Automating Security Mediation Placement

Dave King[1], Susmit Jha[2], Divya Muthukumaran[1], Trent Jaeger[1], Somesh Jha[3], and Sanjit A. Seshia[2]

[1] Pennsylvania State University*
[2] University of California, Berkeley**
[3] University of Wisconsin

Abstract. We present a framework that automatically produces suggestions to resolve type errors in security-typed programs, enabling legacy code to be retrofit with comprehensive security policy mediation. Resolving such type errors requires selecting a placement of *mediation statements* that implement runtime security decisions, such as declassifiers and authorization checks. Manually placing mediation statements in legacy code can be difficult, as there may be several, interacting type errors. In this paper, we solve this problem by constructing a graph that has the property that a vertex cut is equivalent to the points at which mediation statements can be inserted to allow the program to satisfy the type system. We build a framework that produces *suggestions* that are minimum cuts of this graph, and the framework can be customized to find suggestions that satisfy programmer requirements. Our framework implementation for Java programs computes suggestions for 20,000 line programs in less than 100 seconds, reduces the number of locations a programmer must consider by 90%, and selects suggestions similar to those proposed by expert programmers 80% of the time.

1 Introduction

Security-typed languages [20,22] 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 in a medical data system is assigned a doctor label or another label until runtime, requiring a runtime authorization check.

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

* Research was supported by NSF grant CNS-0905343.
** Research was supported in part by NSF grants CNS-0627734 and CNS-0644436, and an Alfred P. Sloan Research Fellowship.

consideration to avoid errors. Automatic tools can identify missing mediation statements [8,20,32], but even after the errors have been identified, reaching a consensus on manual placement often takes a long time (e.g., for Jif programs [12] and the X Window Server [28]). 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.

In this paper, we present a method for automatic identification of mediation points in legacy programs that is based on a graph cut approach. A mediation point is a location where a mediation statement can be placed. We were inspired to investigate mediation point placement as a cut problem due to recent work assigning a quantitative measure of leaked information in a program by solving a maximum-flow graph problem [17]. By solving a cut problem, the dual of the maximum-flow problem, we present the programmer with options to insert mediation statements into the program. 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 such that a cut of any information flow graph generated from a program using that type system is equivalent to a placement of mediation statements in the program. We use existing graph algorithms to output each equivalent cut of the graph, thereby providing the user with a set of legal placement suggestions to assess, reducing their effort significantly. We make the following contributions in this paper:

- We define a transformation from a set of information-flow constraints for a program into an information-flow graph, such 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 a framework that computes *suggestions* for mediation points based on finding 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 such an information-flow graph. We also describe how to cluster expressions to prevent many redundant suggestions from being output to the programmer.
- Our framework implementation computes suggestions for Java programs of more than 20K SLOC in less than 100 seconds. In addition, our results show that our suggestions reduce the number of locations that would be required for a programmer to examine given current tools for finding security-typed language errors by approximately 90%, and in programs originally written in a security-typed language, more than 80% of the selected mediation points were classified as similar to those placed by human programmers.

The graph-cut approach presented in this paper provides a framework for programmers to solve practical placement concerns, ensure that solutions resolve all conflicts, contain no redundant mediators, and can account for performance and style considerations.

Related Work: McCamant and Ernst [17] dynamically measure the quantitative information flow that a program leaks by solving a maximum flow problem. However, the corresponding minimum cut of the program's flow may not be the only suitable location to use as a mediation statement. Our work is aimed at providing a static mechanism for determining the mediation points that resolve the type system conflicts for a program. Programmers are currently using security-typed languages to build secure systems [2,13], but these systems have not been used to retrofit existing programs for security. There has also been a recent line of work in manually adding authorization checks to code in applications, such as Linux Security Modules [31], X Windows server [28] and `dbus` [29]. A variety of research aims to enforce type safety guarantees for C code [3,21], but we aim to resolve type errors. Program slicing [26] and type-based analyses [5] have been used to find information-flow errors. The principal advantage of our framework is that viewing the problem as graph cut enables us to find placements that achieve complete mediation that accounts for all of the errors in a single computation. However, the scope of work presented here applies to security types. Generalization is future work.

