.2 contains code showing a typical example of how the `attr` field of a `Window` object is sent to the client. Based on some directive from the client, information about the window's attribute field is sent to the client through an implicit flow. In Figure 7.2, the analysis has determined that the client can view when the `child.sendEvent` and `child.unrealizeTree` procedures have been invoked. Therefore, the fields of `Window` that causes these procedures to be invoked or not must be mediated.

Two sections of code containing identical implementations returned two identical sets of mediation points. Two classes, `DDXWindowImp` and `DDXWindowImpSwing` implement similar functionality (displaying windows to the screen), with the second using Java's swing toolkit. Both classes contain a `setLocation` function with identical logic, and our minimum cut algorithm selected the same 7 mediation points in both. The logic for the code is given in Figure 7.3.

The influence of the choice of minimum cut can be seen by examining the placed mediators. The fields and variables that require mediation are `window.attr`, `x`, and `y`: to determine this, the programmer can inspect the placed mediation statements and the program's callgraph. There is no way for the suggestion engine to mediate every use of a variable or formal with one mediation statement, and so the engine has suggested mediators around each use of `x` and `y`. Note that it does not mediate `window.attr` directly,

```
1   private void unmapSubwindows(Client c) throws IOException {
2       if (firstChild == null) return;
3       boolean wasRealized = (attr & realized) != 0;
4       boolean wasViewable = (attr & viewable) != 0;
5       boolean parentNotify = this.subSend();
6       boolean anyMarked = false;
7       for (Window child = lastChild; child != null; child = child.prevSib) {
8           if ((mediate(child.attr & mapped) != 0) {
9               if (parentNotify || child.strSend()) {
10                  c.cevent.mkUnmapNotify(child.id, 0);
11                  child.sendEvent(c.cevent, 1, null);
12              }
13              child.attr &= ~mapped;
14              if ((mediate(child.attr & realized)) != 0) { child.unrealizeTree(false); }
15          }
16      }
17      if (wasRealized) { Window.restructured(); }
18  }
```

Fig. 7.2. Two mediators resolving a flow from `Window` to `Client`. Based on information from the windows, information about a window's attributes is leaked implicitly to the programmer.

instead mediating the larger expressions that it occurs in; these expressions also contain instances of `x` and `y`.

One problem with this kind of mediation placement is that, though `x` and `y` are the values that require mediation, it is unclear exactly what values these take on. It would be possible to improve these mediators by, rather than mediating these values in the `setLocation` procedure, instead mediating the values that the function is called with. However, because the procedure `setLocation` is called 19 times throughout `weirdx` (including 3 times within both `DDXWindowImp` and `DDXWindowImpSwing`), this would result in a solution that does not result in a minimum cut. A "should-not" or "must-not" constraint could be added to force mediation points to not be placed in the `setLocation` procedure; this would result in more mediation points being placed, presumably in the callers of `setLocation`.

Given the found placement, the programmer would have to write 40 mediation statements: 1 corresponding to the 22 instances of the `attr`, 1 corresponding to the 10 instances of `borderWidth`, and 1 corresponding to the 7 instances of `eventMask`, and 37 corresponding to the remaining flows.

7.0.6 Limitations

The choice of expressions as a mediation points for `weirdx` resulted sets of mediation points that are overly verbose. Most of the flows from the client to the window could be resolved by one mediation check of the client label to the window label, after which all of these flows would be permitted. To allow the suggestion engine to select this kind of mediator within the framework from Chapter 5, the static analysis would need

```
1   public void setLocation(int x, int y) {
2       Point p = this.getLocation();
3       if (mediate(p.x == x && p.y == y)) return;
4       if (window != null && window.parent != null) {
5           int orgx = mediate(p.x - window.parent.borderWidth + window.borderWidth)
6           int orgy = mediate(p.y - window.parent.borderWidth + window.borderWidth)
7           int bitgrabity = window.attr & 15 << 8;
8           if (orgx < 0 || orgy < 0) {
9               if (mediate(orgx < 0 && orgx < x && bitgrabity != 3 << 8 &&
10                                                  bitgrabity != 6 << 8 && bitgrabity != 9 << 8))
11                  orgx = orgx * -1 - mediate(x) * -1;
12                  exposed.x += orgx;
13                  exposed.width -= orgx;
14                  if (exposed.width < 0) exposed.width = 0;
15              }
16              if (mediate(orgy < 0 && orgy < y && bitgrabity != 7 << 8 &&
17                                                  bitgrabity != 8 << 8 && bitgrabity != 9 << 8))
18                  orgy = orgy * -1 - mediate(y) * -1;
19              exposed.y += orgy;
20                  exposed.height -= orgy;
21                  if (exposed.height < 0) {
22                      exposed.width = 0;
23                      exposed.height = 0;
24                  }
25              }
26          }
27      }
28      if (window.screen.windowmode != WeirdX.InBrowser && window.hasFrame()) {
29          window.getFrame().setLocation(
30                 window.origin.x - window.borderWidth + window.parent.borderWidth,
31                 window.origin.y - window.borderWidth + window.parent.borderWidth);
32      } else {
33          super.setLocation(x, y);
34      }
35  }
```

