The Pennsylvania State University

The Graduate School

AUTOMATING THE PLACEMENT OF AUTHORIZATION HOOKS IN PROGRAMS

A Dissertation in

Computer Science and Engineering

by

Divya Muthukumaran

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

December 2013

The dissertation of Divya Muthukumaran was reviewed and approved* by the following:

Trent R. Jaeger
Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Patrick D. McDaniel
Professor of Computer Science and Engineering

Adam Smith
Associate Professor of Computer Science and Engineering

John Hannan
Associate Professor of Computer and Science Engineering

Stephen Simpson
Professor in the Department of Mathematics

Lee Coraor
Associate Professor of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

When servers manage resources on behalf of multiple, mutually-distrusting clients, they must interface with an authorization policy that determines whether a client request to access a resource is allowed. This goal is typically achieved by placing authorization hooks at appropriate locations in server code. The goal of authorization hook placement is to completely mediate all security-sensitive operations on shared resources while satisfying optimality guarantees with respect to an access control policy.

To date, authorization hook placement in code bases, such as the X server and postgresql, has largely been a manual procedure, driven by informal analysis of server code, and discussions on developer forums. There is even lack of consensus on basic concepts, such as the definition of what constitutes a security-sensitive operation.

In this thesis, we propose that program analysis techniques may be used to infer security-sensitive objects and operations in programs and automatically identify a hook placement that satisfies certain optimality guarantees. To support this thesis, we first discuss an approach that uses graph cuts to solve information flow errors. We show how this approach can be used to solve errors in both programs and distributed system deployments. However, this approach is not effective enough to deal with all aspects of the authorization hook placement problem, specifically inferring security specification.

Next, We designed an approach to identify security-sensitive objects and operations that is motivated by a novel observation — whenever a user makes *deliberate choice* of objects from a collection of objects and requests a specific operation to be performed on the object, this operation must be authorized. We have built a tool that uses this observation to statically analyze the server source. Using real-world examples (the X server and postgresql), we show that hooks placed using our approach are just as effective as hooks that were manually placed over the course of several years.

Finally, we propose a formal approach to reason about optimal hook placements with respect to placement choices that programmers have to make. We demonstrate how constraints on authorization policies can help minimize authorization hook placeements. We also show that using our hook placement approach reduces the number of manual choices programmers have to make.

Table of Contents

List of Figures **ix**

List of Tables **x**

Acknowledgments **xi**

Chapter 1
Introduction **1**
1.1 Security is not a ‘blocker’ 1
1.2 The trouble with retroactive security 2
1.3 Retrofitting code for authorization 3
1.4 Challenges in authorization hook placement 4
1.5 Contributions of the thesis 5

Chapter 2
Background **8**
2.1 Dependence Analysis in Programs 8
2.2 Control Dependence Graph 9
2.3 Taint Analysis 10
2.3.1 Dynamic Taint Analysis 11
2.3.2 Static Taint Analysis 11
2.4 Principles of static taint analysis 12
2.4.1 Procedure for Static Taint Tracking 14
2.5 Automated Specification Inference 15
2.6 Authorization Hook Placement Problem 16
2.6.1 Case Study: The X Windowing Server 20
2.6.1.1 Retrofitting the X server for authorization 22

2.7 Authorization Hook Placement 25
2.7.1 Automated Hook Placement 25
2.7.2 Automated Hook Verification 26
2.7.3 Implementing Access Checks 27

Chapter 3
Resolving information flow errors using graph cuts **29**
3.1 Introduction 29
3.2 Background 32
3.3 Approach 33
3.3.1 Cut Mediation Equivalence 33
3.3.2 Resolving errors in general lattices 34
3.3.3 Adding constraints to the cut problems 36
3.4 Solving real-world information flow errors 37
3.4.1 Resolving Information flow errors in programs 38
3.4.1.1 Specifying Security Requirements: 39
3.4.1.2 Specifying Placement Requirements 40
3.4.1.3 Placing mediators in programs 41
3.4.1.4 Evaluation of graph cut approach in programs 42
3.4.2 Resolving Information flow errors in distributed system deployments 43
3.4.2.1 Creating a system-wide information flow graph 44
3.4.2.2 Creating an integrity lattice 44
3.4.2.3 Placing mediators in distributed system deployments . . 45
3.4.2.4 Evaluation of graph cut approach in distributed system deployments 46
3.5 Discussion 48

Chapter 4
Leveraging 'Choice' to automate authorization hook placement **50**
4.1 Introduction 50
4.2 Approach Overview 52
4.3 Design 54
4.3.1 Stage 1: Tracking User Requests 56
4.3.2 Stage 2: Finding Security-Sensitive Objects 58
4.3.3 Stage 3: Finding User-Choice Operations 60
4.3.4 Stage 4: Placing Authorization Hooks 63
4.3.4.0.1 Hoisting common operations 64
4.3.4.0.2 Avoiding redundant mediation 65
4.4 Implementation 66
4.4.1 The CIL Infrastructure 66

4.4.2 Building a Control Dependence Graph 68
4.4.3 Input specifications 68
4.5 Evaluation 69
4.5.1 Reduction in Programmer Effort 70
4.5.1.0.3 Memcached 71
4.5.1.0.4 pennmush 71
4.5.2 Accuracy Relative to Manual 72
4.5.2.0.5 X Server 72
4.5.2.0.6 postgres 74

Chapter 5
Generating the Least Mediation for Least Privilege **76**
5.1 Introduction 76
5.2 Motivation 81
5.2.1 Background on Hook Placement 81
5.2.2 How Manual Placements Differ 83
5.2.3 Balancing Minimality with Least Privilege 84
5.3 Authorization Hook Placement Problem 86
5.3.1 Identifying security-sensitive operations 86
5.3.2 Consolidating Hook Placements 87
5.4 Design 89
5.4.1 Deriving a constrained placement 90
5.4.1.1 Hoisting 91
5.4.1.2 Removal 92
5.4.2 Deriving a set of authorization constraints 94
5.4.2.1 Deriving suggested constraints bottom-up 95
5.4.2.2 Deriving implied constraints top-down 96
5.4.2.3 Using constraint selectors to make choices 97
5.4.3 Hook Placement Properties 98
5.5 Evaluation 100
5.5.1 Using constraint selectors to automate hook placement 102
5.5.1.1 Hook Placement using MLS 105
5.5.1.2 Hook Placement using RUN 107
5.5.1.3 Hook Placement using ‘MLS+RUN’ 108
5.5.2 Evaluating the top-down approach 109
5.5.3 Discussion 110
5.5.3.1 Effect of aliasing 110
5.5.3.2 Transferring hook placements between versions 111

Chapter 6
Conclusion **112**

Bibliography **113**

List of Figures

2.1 Shows the difference between a Control Flow Graph(CFG) and a Control Dependence Graph (CDG) 10

3.1 Ordering the cut problems 35
3.2 Greedy Mediation Resolution Algorithm 36
3.3 Model for specifying placement requirements 40

4.1 Program model of the authorization hook placement problem 52
4.2 Shows the sequence of stages in automated authorization hook placement. The light shaded boxes shows the required programmer specification for every program, the dark shaded box shows language-specific specification, and the dashed box shows optional specification. 55
4.3 Shows the CDG with the user-choice operations for the DIXCHANGEWINDOWPROPERTY function 62

5.1 Hook Placement for functions MAPWINDOW, MAPSUBWINDOWS, and CHANGEWINDOWPROPERTY 82
5.2 Authorization hook placement method 89

List of Tables

2.1 Tabulation of access modes in the X server 23

3.1 The number of mediators needed for: (1) infrastructure mediation per VM (default); (2) remote threats when application is deployed; and (3) local threats to deployment. For "unique mediators" we count each mediator only once even if it is used in a different VM. 47

4.1 Table showing the reduction in programmer effort that the automated approach provides. 71

5.1 Table showing the lines of code (LOC), number of manual hooks (MAN), default hooks (DEF), and number of hooks generated using the MLS, RUN and MLS+RUN constraint selectors for hook placement. 102
5.2 Table showing the hoisting (HST) and removal (REM) suggestions in the bottom-up approach for the default placement and placements generated using the constraint selectors. 102
5.3 Distribution of hooks and hoisting and removal choices among files in X Server 1.13. These 6 files have 80% of all the hooks in X Server. Column 'REM' refers to removal choices, and 'HST' refers to hoisting choices. . . 103
5.4 The number of hooks placed using the 'RUN' constraint selector with three different thresholds - 0.7, 0.8 and 0.9 108
5.5 Table showing the equivalence and subsumption constraints implied by manual placements by comparing against the default placement and those generated using constraint selectors for X Server. 110

Acknowledgments

I am deeply indebted to my advisor Trent Jaeger for seven years of guidance and support. I am also grateful to have been part of the SIIS lab where I met colleagues who were extremely motivated and provided me with encouragement and a sounding board whenever I needed one. I thank Dave King for introducing me to programming language security and for being an early mentor in the area.