The remainder of the paper is structured as follows: In Section 2, we survey some problems related to placing mediation statements in code. We provide background about security type-checking in Section 3. In Section 4, we describe how to transform information-flow constraints into a graph that has the property that a cut of the graph is equivalent to a set of statements that mediate the corresponding program's illegal flows. In Section 5, we outline the design of our framework, which outputs mediation suggestions from this graph. In Section 6, we give the results of experiments, where we apply our tool to place mediators in eight different programs.

2 Overview

In this section, we introduce some of the challenges in 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* [9]: 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 [24], 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 violate 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 [13], a version of `logrotate` has been written in the security-typed language Jif [20], 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, it is not clear whether or not `logrotate` violates the secrecy and integrity guarantees of the system: it is possible that it reveals configuration data through viewing the results of log rotation or that it compromises log data by allowing a user of `logrotate` to modify log file data that she does not have access to. We highlight three individual flows from the configuration file to the rotated logs in Figure 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 rotate logs containing evidence of attacks on the system.
- 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 in Figure 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 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 labels. Next, the programmer must examine each line of code contributing to errors that result. She can use automated methods to identify possible causes of information-flow errors [15]. 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 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

```
label config, log_lbl, LogInfo[{config}]{config} log;
String{config} filename = log.getFilename(logNum);
int{config} rotateCount = log.getRotateCount();
File[{log_lbl}]{log_lbl} disposeFile =
  Runtime.getFile(filename+"."+(rotateCount+1),log_lbl);
File[{log_lbl}]{log_lbl} newlogfile, oldlogfile = null;
// rename messages.n to messages.n + 1
for (int i = mediate(rotateCount,log_lbl) ; i >= 0; i--) {
   String newName = filename + "." + i;
   String oldName = filename + "." + (i-1);
   newlogfile = Runtime.getFile( mediate(newName,log_lbl) ,
                                    log_lbl);
   oldlogfile = Runtime.getFile( mediate(oldName,log_lbl) ,
                                    log_lbl);
   if (oldlogfile != null)
      oldlogfile.renameTo(newlogfile);
}
```

Fig. 1. Example from `logrotate` that performs rotation of log files shown with mediation statements inserted

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 [24]. 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.

3 Background on Information-Flow Checking

Security-typed languages [20,22] 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 greater than or equal to the program counter.

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 [19]. 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.

4 Constraint Methodology

In this section, we show how to generate information-flow constraints so that finding a set of mediation statements can be solved as a graph-cut problem. We introduce sIMP, a constraint-based type system for IMP, a simple imperative language [30]. The IMP language contains conditionals, variable assignment, and while loops, and is presented as a simple foundational language. 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.* [27], 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 [6,7,11,25], 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 label variable or lattice element and τ is a join of atoms, see the formal definitions in the next section), 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 4.4.

4.1 A Constraint-Based Type System for 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[1]:

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

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 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 its presentation.

Label Constraints: To enforce information-flow security on IMP, we define a *constraint-based type system* that determines *label constraints* from a command c and 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.

$$\begin{array}{lllllll}
\text{Label Variables} & \alpha & ::= \alpha, \beta, \ldots \in \mathcal{V} & \text{Security Labels} & l & ::= l \in \mathcal{L} \\
\text{Atoms} & a & ::= \alpha \mid l & \text{Label Joins} & \tau & ::= a \mid a \sqcup \tau \\
\text{Constraints} & \xi & ::= \tau \leq a & & &
\end{array}$$

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 states that an expression has not been explicitly been labeled by the programmer. A label l represents an expression that has a predefined security semantics defined by the lattice $\mathcal{L}$: for example, a key used for encryption that has been read from a file would be given a Secret security label that would prevent it from being leaked to security labels in the lattice that it dominates, including Secret.

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

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 as follows:

Expressions

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

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

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

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

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

Commands

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

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

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

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

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

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 [4]. 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.

Constraint Example: We now investigate the information-flow constraints associated with the main loop in Figure 1 by building the information-flow constraint set C. The constraints 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

$$\{\ \alpha_{nn,11} \leq \texttt{log\_lbl},\ \texttt{log\_lbl} \leq \alpha_{nlf},\\ \alpha_{on,13} \leq \texttt{log\_lbl},\ \texttt{log\_lbl} \leq \alpha_{olf}\ \}$$

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.