Fig. 7.3. Example in WeirdX code showing how minimum cut dictates placement.

to generate information-flow constraints that provide at each point a variable that would serve as a candidate mediation point: cutting this variable would represent the effect of mediating two labels at each future point in the program's execution. This may require an auxiliary dominator analysis to determine which points in the program always follow one another.

More expressive label analyses can give stronger guarantees to the system. For example, an analysis that distinguishes between different instances of the same class (an *object-sensitive* analysis) could guarantee that, for example, the `weirdx` system does not allow information to flow between two different instances of a `Window` class, the program representation of an X window on the screen. With the current analysis, such guarantees need to be made by appealing to the program's structure. For `weirdx`, we know such flows cannot exist by examining how Window instances are created and managed in the program.

Chapter 8

Conclusion

While it would be desirable for programs to enforce security policies, we have seen that currently there are several problems with retrofitting programs to support this goal: the difficulty of understanding security errors and the amount of manual work necessary to audit a program's security-relevant flows. In this thesis we have shown that these problems can be overcome through a combination of static analysis and graph techniques, and have presented a model for using a minimum-cut based mediator point placement algorithm to comply with policy and application-specific placement requirements.

Future work arising from this line of research will investigate what modifications to the process are necessary to support a wider variety of security policies beyond information flow. It is important to understand what dimensions of the static analysis best reduce the burden of the programer. Additionally, we envision automatically hoisting returned mediation points as in recent work for placing reference monitor queries for security-sensitive operations [42].

References

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

[2] Martín Abadi and Cédric Fournet. Access control based on execution history. In *Proceedings of the Network and Distributed System Security Symposium*, 2003.

[3] J. P. Anderson. Computer security technology planning study, volume II. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, October 1972.

[4] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science, 16-19 June 2001, Boston, Massachusetts, USA*, 2001.

[5] Aslan Askarov and Andrei Sabelfeld. Secure implementation of cryptographic protocols: A case study of mutual distrust. In *ESORICS*, LNCS, Milan, Italy, September 2005. Springer-Verlag.

[6] Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.

[7] Lennart Augustsson. Cayenne - a language with dependent types. In *In International Conference on Functional Programming*, pages 239–250. ACM Press, 1998.

[8] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information System Security*, 3(3):186–205, 2000.

[9] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL '03*, pages 97–105, 2003.

[10] Gilles Barthe, David Pichardie, and Tamara Rezk. Programming languages and systems, 16th european symposium on programming, esop 2007. In *ESOP*, pages 125–140, 2007.

[11] Gilles Barthe and Tamara Rezk. Non-interference for a jvm-like language. In *Proceedings of TLDI'05: 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 103–112, 2005.

[12] Gilles Barthe, Tamara Rezk, and David Naumann. Deriving an information flow checker and certifying compiler for java. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 230–242, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM.

[14] David E. Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, The MITRE Corporation, March 1976.

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

[16] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03*, pages 196–207. ACM Press, 2003.

[17] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[18] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[19] Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *Computer Security Foundations*, pages 98–111, June 2008.

[20] Stephen Chong, K. Vikram, and Andrew C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th USENIX Security Symposium.*, 2007.

[21] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *J. Comput. Secur.*, 15(3):321–371, 2007.

[22] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *IEEE Symposium on Security and Privacy*, 2008.

[23] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *ESOP*, pages 520–535, 2007.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd edition.* MIT Press, McGraw-Hill Book Company, 2000.

[25] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[26] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From hotjava to netscape and beyond. In *IEEE Symposium on Security and Privacy*, pages 190–200, 1996.

[27] Zhenyue Deng and Geoffrey Smith. Type inference and informative error reporting for secure information flow. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 543–548, New York, NY, USA, 2006. ACM Press.

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

[29] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[30] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*, pages 167–182, 1998.

[31] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 2005 ACM Symposium on Operating System Principles*, October 2005.

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

[33] Ulfar Erlingsson and Fred B. Schneider. Irm enforcement of java stack inspection. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 246, Washington, DC, USA, 2000. IEEE Computer Society.