My parents have always given me and continue to give me more than I could ever want. I thank them for buying the first computer in our family and for letting me play hours of computer games as a child. I am especially grateful that my father encouraged me to learn programming during my summer breaks in high school. Last, but most importantly, I thank my husband Vikram who shared this journey with me and supported me through all the good and bad times.

Funding: My work was supported by NSF grant CNS0905343 for which I am very grateful.

Chapter 1

Introduction

1.1 Security is not a 'blocker'

For many years, the security community has been advocating security by design - a prescription that any software be designed from the ground up with security as a major focus. However, more often than not, this advice falls by the wayside and, in practice, functionality concerns trump security concerns. Indeed, programmers do not think of security issues as a 'blocker' - a bug that prevents a software from being shipped. The following quote is an excerpt from a discussion between programmers on a real software project to highlight the attitude that programmers typically have towards security issues. Following a bug ticket raised for incorporating security hooks into the code that would prevent keystroke sniffing attacks [1], programmers raised concerns about dedicating manpower to dealing with security issues when some functionality issues were yet to be dealt with.

> *"It isn't clear this (security bug) qualifies as a blocker under any circumstances. Its priority, along with any kernel support for isolation, ranks behind power management and suspend/resume in our base system work.*
>
> *The importance of security increases only as we are into serious deployment and start becoming a target. First things, first...."*

This highlights the trend of reactively fixing security issues - which is, to deploy insecure software, wait for it to become pervasive enough for attackers to target it, and then patch vulnerabilities as attackers identify and exploit them. Despite this, there are a handful of examples of systems such as the Postfix mail program [2] and some database servers [3, 4] that have been proactively designed for security. However, we are still left with several large, legacy insecure codebases that need to be retroactively fitted with security.

1.2 The trouble with retroactive security

Currently, retrofitting of code for security was done manually. Manual retrofitting efforts have been undertaken for large codebases such as the Linux kernel, OpenSSH, and postgres. Manually sifting through large codebases, needless to say, is a tedious process, one that programmers would rather do without. Indeed these manual efforts have been beset with problems. First, very few people have the necessary expertise regarding any single piece of legacy software. Second, without expertise, it is very difficult to reason about security of the software, leading to the manual effort being buggy or otherwise incomplete. Third, on large codebases, reaching a consensus on the placement of security code can take many years. And finally, systems that are designed for security may yet evolve as newer functional requirements emerge. Such changes may break existing security mechanisms and trigger a reevaluation of security of the system. Karger and Schell [5] showed that modifications to improve the usability of Multics broke several of its design assumptions leading to exploitable vulnerabilities.

All of this motivates the need for automated techniques to reason about security, derive security specifications, and finally retrofit code with security mechanisms. Having realized the importance of this problem, researchers have proposed automated code retrofitting tools to tackle some important problems in recent years. CCured [6] both statically verifies pointers to be type-safe where possible and inserts runtime checks where they cannot be verified statically. Privtrans [7] refactors code for privilege separation. Other efforts target retrofitting code for placing mediation statements [8], and perform code weaving to transform code to work with specific policies such as Decentralized Information Flow Control (DIFC) [9, 10]. This thesis specifically targets the retrofitting of code to place authorization hooks in legacy programs.

1.3 Retrofitting code for authorization

Access control is the process of mediating every request to resources and data maintained by a system and determining whether the request should be granted or denied. A mechanism called the *reference monitor* enforces regulations established by a *security policy* and returns an access control decision. The security policy defines the set of security-sensitive operations on objects that each subject is allowed to perform. The aim of a reference monitor interface is to mediate all security-sensitive operations in order to enforce of the security policy.

The Anderson Report [11] stipulates the requirements for developing a secure reference validation mechanism, which includes both the authorization hooks and authorization mechanism.

- The reference validation mechanism must always be invoked (*complete mediation*)
- The reference validation mechanism must be tamperproof (*tamperproof*).
- The reference validation mechanism must be small enough to be subject to analysis and tests (*verifiable*).

Over the last ten years, several operating systems and server applications have been retrofit with reference monitor interfaces [11]. For example, while the Linux operating system was proactively designed with a discretionary access control (DAC) mechanism [12, 13], this mechanism was found to be insufficient to protect the security of hosts in a networked world [14]. As a result, the Linux kernel was retrofit with a reference monitor interface for loadable kernel modules, called the Linux Security Modules (LSM) framework [15]. Many applications other than the operating system kernel also need retrofitting for security. For example, the X Server, like many server applications, enables multiple clients to make requests on objects that it manages. Void of isolation within the server, malicious clients can compromise the integrity and privacy of other clients handled by the application—well-documented instances of attacks on similar server applications abound in the literature (e.g., [16, 17, 18, 19]). Now, a variety of server applications[20, 21, 22, 23] as well as other operating systems [24, 25, 26] and virtual machine monitors [27, 28] have been retrofit with reference monitor interfaces.

Unfortunately, these efforts have been beset with problems. This is because the identification of security-sensitive operations and the placement of hooks is a manual procedure, largely driven by informal discussions on mailing lists in the developer community. There is no consensus on a formal definition of what constitutes a security-sensitive operation, no tool support to identify their occurrence in large code-bases. Discussions of which hooks to deploy can often last years. For example, the development of the LSM framework from the myriad of patches [29, 30, 31, 32, 33, 34] was a manual process of collecting access decision points or *authorization hooks* from each patch and resolving inconsistencies among these choices. Despite all the prior work put into these patches, it still took over two years for the LSM framework to be accepted into the mainline Linux kernel, and a number of bugs were found and fixed along the way [35, 36] and several years after release [37].

1.4 Challenges in authorization hook placement

There are two main challenges in the placement of authorization hooks. The first step is the identification of the *objects* to be protected, the *subjects* that execute activities and request access to objects, and the *operations* that can be executed on the objects, and that must be controlled. Subjects, objects, and operations may be different in different systems or application contexts. For instance, in the protection of operating systems, objects are typically files, directories, or programs; in database systems, objects can be relations, views, stored procedures, and so on. In this work we tackle servers that manage resources for and provide services to multiple, mutually-distrusting clients. Therefore we need to infer program specific subjects, objects and operations.

Second, after we have an initial set of specifications, we need to determine how to place hooks that satisfy a set of requirements.

- The hook placement must satisfy the *complete mediation* guarantee of a reference monitor[1].
- The hook placement must satisfy the *principle of least privilege*, i.e., a subject must only have permissions required to perform the operation requested.
- The hook placement must be *minimal*[2] with respect to a set of *authorization constraints*.

The first two are security guarantees typically expected of any access control mechanism. In addition, we note that when programmers place authorization hooks manually, they make assumptions about the authorization policies that will be written for the application. Constraints on the set of policies that will be enforced by the hook placement are used to reduce the number of hooks in the placement. In other words, we need to aim for a minimal placement that is capable of enforcing access control policies that satisfy a set of constraints.

1.5 Contributions of the thesis

In this dissertation, we develop techniques that automatically infer security specifications pertaining to placing authorization hooks, determine the placement of hooks that satisfy se-

[1]The *verifiability* property of reference monitors concerns the actual hook code. Since this work is about identifying locations for hooks, we do not deal with this property. The *tamper-proof* property concerns the deployment of the application, and is beyond the scope of this work.