4.2 Constraints as an 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 4.4.

We now define a translation of an information-flow constraint set C into an information flow constraint graph $\mathcal{G}_C$, which 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 1 (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 $\tau_0 \sqcup \cdots \sqcup a \sqcup \cdots \tau_n \leq a' \in C$.*

4.3 Correspondence of Graph Cuts and Mediation Points

We now show that a vertex cut of the information-flow graph containing only expression variables corresponds to 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 errors caused by the flows in the command c. We use the Rehof-Mogensen constraint solver [23], introduced in Section 3, in proving these claims.

The following lemma connects paths in the information-flow graph to the unsatisfiability of the constraints set C.

Lemma 2. *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. Please refer to the tech report [16] for the proof.

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 3. *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 4. *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 (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. Please refer to the tech report [16] for the proof.

Cut Example: The logrotate 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. 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 vertices corresponding to `rotateCount` in line 8, `oldName` in line 13, and `newName` line 11 separates `config` from `log_lbl`, and corresponds to placing mediation statements mediating those expression in those lines.

4.4 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 call this problem, *general lattice cut-mediation* (GLC). We will show that the GLC problem is an instance of the graph problem of *cut-conjunction* for directed graphs (DCC), which currently has unknown complexity [14]. Thus, we adopt an approximation strategy to solve GLC that employs the *hitting set* problem, which is known to be an NP-complete problem, but for which several good approximation algorithms exist.

Comparison to the *cut-conjunction* Problem: We first introduce the DCC 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 if and only if none of the pairs of vertices in $\mathcal{P}$ are connected in $G' = (V, E \setminus E')$. The *cut-conjunction (CC)* problem is the following: given a graph $G = (V, E)$ and $\mathcal{P} \subseteq V \times V$ find a subset of edges $E' \subseteq E$ 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.

Given an 0 security lattice $\mathcal{L}$ for a GLC problem, let $\mathcal{P}_{\mathcal{L}}$ be the set of all pairs of labels (l_1, l_2) such that $l_1 \not\leq l_2$. The following lemma generalizes Lemma 2 to $\mathcal{P}_{\mathcal{L}}$-cuts.

Lemma 6. 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 $\mathcal{G}_C$ and $C_{E'}$ be the constraint set generated by the program where the expressions corresponding to the edges in E' have been mediated to $\bot$. The constraint set $C_{E'}$ is satisfiable.

The solution to the cut-conjunction problem for our constructed information-flow graph for a GLC problem then corresponds to the expressions that mediate all illegal flows through the program associated with the information-flow graph. However, the complexity of DCC is unknown [14], so we use an approximation in order to solve GLC.

Placement Algorithm for General Lattices: The algorithm we use to solve GLC consists of two steps: first, we solve the min-cut problem on a per-source

```
MediationPoints($\mathcal{G}_C, \mathcal{P}_\mathcal{L}$)
1  $Labels \leftarrow \{l \mid (l, l') \in \mathcal{P}_\mathcal{L}\}$, $\mathcal{X} \leftarrow \emptyset$
2  for each $l \in Labels$
3      $T_l \leftarrow \{l' \mid (l, l') \in \mathcal{P}_\mathcal{L}\}$
4      $\mathcal{X}_l \leftarrow$ AllMinimumCuts($\mathcal{G}_C, l, T_l$)
5  $S \leftarrow$ MinGHS($\mathcal{X}_{l_0}, \ldots, \mathcal{X}_{l_{|Labels|}}$)
6  return ExpressionsFromEdgeCut($S$)
```

Fig. 2. An algorithm for choosing a set of mediation points for a general lattice based on the generalized hitting set problem

basis, and then we use an algorithm that solves the *hitting set* problem to combine the results. This is an approximation of an optimal solution for the GLC problem, as the per-source cuts are local minima solutions. The hitting set problem is NP-complete, but there are known approximations [1,10].

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 consider a generalized version of this problem where each of the elements in S_i is in turn a subset of T, i.e., S_i is a collection of sets. 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. Let MinGHS($C_1, \ldots, C_n$) be a procedure that solves the hitting set problem. Figure 2 contains an algorithm for placing security mediators for a general lattice that relies on an external procedure for MinGHS to solve the hitting set problem. It is easy to see that if MediationPoints($\mathcal{G}_C, \mathcal{P}_\mathcal{L}$) $= \mathcal{X}$, for all $(l, l') \in \mathcal{P}_\mathcal{L}$, 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 6, mediating the expressions specified in a $\mathcal{P}_\mathcal{L}$-cut results in a satisfiable constraint set.