[34] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *In IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[35] Manuel Fahndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *In Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[36] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. *SIGPLAN Not.*, 31(5):23–32, 1996.

[37] Jeffrey S. Foster, Manuel Fhndrich, and Alexander Aiken. A theory of type qualifiers. In *In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI99*, 1999.

[38] Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. Syst.*, 25(3):360–399, 2003.

[39] Timothy Fraser, Nick L. Petroni, Jr., and William A. Arbaugh. Applying flow-sensitive cqual to verify minix authorization check placement. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 3–6, New York, NY, USA, 2006. ACM.

[40] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Automatic placement of authorization hooks in the linux security modules framework. In *CCS'05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 330–339, Alexandria, Virginia, USA, November 2005. ACM Press, New York, NY, USA. http://doi.acm.org/10.1145/1102120.1102164.

[41] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Retrofitting legacy code for authorization policy enforcement. In *SP'06*, pages 214–229, Oakland, California, USA, May 2006. IEEE Computer Society Press. http://doi.ieeecomputersociety.org/10.1109/SP.2006.34.

[42] Vinod Ganapathy, David King, Trent Jaeger, and Somesh Jha. Mining security sensitive operations in legacy code using concept analysis. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, page TBD, Minneapolis, Minnesota, USA, May 2007. IEEE Computer Society Press, Los Alamitos, California, USA. To Appear.

[43] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 1982.

[44] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006.

[45] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for Java based on path conditions in dependence graphs. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, 2006.

[46] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 291–301, 2002.

[47] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[48] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. Understanding practical application development in security-typed languages. In *ACSAC*, 2006.

[49] Boniface Hicks, Dave King, and Patrick McDaniel. Jifclipse: Development tools for security-typed applications. In *PLAS '07*, 2007.

[50] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification: High-level policy for a security-typed language. In *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '06)*, Ottawa, Canada, June 10 2006. ACM Press.

[51] Boniface Hicks, Timothy Misiak, and Patrick McDaniel. Channels: Runtime system infrastructure for security-typed languages. In *23rd Annual Computer Security Applications Conference (ACSAC)*, Miami, Fl, December 2007.

[52] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference*, Santa Clara, CA, USA, June 2007.

[53] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[54] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[55] Leonid Khachiyan, Endre Boros, Khaled Elbassioni, Vladimir Gurvich, and Kazuhisa Makino. Enumerating disjunctions and conjunctions of paths and cuts in reliability theory. *Discrete Appl. Math.*, 155(2), 2007.

[56] D. Kilpatrick, W. Salamon, and C. Vance. Securing the X Window system with SELinux. Technical Report 03-006, NAI Labs, March 2003.

[57] D. H. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit Flows: Can't live with 'em, can't live without 'em. In *Proceedings of Fourth International Conference on Information Systems Security*, December 2008. To appear.

[58] Dave King, Trent Jaeger, Somesh Jha, and Sanjit A. Seshia. Effective blame for information-flow violations. In *FSE 2008*, November 2008.

[59] Maxwell Krohn and Eran Tromer. Noninterference for a practical difc-based operating system. In *IEEE Symposium on Security and Privacy*, 2009.

[60] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Symposium of Operating System Principles*, October 2007.

[61] Butler W. Lampson. A note on the confinement problem. *Communication of the ACM*, 16(10):613–615, October 1973.

[62] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[63] Benjamin Livshits, Aditya V. Nori, and Sriram K. Rajamani andAnindya Banerjee. Merlin: Specification inference for explicit information flow problems. In *ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI), Association for Computing Machinery, Inc.*, June 2009.

[64] Pasquale Malacaria. Assessing security threats of looping constructs. In *POPL '07*, pages 225–235. ACM Press, 2007.

[65] Pasquale Malacaria and Han Chen. Quantitative analysis of leakage for multi-threaded programs. In *PLAS '07*, New York, NY, USA, 2007. ACM.

[66] Heiko Mantel and Andrei Sabelfeld. A generic approach to the security of multi-threaded programs. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001)*, pages 126–, 2001.

[67] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *PLDI '08*, 2008.

[68] Myers and Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security & Privacy*, 1998.

[69] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL '99*, pages 228–241, January 1999.

[70] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *In Proc. 17th ACM Symp. on Operating System Principles (SOSP*, pages 129–142. ACM Press, 1997.

[71] Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java + information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001.

[72] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[73] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[74] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 229–243, 1996.

[75] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the '98 Conference on Programming Language Design and Implementation*, pages 333–344. ACM Press, 1998.

[76] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. In *In Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL*, 1997.