[2]Minimal Placement means that if any hook is removed from the placement, then the complete mediation property no longer holds.

curity and optimality guarantees. The thesis that this dissertation supports is the following:

We can use program analysis techniques namely, taint tracking and control dependence analysis, to retrofit server applications automatically with a minimal number of authorization hooks necessary to enforce desired authorization policies while satisfying complete mediation and least privilege guarantees

Our contributions to support this thesis are as follows:

1. *Automating mediation placement with graph cuts:* First, several security problems are manifestations of information flow errors. Inspired by network flow, we discuss a technique to resolve such information flow errors guaranteeing complete mediation. We discuss the application of this technique to solving errors in programs and system-wide access control policies demonstrating its usefulness. We then discuss the shortcomings of the approach especially in dealing with the problem of authorization hook placement. Chapter 3 discusses this work.

2. *Inferring access control specifications:* We provide a technique to automatically infer specification of security-sensitive objects and operations in server programs for authorization hook placement. In a server, clients make requests and by means of this request are able to choose objects and operations in the program. Based upon this observation, we design a static taint analysis that tracks user choice to identify both security-sensitive objects and the operations that the server performs on them. Chapter 4 discusses this work.

3. *Authorization hook placement:* We propose a technique based on control dependence analysis that can identify minimal hook placements capable of enforcing any access control policy written for the set of identified security-sensitive operations while also providing complete mediation and least-privilege guarantees. We have implemented a prototype tool that applies this technique to C programs and evaluate the technique

on several real-world userspace server programs. We show that our technique results in more than 90 % reduction in the programmer effort required to place hooks. Chapter 4 discusses this work.

4. *Reducing programmer choice in hook placement:* We provide formalism to reason about hook placements that can enforce authorization policies that satisfy a set of *authorization constraints*. We show techniques to generate authorization constraints in a semi-automated fashion and describe how a programmer would go about placing hooks using this framework. We then discuss the results from applying this technique to real-world server programs. We make a case for how our approach reduces the number of choices the programmer has to make. This is discussed in Chapter 5.

In this thesis, we have developed static program analysis techniques to help programmers place authorization hooks in programs by automating both the identification of security-sensitive operations and placing authorization hooks that satisfy a set of security-guarantees.

Chapter 2

Background

2.1 Dependence Analysis in Programs

Our techniques in this thesis rely on static dependence analysis of programs. Dependence analysis is the automatic identification of the potential for one element (such as a program statement) to affect or be affected by other elements of a program or system or its environment [38]. There are four main kinds of dependencies that are explored during the analysis of programs

- *Control Flow Analysis:* Control flow analysis is the process of identifying which statements in a program have the potential to lead directly to the execution of other statements based on the syntax of the program code. Accurate representation of potential flows among program statements is essential to the identification of which variable definitions can reach variable uses and which statements control the execution of other statements. Control flow analysis is fundamental to dependence analysis. Control flow relationships among statements in a program are transitive and can be represented in a graph, referred to as a control flow graph (CFG).

  Definition 1. *A Control Flow Graph (CFG) of a program P a digraph $CFG = (V, E)$ where V is a set of vertices, one for each statement s_i in the program; and*

E is a set of edges $\{e_1, ..., e_n\}$ where each e_i represents a control flow between two statements in the program. There are special entry and exit nodes in the graph.

- *Data Flow Analysis:* Data ow analysis is the process of identifying the potential for a value computed in one statement to affect the computation in another. During data flow analysis sets of variable information known as "gen" and "kill" sets, representing the variable values that are generated (defined) and killed (redefined) respectively are computed for each statement in the program. These sets are manipulated by data flow equations in order to solve problems relating to the potential for a variable definition to affect a computation later in a program's execution

- *Control Dependence Analysis:* Control dependence captures the effect of predicate statements on program behavior. Control dependence determines which other program statements must be executed for a given statement to execute in the program. It can be extracted from the control flow graph of the program.

- *Data Dependence Analysis:* Data dependence analysis determines whether the value of a variable a program location can affect the value of another variable at at different program location.

In this work we concentrate mainly on control dependence and data dependence analysis. We now provide an overview of control dependence graphs and taint-analysis which is a technique to identify data dependence.

2.2 Control Dependence Graph

The control dependence property is typically defined in terms of the control dominance relationships present in the CFG of the program. First we define the dominator and post-dominator relationships in a graph.

Definition 2. *In control flow graphs, a statement X dominates a statement Y if every path from the start node to Y must go through X.*

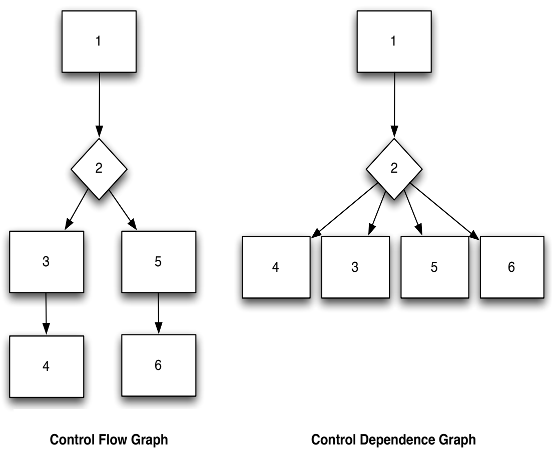

Figure 2.1: Shows the difference between a Control Flow Graph(CFG) and a Control Dependence Graph (CDG)

Definition 3. *In control flow graphs, a statement Y is said to post-dominate a statement X if all paths to the exit node of the graph starting at X must go through Y.*

Definition 4. *A statement Y is control-dependent on X if Y post-dominates a successor of X in the CFG but does not post dominate all successors of X.*

Figure 2.1 illustrates the difference between a control flow and a control dependence graph.

2.3 Taint Analysis

Taint analysis is a well-known technique for software vulnerability detection. It is also known as User-Input Dependency Checking and is typically used to determine if an input from a user can cause unexpected or possibly malicious actions in the program by tracking where the user input can flow to in the program. Taint analysis has been most often used in the detection of Input Validation vulnerabilities such as buffer overflows, format string vulnerabilities, improper data validation, string termination error, missing XML validation etc. The main advantage of taint-tracking approaches is that accurate, application-independent policies can be developed for the above attacks. Researchers have employed both static and dynamic taint analysis in an effort to uncover such bugs.

2.3.1 Dynamic Taint Analysis

Dynamic taint analysis monitors the flow of untrusted input through the program during runtime and checks if it can influence a security sensitive function or data in the program. Dynamic taint analyses have been used in several research efforts at vulnerability detection in programs, for example, to detect memory corruption attacks [39], to detect injection attacks [40] and at the instruction level [41], [42], [43] to prevent control transfers based on tainted data, such as those associated with buffer-overflow attacks.

Dynamic taint analysis has even been built into some programming languages such as PERL [44] which has a runtime *taint mode* that flags any input derived from outside the program as 'tainted' and enforces the requirement that such data should be 'untainted' before being passed to another program or system call. The advantage of dynamic analyses in general are low false positive rates since they only exercise valid program paths. But the disadvantages include an adverse effect of the performance of the program and incompleteness of the analysis since only paths that are exercised during the run of the program are analyzed. BitBlaze [45], Buzzfuzz [46], TaintCheck [47], and Dytan [48] are some of the tools that perform dynamic tainting.

2.3.2 Static Taint Analysis

Static taint analysis over-approximates the set of instructions that are dependent on the user input by statically analyzing the source code of the program. The advantage of static analysis is low false negatives since it examines all possible program paths. The pitfall is imprecision and therefore the presence of false positives since it lacks the information to reason about feasibility of all paths. Parfait [49], CQual [50], SPlint [51] and Pixy [52] are some of the tools available to perform static analysis of programs.

Livshits and Lam [53] used static taint analysis to detect SQL injections, cross-site scripting and HTTP splitting attacks in Java applications. Wassermann and Su [54] used static taint analysis to detect injection vulnerabilities in PHP applications. Shmatikov et al [55] used static taint analysis to track how user inputs can cause denial of service (DOS) attacks by influencing high-complexity control structures in programs. In this work, we use static taint analysis to determine how user input influences the objects the program operates on and the operations it performs on such objects.

2.4 Principles of static taint analysis

In this section we lay out the basics of static taint analysis. We first define the taint analysis problem and then describe the data flow analyses formulation of the problem for a subset of the C programming language.