The running time of this algorithm is primarily dependent on the size of the problem given to MinGHS. The number of cuts generated by AllMinimumCuts depends on the size of the lattice, and the size of the cuts depends on the complexity of the program.

5 Suggestion Framework

The information-flow graph construction in Section 4 constructs, from program code, a graph for which a cut is equivalent to a placement of mediation points. In this section, we discuss how to deploy this method in a framework that outputs sets of mediation points (i.e., *placement suggestions*) for Java programs.

Our tool outputs a set of *suggestions*, each of which is a set of points in the code that completely mediates the illegal information flows from a program. We built a framework that uses minimum graph cuts to select mediation points. The minimum cut of a graph corresponds to the 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. If the programmer wishes to give incentive or decentive to select certain mediation points, then this can be accomplished by modifying the graph cut model.

The framework can be applied to any Java program whose language features are supported by the Jif compiler. 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 labels that the compiler associates with an expression e are affected by both explicit and implicit security information. To make the Jif constraints satisfy cut-mediation equivalence, we modified the constraint generation procedure for every class of expression that could have a visible side effect, so that extra constraints to check implicit flows were included. These additional constraints ensure that information associated with an implicit flow is maintained if α_e is selected as part of a graph cut.

There may be many suggestions of minimum size that resolve the information flows for a given program, as a graph might have several minimum cuts. Therefore, most applications admit an infeasibly large number of minimum sets of mediation points, most of which are very similar. For example, let `h` be a high security integer variable. For the expression `if h == 0 then l := 0`, the expressions `h` and `h == 0` are both part of the minimum set of mediation points. If our framework considers multiple expressions with equivalent security semantics as valid mediation points, the number of minimum cuts quickly becomes exponential in the number of vertices of the information-flow graph $\mathcal{G}_C$. To avoid enumerating an exponential number of mediation points to the programmer, we consider an expression e redundant if its value only flows to another expression e' in the information-flow graph. Suppose $\alpha_e, \alpha'_e \in \mathcal{G}_C$ and let l be a lattice label. If $\alpha_{e'}$ postdominates[2] α_e at exit node $T_l = \{l' \mid l \not\leq l'\}$, do not consider α_e as a mediation point for l [18]. 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 2).

[2] Given a graph $G = (V, E)$, let $n, m \in V$, then m *postdominates* n if m is different from n and m is on every path from n to the end node.

Table 1. Runtime performance of our mediation placement algorithm. We separate Java programs (top) from Jif programs (bottom). Per 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.

Application	Code Lines	# of Constraints	Min Cut Probs.	Avg. Graph Vertices	Avg. Vertices per Min Cut	Cluster 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	8,312.00	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	92,802.00	88.00	15.54	21.61	83.74
logrotate	911	6,063	2	1,654.00	3.50	0.11	0.006	1.34
JPMail (reader)	3,934	8,438	59	3,151.29	3.31	3.88	0.46	13.37
JPMail (sender)	3,932	14,495	32	3,844.69	4.28	4.95	0.12	14.84
Mental Poker	1,578	13,344	1	3,553.00	4.00	0.25	0.24	2.21
Civitas (voter)	13,828	67,135	5	17,658.00	1.4	7.11	0.62	28.71

6 Experimental Results

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

Experimental Setup: Our mediation placement algorithm is written in 1,001 lines of C++ code[3], 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. We ran our analysis on eight separate applications as shown in Table 1: four Java applications for which mediation is added from scratch and four Jif applications in which the manually placed mediators are removed. The labeling and policy were determined per application. To generate the information-flow constraints, 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. An issue with every static analysis is the presence of false positives but our current analysis was sufficient for our examples; while we encountered some false positives, these were easily detected and removed. However, an improved analysis will be necessary in general.

Performance: Table 1 contains metrics about the performance of our system. We allow a programmer to specify a function as the starting point of the analysis. For example, the analysis of Civitas focused on six vote tallying methods.