[77] Marco Pistoia, Anindya Banerjee, and David A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 149–163, Washington, DC, USA, 2007. IEEE Computer Society.

[78] François Pottier and Vincent Simonet. Information flow inference for ML. In *POPL '02*, pages 319–330, New York, NY, USA, 2002. ACM Press.

[79] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Existential label flow inference via CFL reachability. In Kwangkeun Yi, editor, *Proceedings of the Static Analysis Symposium (SAS)*, volume 4134 of *Lecture Notes in Computer Science*, pages 88–106. Springer-Verlag, August 2006.

[80] J. Scott Provan and D. R. Shier. A paradigm for listing (s, t)-cuts in graphs. *Algorithmica*, 15(4):351–372, 1996.

[81] Jakob Rehof and Torben AE. Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2–3):191–221, 1999.

[82] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[83] Torsten Robschink and Gregor Snelting. Efficient path conditions in dependence graphs. In *ICSE*, pages 478–488, 2002.

[84] Sandra Rueda, Dave King, and Trent Jaeger. Verifying compliance of trusted programs. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 321–334, 2008.

[85] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *ISSS*, pages 174–191, 2003.

[86] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 2007, to appear.

[87] Robert W. Scheifler. 1994.

[88] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[89] C. E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27, 1948.

[90] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1997. ACM.

[91] Micha Sharir and Amir Pnueli. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234, Englewood Cliffs, NJ, 1981. Prentice Hall.

[92] Stephen Smalley. Configuring the SELinux Policy. Technical Report NAI-02-007, National Security Agency, 2005.

[93] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing SELinux as a linux security module. Technical Report 01-043, NAI Labs, 2001.

[94] Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, pages 355–364, 1998.

[95] Scott F. Smith and Mark Thober. Refactoring programs to secure information flows. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 75–84, New York, NY, USA, 2006. ACM Press.

[96] Scott F. Smith and Mark Thober. Improving usability of information flow security in java. In *PLAS '07*, pages 11–20. ACM Press, 2007.

[97] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 372–382, 2006.

[98] Q. Sun, A. Banerjee, and D. Naumann. Modular and constraint-based information flow inference for an object-oriented language, 2004.

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

[100] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, volume 00, page 179. IEEE Computer Society, 2004.

[101] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3), 1996.

[102] Dennis M. Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. In *CSFW*, pages 34–43, 1998.

[103] Dan S. Wallach and Edward W. Felten. Understanding java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 52–63, 1998.

[104] E. Walsh. Integrating X.Org with Security-Enhanced Linux. In *Security-Enhanced Linux Workshop*, March 2007.

[105] Mitchell Wand. Finding the source of type errors. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 38–43, New York, NY, USA, 1986. ACM Press.

[106] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 32–41, 2007.

[107] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 171–180, 2008.

[108] Mark Weiser. Program slicing. In *ICSE '81*, pages 439–449. IEEE Press, 1981.

[109] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.

[110] Hongwei Xi and Dana Scott. Dependent types in practical programming. In *In Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227. ACM Press, 1998.

[111] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *USENIX Symposium on Operating System Design and Implementation*, November 2006.

[112] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, Berkeley, CA, USA, 2002.

Vita

David Holliday King is currently working as a developer for Rackspace Email & Apps in Blacksburg, VA.

Education

- The Pennsylvania State University (2003-2009). PhD Candidate in Computer Science.
- University of Illinois at Urbana-Champaign (1999-2003). BS in Computer Science (2003), BS in Mathematics (2003).

Publications

- **Implicit Flows: Can't Live With 'Em, Can't Live Without 'Em.** Dave King, Boniface Hicks, Trent Jaeger, Michael Hicks. ICISS 2008.
- **Effective Blame for Information-Flow Violations.** Dave King, Trent Jaeger, Somesh Jha, Sanjit A. Seshia. FSE 2008.
- **Verifying the Compliance of Trusted Programs.** Sandra Rueda, Dave King, Trent Jaeger. USENIX '08.
- **Jifclipse: Development Tools for Security-Typed Languages.** Boniface Hicks, Dave King, and Patrick McDaniel. PLAS '07.
- **Mining Security-Sensitive Operations in Legacy Code using Concept Analysis.** Vinod Ganapathy, Dave King, Trent Jaeger, Somesh Jha. ICSE '07.
- **Leveraging IPsec for Mandatory Access Control Across Systems.** Trent Jaeger, David H. King, Kevin Butler, Serge Hallyn, Joy Latten, and Xiaolan Zhang. SecureComm '06.
- **Trusted Declassification: High-level policy for a security-typed language.** Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. PLAS '06.