Definition 5. *Given a program P and a set of variables V in P, a variable v_i is* data dependent *on a variable v_j if there exists an assignment $v_i := e$ such that expression e either contains v_j or contains another variable v_k which is itself data-dependent on v_i. Let $\mathcal{D}(v_i, v_j)$ be a relation which is true if variable v_i is data dependent on variable v_j.*

Definition 6. *The taint analysis problem consists of the source-sink tuple (V_I, V_S) where V_I is the set of variables representing the user-input and V_S representing the set of target variables. The result is a 'taintedness' mapping $\mathcal{T} : V_S * Bool$ from each target variable $v_s \in V_S$ to a boolean determined by $\bigvee_{v_i \in V_I} D(v_s, v_i)$*

To model taint analysis as a dataflow analysis problem we need to specify three things:

- The control flow graph(CFG) of the program.
- The dataflow being tracked.
- The transfer function that determines how the data flow is updated at each node.

First, we construct the Intraprocedural Control Flow Graph(CFG) of each procedure which consists of the tuple $(V, E, Label, Entry, Exit)$ where V is the set of vertices of the control flow graph and E is the set of edges that represents the control flow of the function. $Label$ is a mapping from each node to the semantics of the node. For the purpose of this section we only consider two types of labels - assignments and function calls. Finally, each procedure has an $Entry$ node that specifies the formal parameters of the function and a $Exit$ node that specifies the value returned by the function to its caller at its exit.

Second, we need to determine the data flow that is to be computed by the analysis. In our case, it is the 'taintedness' relation $\mathcal{T} : V * C * Bool$ for each variable. We also qualify the 'taintedness' relation with a calling context C to determine under which contexts a variable v be tainted. At the initialization of the dataflow analysis we set the 'taintedness' for the source variables to be $True$ in all contexts.

$$\mathcal{T}(v, *) = \begin{cases} True & \text{if } v \in V_I \\ False & \text{otherwise} \end{cases}$$

Finally, we need a transfer function $\mathcal{F} : V * C * Label * \mathcal{T} \rightarrow \mathcal{T}$ for each node that determines how it updates the 'taintedness' relation based on its semantics and the current context. We now discuss the transfer functions for assignments and function calls and returns.

An expression e is tainted if any of its constituent variables are tainted.

$$\frac{v := e \qquad \mathcal{T}(e, C)}{\mathcal{T}(v, C)}$$

Function calls are modeled as tuples $(p, q, aList, v_i)$ where p is the caller and q the callee, $aList = \{a_1, a_2, ..., a_k\}$ is the list of actual call parameter variables and v_i is the variable in p that receives the value returned by the procedure call. The corresponding formal parameters are specified in the $Label$ map of the $Entry$ node of q as $q(f_1, ..., f_k)$. The callee is evaluated within each distinct context. If the actual parameter to a call a_i is tainted, then the corresponding formal parameter f_i in the callee is tainted.

$$\frac{(p, q, [a_1, ..., a_k], v_i) \qquad q(f_1, ..., f_k) \qquad \mathcal{T}(a_i, C)}{\mathcal{T}(f_i, r :: C)}$$

At the exit node we need to ensure that the taint from the returned value is propagated to the caller. The exit node of q contains the label map $q(v_q)$.

$$\frac{(p, q, [a_1, ..., a_k], v_i) \qquad q(v_q) \qquad \mathcal{T}(v_q, q :: C)}{\mathcal{T}(v_i, C)}$$

If the formal parameters are references then we would also need to do a reverse taint from the actual to the formal parameters. In addition, we also need to handle pointers. We choose to omit them here for expository reasons.

2.4.1 Procedure for Static Taint Tracking

Our goal is to model the dataflow analysis context sensitively. But in general, we may end up with an exponential number of contexts. In order to avoid re-analyzing each procedure within each context, we use function summarization as defined by Sharir and Pnueli [56]. This allows us to analyze a procedure once and create an analysis-specific summary of the procedure that can be used within each calling context. In the case of taint analysis, the summary for each procedure q is the taint at the return of the function (returns and call by reference parameters) in terms of the taint at the formal parameters. At every function call, the taint at the actual call parameters can be used as arguments to the function summary.

The taint analysis is performed in two pass through the call graph of the program.

1. *Bottom-up summarization* phase visits each node of the call-graph in the *reverse topological sort* order and creates procedure summaries. Performing a strongly-connected-component decomposition of the call graph ensures that before a summary for a procedure is created, summaries already exist for all its callees. Recursive procedures are treated context-insensitively.

2. *Top-down propagation phase* visits the call graph in topological order, computing the actual taint at each variable, using procedure summaries computed in the previous stage to resolve procedure calls. The ‘taintedness’ $\mathcal{T}(v)$ for a variable v in a

procedure q is computed by combining the ‘taintedness’ $\mathcal{T}(v, C)$ under all possible calling contexts C for q.

2.5 Automated Specification Inference

Program specifications are a vital component for any program verification and retrofitting tool. Program specifications can be used both prescriptively to prevent bugs before deployment and retroactively to verify and rectify bugs. In the past, specifications were largely manual and writing specifications even with the best annotation systems can be daunting task [57, 58]. Consider Jif [59] which extends the Java programming language with security annotations and a security type compiler which ensures that programs do not violate information flow security specifications. One of the roadblocks to widespread adoption of tools like Jif is that the number of annotations can be significant even for small programs. For example, the often cited Mental Poker program of only 1369 lines of code has around 1175 annotations [60]. Askarov and Sabelfeld [61] report 150 person-hours to develop a security-typed implementation of a cryptographic protocol, compared to 60 person-hours for a non-security-typed implementation. Automating the inference of security specifications helps the programmer both during software development and during verification and retrofitting of legacy programs. During development, it helps ensure the completeness of specifications and during the retrofitting or verification of legacy programs it saves several valuable man hours that might otherwise be spent suffering through possibly unfamiliar code.

Static specification inference has predominantly been probabilistic in nature. Kremenek et al [62] observe that “there are many sources of knowledge, intuitions and domain-specific observations that can be automatically leveraged to help infer specifications from programs automatically”. They use the intuition that programs are typically well-structured and programs behave in the way programmers intended them to. They posit that the more we can observe that an object serves a specific role in the program, the more we believe that they were designed for that specific role. In their work they try to model object creation, ownership and consumption in a program and infer them using probabilistic factor graphs. The evidence for the specifications can come from different sources such as behavioral signatures, prior beliefs and ad hoc knowledge. In Merlin [63] the authors automatically infer information flow specifications and classify nodes in a propagation graph into sources, sinks and sanitizers. They begin with assumption that most program paths are secure and pass through a sanitizer routine and use probabilistic path constraints to generate the specifications. A probabilistic constraint is a path constraint parameterized by the probability that the constraint is true. They find that their tool works better if it is used to 'complete' an initial rudimentary specification rather than being used to infer specifications from scratch. There has also been a significant body of work in dynamic invariant generation. Tools such as Daikon [64] and DIDUCE [65] generate invariants from multiple program runs and have been used in several other tools to generate security specifications [66, 67]

2.6 Authorization Hook Placement Problem

In this section, we define the authorization hook placement problem and follow with a case study of the authorization hook placement problem and proposed solution for the X Windowing Server.

Authorization is the process of permitting a *subject* (e.g., user) to perform an *operation* (e.g., read or write) on an *object* (e.g., program variable or channel), which is necessary to control which subjects may perform security-sensitive operations. Lampson [68] was the first to introduce the notion of authorization using an access control matrix. Each column of the matrix corresponds to a system resource, and each row corresponds to a system user. Each entry *(subject, object)* denotes the rights that a *subject* has on system resource *object*. The safety problem asks the question of whether a *subject* can ever access gain a certain right on a *resource* through a series of commands. This problem was shown to be undecidable [69].

The access control matrix depicts the state of possible access rights of a subject at a given instant in the system, i.e., it represents the protection state of a system. Access matrices are usually sparse, and rarely implemented as matrices. The typical ways to represent access control are either through access control lists (columns of the access control matrix) or capability-based security (rows of the access control matrix). An access control list (ACL) is a list of permissions attached to a resource. An ACL specifies which users or system processes are granted access to objects, as well as what operations are allowed on given objects. A capability is an unforgeable token of authority. Each subject may possess a set of capabilities and may present these capabilites to the 'guard' of a resource in order to gain access.