[3] Our constraint-generation and mincut tools are available for download at `http://siis.cse.psu.edu/jlift/jlift.html`

Since code contains whitespace and comments, total number of constraints generated by the analysis (column 3) is a more accurate metric for the difficulty of the graph problem than file sizes (column 2). Two major factors affected the running time of our tool: Number of minimum-cut problems to be solved per program (column 4) and 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: because domination is a source-sink computation and different minimum-cut problems have different sources and sinks, clustering is performed once for each minimum-cut. Also, it took a longer time for the minimum cut algorithm to run for programs whose cuts had a higher number of vertices, as the Ford-Fulkerson method depends on finding augmenting paths to an existing cut. Our largest code example was an X Server written in Java that contained over 22,000 lines of code, corresponding to 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. A pattern in all of our experiments was the small size of the minimum cut relative to the size of the overall graph indicating that our suggestion algorithm should scale well on even larger programs.

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 information-flow errors for previously unmediated Java programs (JES, Java Card Purse, tinySQL and an X Server implementation called WeirdX). Recent work [15] proposed a tool to display complete and minimal *error traces* that show how an information-flow constraint becomes unsatisfiable, enabling a programmer 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 to run the analysis multiple times to resolve all of the errors per constraint. The results of comparing our tool to such error traces are given in Table 2. These results show that our tool reduces the number of locations by 90% or more for all but one case (Java Card Purse), which is nearly 90%.

Table 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 *error traces* in a prior work [15], while the fourth column gives the number of mediation points selected by our tool in all suggestions. Only tinySQL has multiple suggested min-cuts of the same size (48 of them).

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

Quality of Placed Mediators: To investigate the quality of placed mediation points, we ran our tool on a number of applications (logrotate, JPMail, Mental Poker and Civitas) 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. We classified each selected mediation point as either being *similar* or *not similar* and those classified as similar belonged to one of three categories: Mediation point that mediates the exact same data in the exact same location as the original (**Exact**), is in the same block of code as the original mediation point (**SameBlock**) and mediates the exact same value as the original (**SameData**). Our results in Table 3 show that in all the 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. This means that there are other factors used by expert programmers that need to be assessed in placement. Our framework supports programmer control through the adjustment of weights on the graph edges. We currently use this to enable programmers to prohibit locations (e.g., increase edge weights to ∞) or select locations (i.e., require them in every cut). A key issue appears to be if a programmer has a specific mediator in mind. Ensuring that a location is chosen only if it satisfies the functional requirements of a mediator or other programmer requirements are future work.

Table 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. Additionally, we report the number of suggestion sets returned.

Application	Candidate Mediation Points	Total Mediation Points Suggested	Similarity				Suggestions (# of Sets)
			Exact	Block	Data	Not	
logrotate	1,540	9	1	7	1	0	3
Mental Poker	3,569	7	3	0	1	3	4
JPMail (reader)	2,434	37	1	15	14	7	25
JPMail (sender)	3,976	74	2	52	19	1	23
Civitas (voter)	19,977	9	6	0	2	1	6

7 Conclusion

In this paper we have presented a framework to assist programmers in placing security mediation points. Our framework implements a method that constructs a graph $\mathcal{G}$ such that a minimum cut of $\mathcal{G}$ corresponds to a minimum placement of mediation points in the program. This framework reduced the number of expressions that need to be examined to resolve information-flow errors in four Java programs and placed mediation statements in locations similar to those placed by the original application programmers for four Jif programs. In the future, we plan to provide support for extracting functional requirements from programs that influence placements to improve accuracy.