Historically there have been two models of access control: Discretionary Access Control (DAC) and Mandatory Access Control (MAC). In the discretionary model, the owner of an object specifies which subjects can access the object and delegation of access is possible. This model is called discretionary since the control of access is at the discretion of the owner. In mandatory access control, a central authority such as a system administrator decides access control and it cannot be changed by individual users of the system. Mandatory access control is based on the idea of attaching labels to subjects and objects. Subjects are given a security clearance (such as Secret, Top Secret, Public) and objects are given categories (such as Secret, Top Secret, Public). The subject's labels are matched with the object's labels when an access request is made. Traditionally most operating systems followed the DAC model of access control while MAC was reserved for military applications. Subsequently, SELinux brought MAC to commercial operating systems (integrated into the Linux kernel since 2.6). More recently, mandatory integrity control was introduced into Windows Vista and other mandatory schemes derived from FreeBSD have been used in iOS and OS X. These commercial implementations of MAC have done away with the rigor of earlier MLS systems, focusing instead on network attacks and malware.

An authorization hook is a program statement that submits a query for the request *(subject, object, operation)* to an authorization mechanism, which evaluates whether this request is permitted (e.g., by a policy). The program statements guarded by the authorization hook may only be executed following an authorized request. Otherwise, the authorization hook will cause some remedial action to take place that is customized to the program.

A reference monitor must satisfy three key properties:

1. *Complete Mediaton:* Every security-sensitive operation to a shared resource must be preceded by an access control hook.

2. *Tamper Resistance:* An attacker must not be able to circumvent the mechanism, e.g, by rewriting the code of the reference monitor.

3. *Verifiability:* The reference monitor must be small enough to allow for verification.

The placement of authorization hooks mainly addresses complete mediation, which requires that all security-sensitive operations be mediated by an authorization hook to ensure that the security policy permits all such operations. However, we find that the placement of authorization hooks also impacts tamperproofing, by authorizing requests that may affect the authorization process (e.g., change the policy), and verifiability, by determining where authorization queries are deemed necessary. Verifiability, in particular, may be aided by an automated method for authorization hook placement, as it often takes years for developers to arrive at a consensus regarding an acceptable placement.

The main challenge in producing an authorization hook placement is identifying what a security-sensitive operation is. To date there is no formal definition of this concept nor even a decent working definition, as discussed in the Introduction. As an operation must be applied to an object, the definition for security-sensitive operations must identify both security-sensitive objects and operations on these objects that may impact the program's security enforcement. In prior approaches, these definitions are program-specific, require significant manual input, and/or lack information necessary to choose placements unambiguously.

First, identifying security-sensitive objects is difficult because any program variable could be a security-sensitive. At present, there is no principled approach to determine which are security-sensitive, so the prior methods proposed to assist in authorization hook placement [36, 70, 71, 72, 73, 37, 74, 75, 76] expect programmers to specify the data types whose variables require authorization manually, which requires extensive domain knowledge. The identification of these data structures is not a trivial task and often takes multiple iterations to get right. For example, the X server version 1.4 did not have hooks for accessing certain classes of objects (such as ”selection” objects), which were added in version 1.5. The set of security sensitive-objects for *postgresql* is still under discussion by the community.

Second, identifying security-sensitive operations upon these objects is difficult because any program statement that accesses a security-sensitive object could be security-sensitive. Traditionally, researchers use the structure member accesses on the security-sensitive user-defined types as security-sensitive operations. However, there are many such operations (9133 in X server), and clearly there are many fewer authorization hooks. In some cases, researchers identify specific types of operations as security-sensitive, such as interfaces to database functionality in PHP programs [74] or method calls in object-oriented languages [77]. However, such interfaces may not be present in the program or may only cover a subset of the security-sensitive operations. Prior methods have identified security-sensitive operations at the level of APIs [73], but this may be too coarse a granularity. For example, hook changes between version 1.4 and 1.5 of the X server mediate finer-grained operations to reduce the privilege given to some subjects.

Finally, once the security-sensitive operations are identified, authorization hooks must be placed. However, authorization hook code should not be scattered through the program. Typically, hook placement efforts aim to achieve complete mediation while optimizing hook placement relative to an access control policy. A naive hook placement would be to mediate all operations at the beginning of the program, but this would result in the subject being authorized to perform *all* operations or will be denied from performing any. For example, when retrieving a window, we could have one hook that authorizes all operations on windows that are possible and all operations on window properties of those windows.

However, this would prevent a subject who is only authorized for a subset of the accesses to a window from performing any accesses. Thus, a placement of authorization hooks must only request authorization queries for the operations that are actually going to be executed given that placement location.

As a result, we state that the *authorization hook placement problem is to find an optimal set of authorization hooks that completely mediates only the security-sensitive operations executed on anypath.* This definition means that solutions depend heavily on this imprecise notion of security-sensitive operation. Ganapathy *et al.* explored techniques to group statement-level operations into sets that represent security-sensitive operations using dynamic analysis and concept analysis from manually highlighted interfaces [72, 73]. Even with this information, the programmer must still make manual decisions about the granularity to place authorization hooks. In this thesis, we propose a novel method for identifying security-sensitive objects and operations that more closely approximates the programmers' intuition about the granularity of security-sensitive operations, as we find by evaluating our approach against manual authorization hook placements in Section 5.5.

2.6.1 Case Study: The X Windowing Server

The X Window System is a windowing system for bitmapped graphics displays based on a client-server architecture. The server controls the display and associated input devices such as the keyboards, pointing devices, touchscreens, and so on, and the clients are the graphical programs that access those services. X provides the basic services for building GUI environments such as drawing and moving windows on the display and interacting with devices. Clients connect to the server via Unix domain sockets (if local) or TCP/IP (if remote) and pass requests to the X server and receive events from the server by a protocol known as the XProtocol. The clients can only communicate with the X server through this connection, and cannot directly communicate with other clients over this connection, albeit actions taken by one client may frequently be visible to others.

The X server also maintains resources as objects that can be shared between clients.

These resources are stored in data structures in the server. Clients name resources by resource identifiers and client applications frequently use these resources to communicate with other client applications. This results in the X server being used as a communications channel. Requests are sent from the client to the server, and replies and errors are sent from the server to the client. Replies contain data requested by a client. X is an event-driven protocol and events can be sent to clients either as part of a request or asynchronously if the client has expressed interest in it. Requests manipulate server-side resources. Requests from clients lead to the creation of resources and an integer valued identifier is used to refer to the resource thereafter. The X server may change that identifier, usually adding a client id to the high bits of the integer. Resources are freed by requests or when connections are closed. Most resources are potentially sharable between applications. Clients normally get resource identifiers of objects in the X server by using various query methods. However, clients can reference unknown resources by guessing at the resource identifier.

Without adequate isolation of clients, security vulnerabilities pertaining to *confidentiality*, *integrity* and *availability* can arise.

- *Confidentiality:* X servers typically offer no protection for an application's output. Clipboard managers can grab content from an application automatically. Malicious applications can currently probe the entire state of the X server. Consequently, attackers can use utilities such as *xspy* [78] to sniff keypresses on remote X servers and *scrot* [79] to take screenshots.

- *Integrity:* Clients can manipulate another client's resources such as windows and can send input, or any other type of event, directly to a client. A malicious client can cause another client to present false information to the user. An attacker could insert malicious commands into the input stream of a terminal emulator.

- *Availability:* Clients can close the windows of other clients, terminate the session of another client, manipulate the font lists, and manipulate the host access lists.

2.6.1.1 Retrofitting the X server for authorization

The X Consortium developed a simple security extension ("Security") in 1996. The Security extension provided a simple two-level, coarse-grained, trust hierarchy for client connections. The "untrusted" clients were restricted in several ways. Although the extension itself was too coarse-grained, its authors did a thorough analysis of the core X protocol, identifying places in which untrusted clients should be restricted and introducing checks into the X server code at those places. Trusted Extensions for Solaris has an X extension (Xtsol) [80] which also added security functionality. In 2003 NSA undertook the task of adding LSM-style security hooks to provide the enforcement logic in the X server. These hooks were added by what is now known as the X Access Control Extension (XACE). Much of the XACE development process consisted of replacing the checks proposed by the X consortium's security checks and Xtsol's security checks with more generic callbacks.

X server stores its resources such as windows, pixmaps, cursors, fonts, and colormaps, in a large hash table. The identifier for resources allows each resource to be assigned a client "owner"- typically the client that requested its creation. The resource system is extensible, allowing X protocol extensions to create new resource types for objects that they introduce. When clients make requests they may name the resource they want the server to operate upon. The request handling code then calls a lookup function to retrieve a pointer to the object itself. The server can perform several different operations on the retrieved object based on the request. There are two kinds of hooks in XACE. First, there is typically a hook present in the resource lookup function. The hook includes as parameters the resource ID to determine the client owner and type of resource, and the client on whose behalf the lookup is being made. This important hook allows security modules to vet all resource lookups. The hook also has an "access_mode" field that determines what operation is being performed on the resource object subsequent to its lookup. Listing 1 shows an example for such a lookup function. Here DIXLOOKUPWINDOW is a lookup function that is passed the subject parameter `client`, the identifier for the object `stuff->id`, and the access mode `DixGetAttrAccess` to check if the client is authorized to read the window's attributes.

Access Mode	**Operation**
DixReadAccess	inspecting the object
DixWriteAccess	changing the object
DixDestroyAccess	destroying the object
DixCreateAccess	creating the object
DixGetAttrAccess	get object attributes
DixSetAttrAccess	set object attributes
DixListPropAccess	list properties of object
DixGetPropAccess	get properties of object
DixSetPropAccess	set properties of object
DixGetFocusAccess	get focus of object
DixSetFocusAccess	set focus of object
DixListAccess	list objects
DixAddAccess	add object
DixRemoveAccess	remove object
DixHideAccess	hide object
DixShowAccess	show object
DixBlendAccess	mix contents of objects
DixGrabAccess	exclusive access to object
DixFreezeAccess	freeze status of object
DixForceAccess	force status of object
DixInstallAccess	install object
DixUninstallAccess	receive from object
DixUseAccess	use object
DixManageAccess	manage object
DixDebugAccess	debug object
DixBellAccess	audible sound
DixPostAccess	post or follow-up call

Table 2.1: Tabulation of access modes in the X server

The lookup function itself will contain a call to XACEHOOK which performs the check. However, since the access mode is passed at the lookup function we consider each lookup function to be a hook by itself. There are also hooks that do not mediate when the object is looked up but rather just before it used in an operation. This typically happens when it is unclear at lookup which operation is going to be performed on the object.

In 2003 Kilpatrick et al [17] presented the XACE specification as a comprehensive set of hooks to be retrofitted into the X Server. It wasn't until 2007 that it was deployed as part

Listing 1 Shows the code for function PROCGETWINDOW in X server to highlight XACE hook

```c
int
ProcGetWindowAttributes(ClientPtr client)
{
    WindowPtr pWin;
    REQUEST(xResourceReq);
    xGetWindowAttributesReply wa;
    int rc;
    REQUEST_SIZE_MATCH(xResourceReq);
    rc = dixLookupWindow(&pWin, stuff->id, client, DixGetAttrAccess);
    if (rc != Success)
        return rc;
    memset(&wa, 0, sizeof(xGetWindowAttributesReply));
    GetWindowAttributes(pWin, client, &wa);
    WriteReplyToClient(client, sizeof(xGetWindowAttributesReply), &wa);
    return Success;
}
```

of X.org mainline. However, subsequent revisions have made many changes to the access control modes and hook placement locations and many hooks were added, removed and moved. For example, in the original version, there were only four possible modes for the ”access_mode” field of the hook: read, write, create, and destroy. However, the subsequent version introduced 28 access modes that were more closely tied to the functionality of the X Server. We show the list of new operations in Table 2.1. The X Server version 1.4.1 released in 2007 had 173 hooks whereas the version 1.9.3 released in 2010 had 239 hooks. This shows that it takes years for a large legacy system to be retrofitted with authorization hooks, with iterative refinements on the placement of authorization hooks.

Prior work to address this problem has focused both on verification of authorization hook placement to ensure complete mediation [35, 37, 74] and on mining security-sensitive operations in legacy code [72, 73]. The work on mining security-sensitive operations is the most closely related to this thesis, and uses static [73] and dynamic program analysis [72] to identify possible hook placement locations. However, these mining techniques rely on domain-specific knowledge (e.g., a specification of the data types that denote security-sensitive objects [73]), providing which still requires significant manual effort and a detailed understanding of the server's code base.

2.7 Authorization Hook Placement

We first discuss the prior work that covers two different but related problems.

1. *Automatic Hook Placement*: In past efforts, this typically involved using manual specification of some combination of subjects, objects operations and even hook code in order to place authorization hooks. In comparison, the only program-specific input specification we need is the few variables that represent the entry point of user requests in the program.

2. *Automatic Hook Verification*: This problem assumes that a certain number of hooks have already been placed correctly. This information is used to derive security-sensitive operations and the approach verifies if all instances of these operations are mediated.

We then discuss how to implement the actual authorization checks.

2.7.1 Automated Hook Placement

Ganapthy et al. [71] presented a technique for automating hook placement assuming that the module implementing the hooks was already available. Their tool also required manually written code-level specifications of security sensitive operations. Using this information, they identified the set of operations that each hook protects and also the set of operations the program performs. Placing hooks is then a matter of matching operations in the code with the hooks that mediate those operations. Following this, they presented a hybrid static/dynamic analysis tool [72] that used program traces of security sensitive operations to derive the code level specifications automatically. For this technique to be effective, the user must know precisely which operations are security-sensitive and gather traces for them. In our tool, we automatically infer the set of security-sensitive operations as well as their code level specification using static analysis. The work most closely related to our approach is the one by Ganapathy et al [73] that used Concept Analysis [81] to group structure member accesses in program APIs in order to identify sets of structure member accesses that were frequently performed together in the program. In contrast to this work, we use a more intuitive technique for grouping structure member accesses into operations which falls out of the user's ability to make choices of objects and operations in the program. We envision that our hooks will therefore be at a granularity that is closer to a manual effort. The concept analysis approach also requires the user of the tool to specify APIs that are entry points into the program and the security-sensitive data structures. We identify the latter automatically.

2.7.2 Automated Hook Verification

There have been several tools to verify hook placement and detect missing hooks that employ both static and dynamic analyses. Zhang et al. [36] used simple manually specified security rules to verify the complete mediation property of reference monitors while Edwards et al. [70] used dynamic analysis to detect inconsistencies in the data structure accesses. Tan et al. [37] start with the assumption that production level code is already fairly mature and most of the hooks are already in place. They verify consistency of hook placement by using existing hooks to characterize security-sensitive operations in terms of structure member accesses and then check for the presence of unmediated operations. Srivastava et al. [82] use the notion that modern API's have multiple independent implementations. They use a flow and context sensitive analysis that takes as input multiple implementations of an API, and the definitions of security checks and security-sensitive events to see if the two implementations enforce the same security policy. In contrast to this, we attempt a first stab at solving the authorization hook problem on legacy programs and in the process define program level notions of security sensitive objects and operations which can also serve as a specification for automatically verifying that the existing authorization hooks in programs are comprehensive.

More recently there has been some work [74, 76] on automating authorization hook verification for web applications that leverage programming paradigms of the web domain as specifications. In RoleCast [74] the authors verify the consistency of access hooks placed in web applications. They assume the presence of existing hooks, and the set of security-sensitive objects and operations are the backend database and database update operations which are the same for all web applications they consider. The applicability of their approach is limited to the programming paradigm where web applications are built around pre-defined roles (such as admin, regular user) and different roles have different mediation requirements. So they use a combination of static analysis and heuristics to identify the portions of the application that implement the functionality relating to different roles. Sun et al. [76] mediate web applications at a much coarser granularity of web pages while also using a specification of roles, and infer the difference between sitemaps of privileged and unprivileged roles as the definition of security-sensitive operations. In contrast, our approach proposes concepts that are applicable uniformly in all domains.