References

1. Bar-Yehuda, R., Even, S.: A linear-time approximation algorithm for the weighted vertex cover problem. Journal of Algorithms 2(2), 198–203 (1981)
2. Clarkson, M.R., Chong, S., Myers, A.C.: Civitas: Toward a secure voting system. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy, May 2008, pp. 354–368 (2008)
3. Criswell, J., Lenharth, A., Dhurjati, D., Adve, V.: Secure virtual architecture: a safe execution environment for commodity operating systems. SIGOPS Oper. Syst. Rev. 41(6), 351–366 (2007)
4. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4), 451–490 (1991)
5. Deng, Z., Smith, G.: Type inference and informative error reporting for secure information flow. In: ACM-SE 44: Proceedings of the 44th annual Southeast regional conference, pp. 543–548. ACM, New York (2006)
6. Fahndrich, M., Foster, J.S., Su, Z., Aiken, A.: Partial online cycle elimination in inclusion constraint graphs. In: Proceedings of PLDI 1998, pp. 85–96 (1998)
7. Flanagan, C., Flatt, M., Krishnamurthi, S., Weirich, S., Felleisen, M.: Catching bugs in the web of program invariants. SIGPLAN Not. 31(5), 23–32 (1996)
8. Fraser, T., Petroni Jr., N.L., Arbaugh, W.A.: Applying flow-sensitive CQUAL to verify minix authorization check placement. In: Proceedings of PLAS 2006, pp. 3–6. ACM, New York (2006)
9. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proceedings of the 1982 IEEE Symposium on Security and Privacy, April 1982, pp. 11–20 (1982)
10. Halperin, E.: Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. In: SODA 2000: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 329–337. Society for Industrial and Applied Mathematics (2000)
11. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In: Proceedings of PLDI 2001, June 2001, pp. 254–263 (2001)
12. Hicks, B., Ahmadizadeh, K., McDaniel, P.: From languages to systems: Understanding practical application development in security-typed languages. In: ACSAC 2006: Proceedings of the 22nd Annual Computer Security Applications Conference, pp. 153–164. IEEE Computer Society, Los Alamitos (2006)
13. Hicks, B., Rueda, S., Jaeger, T., McDaniel, P.: From trusted to secure: Building and executing applications that enforce system security. In: Proceedings of the USENIX Annual Technical Conference, June 2007, pp. 1–14 (2007)
14. Khachiyan, L., Boros, E., Elbassioni, K., Gurvich, V., Makino, K.: Enumerating disjunctions and conjunctions of paths and cuts in reliability theory. Discrete Appl. Math. 155(2), 137–149 (2007)
15. King, D., Jaeger, T., Jha, S., Seshia, S.A.: Effective blame for information-flow violations. In: SIGSOFT 2008/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 250–260. ACM, New York (2008)

16. King, D., Jha, S., Muthukumaran, D., Jaeger, T., Jha, S., Seshia, S.: Automating Security Mediation Placement. Tech. Rep. NAS-TR-0123-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA (January 2010)
17. McCamant, S., Ernst, M.D.: Quantitative information flow as network flow capacity. In: Proceedings of PLDI 2008, pp. 193–205. ACM, New York (2008)
18. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
19. Myers, A., Liskov, B.: Complete, safe information flow with decentralized labels. In: Proceedings of the IEEE Symposium on Security & Privacy, May 1998, pp. 186–197 (1998)
20. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: Proceedings of POPL 1999, January 1999, pp. 228–241 (1999)
21. Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy software. ACM Transactions on Programming Languages and Systems 27(3), 477–526 (2005)
22. Pottier, F., Simonet, V.: Information flow inference for ML. In: Proceedings of POPL 2002, pp. 319–330. ACM Press, New York (2002)
23. Rehof, J., Mogensen, T.A.: Tractable constraints in finite semilattices. Science of Computer Programming 35(2-3), 191–221 (1999)
24. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: CSFW 2005: Proceedings of the 18th IEEE Workshop on Computer Security Foundations, pp. 255–269. IEEE Computer Society, Los Alamitos (2005)
25. Shapiro, M., Horwitz, S.: Fast and accurate flow-insensitive points-to analysis. In: Proceedings of POPL 1997, pp. 1–14. ACM, New York (1997)
26. Tip, F., Dinesh, T.B.: A slicing-based approach for locating type errors. ACM Trans. Softw. Eng. Methodol. 10(1), 5–55 (2001)
27. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. Journal of Computer Security 4(3), 167–187 (1996)
28. Walsh, E.: Integrating X.Org with Security-Enhanced Linux. In: Proceedings of the Third Annual Security Enhanced Linux Symposium, March 2007, pp. 33–40 (2007)
29. Wheeler, D.A.: Software/dbus, `http://www.freedesktop.org/wiki/Software/dbus`
30. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)
31. Wright, C., Cowan, C., Morris, J., Smalley, S., Kroah-Hartman, G.: Linux security modules: General security support for the linux kernel. In: Proceedings of the 11th USENIX Security Symposium, August 2002, pp. 17–31 (2002)
32. Zhang, X., Edwards, A., Jaeger, T.: Using CQUAL for static analysis of authorization hook placement. In: Proceedings of the 11th USENIX Security Symposium, August 2002, pp. 33–48 (2002)