In [67] the authors tackle domain specific logic vulnerabilities for web applications by inferring domain-specific constraints. They first use dynamic analysis to infer likely invariants for the current domain and then use model checking to gather support for the likely invariants. We do not deal with logic vulnerabilities but also try to gather domain-specific definition of security sensitive behavior. In contrast to all these papers, we do not assume a mature code base with hooks already in place. In fact, we take a first stab at the placement of hooks.

2.7.3 Implementing Access Checks

Our analysis approach identifies security-sensitive objects and operations in programs using static analysis and identifies code location where access checks need to be placed in order to mediate them. Programmers may either use libraries and frameworks or choose to implement their own custom checks. It is important to disuss the relationship between our work and aspect-oriented programming. Aspect-oriented programming uses code pattern-matching to identfy to locate places to attach remediation. The advice/remediation is developed independently and is used for different concerns such as error-handling and security. The patterns are called *pointcuts*, the matched code locations are called *joinpoints* and the addition of advice is called *aspect weaving*. In our approach we have already identified locations to place access checks, so we already have the joinpoints.

Implementing access control checks in server programs often entails just placing a call to a method that accepts arguments for subjects, objects and operations. In other domains such as web applications, there security frameworks such as Spring [83] and Apache Shiro [84] are gaining popularity. These frameworks make it easier to add authorization checks at different granularities such as mediating URLs, classes, and individual methods inside classes. Spring security also has the option of adding checks using pointcut expressions which is powerful enough to add checks to several methods using a single expression.

An important consideration in implementing authorization checks is that they need to be small enough to enable easy verification and they should not introduce additional errors in programs. FixMeUp [85] extracts program transformation templates based on existing access checks put in by programmers, uses this to identify fault access control logic and proposes repairs without violating the access control policy and introducing any additional dependencies.

Chapter 3

Resolving information flow errors using graph cuts

3.1 Introduction

An information flow error occurs when the flow of information enabled by a system between two components is in violation of a security policy. Several security problems such as confidentiality, integrity and authorization violations are a manifestation of information flow errors. Some questions that can be formulated as information flow error problems are:

- Can unauthorized users access security-sensitive administrator functionality in an application?
- Does an application allow a malicious user to modify security-sensitive data in a database?
- Does an application accidentally leak user passwords?
- Does the system allow adversaries to control the hypervisor?

We find that even within a single system, information flow errors control must be enforced at different layers. For example, to control the propagation of attacks within and across hosts in a distributed system, mandatory access control (MAC) enforcement has been integrated at different levels of the software stack: at operating systems [86, 87, 24], virtual machine monitors (VMMs) [28, 27], user-level programs [88, 23, 22], and integrated with network access control [89, 90, 91]. The different protagonists of the software development and deployment activities need to solve some instance of the information flow problem. Given a complex distributed system deployment, an administrator needs to identify and fix information flow errors resulting from composing disparate security policies at the different granularities. At the application-level, programmers need to enforce information flow policies to control the interaction of the program's inputs, operations and outputs. A recent trend is the emergence of systems support for applications to enforce security policies. In SELinux, system services, such as GConf, dbus and the X Window server, have all been retrofit with a reference monitor to enforce system policy.

In general, information flow errors are resolved by placing *mediators*. Depending upon the problem, mediators work by either blocking or transforming the offending information flow paths to comply with the security policy. Examples of *transforming mediators* include sanitization routines to remove incorrect user input and encryption to obscure data before storing or releasing it. Examples of *blocking mediators* include access checks or guards that control access to security-sensitive resources or functionality. Mediators may be implemented in different ways: as entire processes, such as guards, or as individual program statements, such as endorsers in security-typed languages [59].

While the identification of information flow errors has been studied extensively, resolving them has been a hard problem to solve. Typically, resolving information flow errors has been a manual process requiring human examination of the offending information flows and attempting to place mediators to solve them. There are several problems in this approach. *First*, the set of mediation statements placed needs to satisfy the *completeness* property, i.e., every offending information flow path must be mediated. This is hard to get right manually and takes significant effort to examine each individual error path being reported, place a mediator and determine iteratively which errors still remain to be solved. It is also possible that an error may have multiple causes and the mediator placed must address them all. *Second*, often placing mediators incurs a performance penalty, so it is important to optimize the placement of mediators. It is hard to manually spot opportunities for optimizing the resolution of errors. For example, it may be possible to resolve multiple errors with a single mediation statement placed at the appropriate location. Can we help the programmer by automatically identifying correct and optimized resolutions to information flow errors?

Traditionally, information flow in programs and policies have been modeled as graphs and information flow errors have been identified by posing reachability queries based on the security policy to be enforced. McCamant and Ernst [92] showed that it was possible to quantify the amount of information leaked by the program by modeling it as a network flow problem on the information flow graph. Inspired by this, we turned to the dual of the approach, *minimum graph cuts* to solve information flow errors. By proving that a cut of the information flow graph is tantamount to complete mediation, we came up with a novel idea to resolve information flow errors [8]. While this initial work dealt with simple two-point lattice based policies, we have subsequently extended this approach to deal with security policies that can be represented as general lattices.

We successfully applied the algorithm to solving information flow problems in two different layers of the software stack. We show how we can solve mediation placement problems in user-space programs by applying it to two Java programs: `tinySQL` and `WeirdX`. Second, we show how we help an system administrator put together a secure deployment of a distributed system by pointing out where information flow errors in the entire system can be resolved most effectively.

There were two big takeaways from these experiments that advised the work in subsequent chapters. First, in order to use this approach to solving information flow errors effectively, we need accurate security policy specification. This is an orthogonal problem and a hard one at that. Second, when we consider the semantics of mediation, authorization checks are fundamentally different from sanitizers and endorsers. We argue that we need a different approach to satisfy the placement constraints of authorization checks.

3.2 Background

In this section, we discuss the *information flow problem* and how information flow errors are typically identified. The information flow problem can be expressed using the model below:

Definition 7. *An* information flow problem, $\mathcal{I} = (\mathcal{G}, \mathcal{L}, \mathcal{M})$*, consists of:*

1. *A* directed data flow graph $G = (V, E)$ *consists of a set of nodes* V *connected by edges* E *where an edge* $(u, v) \in E$ *represents the flow of information between the entities represented by nodes* u *and* v*.*

2. *A* lattice $\mathcal{L}$= $\{L, \preceq\}$*. For any two labels* $l_i, l_j \in L$*,* $l_i \preceq l_j$ *means that* l_i *'can flow to'* l_j*.*

3. *A* label mapping function $M : V \rightarrow \mathcal{P}^L$ *where* $\mathcal{P}^L$ *is the power set of* L *(i.e., each node is mapped either to a set of labels in* L *or to* $\emptyset$*).*

The lattice imposes security constraints on the information flows enabled by the data flow graph. If information can flow between a pair of vertices such that there is no 'can flow to' relationship between any pair labels in the label mapping of the vertices is considered an error.

Definition 8. *Each pair* $u, v \in V s.t.\ [u \hookrightarrow_G v \wedge (\exists l_u \in M(u), l_v \in M(v).\ \ l_u \npreceq l_v)]$*, (where* $\hookrightarrow_G$ *means there is a path from* u *to* v *in* G*), is an* information flow error.

It has been shown that information flow errors in programs [93] and MAC policies [94, 95, 96] can be found automatically using such a model by posing graph reachability questions on the labels in the lattice. The lattice and the label mapping to be used are dependent upon the domain and security property to be enforced.

3.3 Approach

In this section, we describe the technical underpinnings of our approach to solving the information flow problem.

3.3.1 Cut Mediation Equivalence

In graph theory, a graph cut partitions the set of vertices in the graph into disjoint sets. A cut-set is the set of edges whose ends are in different subsets after the partition. In network flow, the cut is the set of vertices/edges whose removal prevents any flow from the source to the sink. We have shown [8] that a vertex cut of the information-flow graph corresponds to the set of locations where mediators need to be placed to completely resolve all information flow errors.

Definition 9. *Placing a mediator to resolve information flow errors for a lattice policy containing two levels l_i and l_j is tantamount to generating an edge cut of the information flow graph with the nodes mapped to l_i as the sources and the nodes mapped to l_j as the sinks.*

Theoretically, we show that mediation statements are special nodes in the information flow graph that are capable of changing their own label mapping (a mediator for (l_i, l_j) transforms the label on the data to $l_k \preceq l_j$) such that once data passes through it, it will no longer cause an information flow error.

While this approach is tractable for two-point lattices, solving this problem for general lattices is cumbersome since it creates a set of source-sink cut problems that need to be resolved.

Definition 10. A cut problem set, *is a set of lattice label pairs C defined as $C = \{(l_i, l_j) \mid l_i, l_j \in L \wedge l_i \npreceq l_j\}$. A cut problem (l_i, l_j) is realized in the information flow graph G if $\exists\, u, v \in G \mid u \hookrightarrow_G v \wedge l_i \in M(u) \wedge l_j \in M(v)$. The problem of finding the minimal cut for a cut problem set is called the* multiway cut problem.

3.3.2 Resolving errors in general lattices

Researchers have shown that the multicut problem is NP-Hard for directed graphs [97]. In the context of security lattices, we previously suggested a simple greedy solution to the problem that returns the union of the solutions for each individual cut problem. However, we found that we solving the cut problems in a specific order taking into account the mediation semantics might be a better approach since it allows us to run the mincut algorithm on a potentially smaller subgraph for each lattice label.

Let $V_i \subseteq V$ and $V_j \subseteq V$ but the subset of vertices in G that have $l_i \in L$ and $l_j \in L$ in their label mappings respectively. For a cut problem (l_i, l_j) to be realized in the graph, there needs to exist a path from some node in V_i to some node in V_j. Each cut problem (l_i, l_j) is therefore *realized* in a subgraph of the information flow graph called the *cut-subgraph* constructed by pruning out all paths in G that are not a part of some path between a node in V_i and a node in V_j.

Definition 11. *A cut-subgraph for the cut problem* (l_i, l_j) *and the information flow graph* G *is a subgraph* $G_{ij} = (V' \subseteq V, E' \subseteq E)$ *where* $\exists u, v, w \in V \mid l_i \in M(u) \wedge l_j \in M(v) \wedge u \hookrightarrow_G v \hookrightarrow_G w$.

Mediation dominance: Let G_{ik} be the subgraph where the cut problem (l_i, l_k) is realized in the graph G. Our solution is based on the insight that when we solve the cut problem (l_i, l_k), we have already solved the cut problem (l_i, l_j) where $l_k \preceq l_j$ for the cut-subgraph G_{ik}. By solving (l_i, l_k) first, the portion of the cut problem (l_i, l_j) that remains to be solved is only on the subgraph $G_{ij} - G_{ik}$. Therefore, by solving the cut problems for l_k before l_j, we get two advantages: (1) the size of the cut problem we solve $G_{ij} - G_{ik}$ is at most the size of G_{ij}. This means that we may end up solving a smaller problem for (l_i, l_j) by considering it in the context of the mediation necessary for (l_i, l_k) and, (2) the size of the solution set using the ordered approach is at most the size of the solution without considering the order. This means that if there is an overlap in the graph between the different cut problems of *comparable* lattice levels, then the size of solution set using the ordered cut approach can

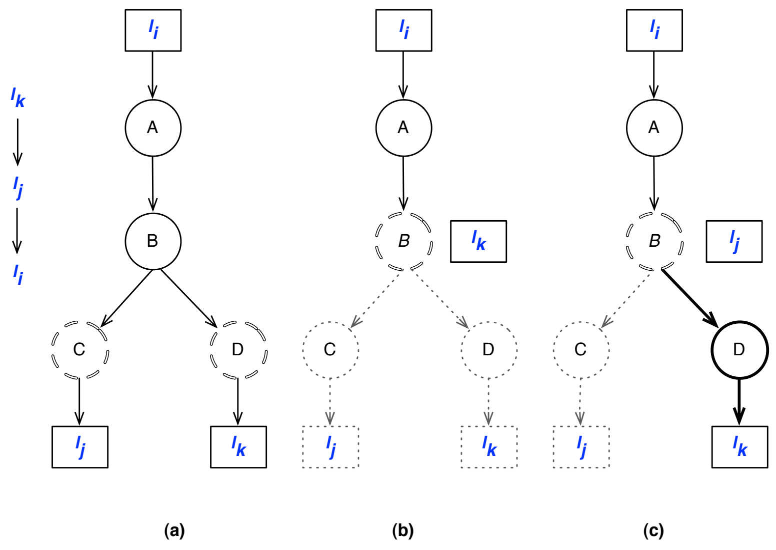

Figure 3.1: Ordering the cut problems

be smaller than the naive solution, a *union* of the individual solutions.

Figure 3.1 illustrates this. When we solve the cut problems for l_j and l_k if the algorithm picks C and D respectively as the mediators as shown in (a), then it doesn't make a difference which order the cut problems are solved. If the algorithm picks A or B then it better if l_k is solved before l_j. The reason is that if we solve l_j before l_k, then B will change the label of the data it receives to l_j, so we will still need to solve the problem (l_j, l_k) between B and D as shown in (c), whereas if we solved the problem in reverse order, then B would change the label of the data to l_k and since $l_k \preceq l_j$ there won't any problem left to solve.

Our algorithm shown in Figure 3.2 solves the general lattice cut problem in the topological sort order of the lattice in order to take advantage of the *mediation dominance* property. Topological sort imposes a linear ordering of the vertices of a directed acyclic graph (DAG) such that if the graph contains an edge (u, v) then u appears before v in the ordering. REALIZABLE generates the cut-subgraph of G for the problem. To solve the cut problem for each label l_j, we only need to solve the problem for the portion of the cut-subgraph that has not already been solved. MINIMUMCUT generates the minimum cut of the input graph for the given sources and sink.

```
MEDIATIONRESOLUTION($G, \mathcal{L}, M$) {
  $L_T \leftarrow TopSort(\mathcal{L})$
  for $(l_k \in L_T)$
    do $\{Sources = \{l_i \in \mathcal{L} \mid l_i \not\preceq l_j\}$
      for $(l_i \in Sources)$
         do $G_{ij} =$ REALIZABLE$(G, l_i, l_j)$
           for $(l_j \preceq l_k)$
              do $G_{ij} = G_{ij} - G_{ik}$


      $G_k = \bigcup_{i \in Sources} G_{ij}$
      $Mediators = Mediators \cup$
        $MinimumCut(G_{ij}, l_j, Sources, M)$
      }
```

Figure 3.2: Greedy Mediation Resolution Algorithm

The running time of the algorithm is governed by the size of the lattice $|L|$ and the time taken to run the min-cut algorithm. The choice of the min-cut algorithm is important. Using Dinitz's algorithm gives us $O(|L| * |V|^2 * |E|)$. The topological ordering helps us improve $|V|$ and $|E|$ by potentially giving a smaller subgraph of the problem to solve with each subsequent cut problem. Note that the it can be shown via counterexamples that the algorithm above does not produce an optimal solution. The problem of finding approximation algorithms to the problem of lattice cuts is still open.

3.3.3 Adding constraints to the cut problems

In order to solve information flow errors using graph cuts, we need the cut algorithm to handle additional constraints. First, the problem instance may dictate that some vertices cannot perform mediation and must therefore not be included in the solution. Therefore, we need a way of specifying that some nodes are unfeasible as mediators. Second, nodes may be limited in their mediation capabilities. For example, there may be a maximum sink label upto which a node can perform mediation, so that a each node may be a viable mediator only in some subset of the cut problems that are solved. In order to handle such constraints, we can adjust the capacity of edges in the graph appropriately. We can solve the problem as follows:

- First, we associate a map called RAISELABELS from each vertex v_i in the graph to $L_i \subseteq L$ indicating the set of labels that the vertex can mediate for.
- Second, for each cut problem (l_i, l_j) we find the subset of nodes that can be mediators for this cut problem. $V_j = v \in V \mid l_j \in \text{RAISELABELS}(v)$.
- Third, we convert the vertex cut problem into an edge cut problem. For each vertex $v_j \in V_j$ we create a dual vertex v'_j and add the edge (v_j, v'_j) to the graph.
- Finally, we assign an appropriate capacity to the newly created edges. The simplest idea is to set the capacity of these edges to one and set the capacity for all other edges to infinity. We may also be able to assign relative weights to edges in to indicate preference of some edges over others.

3.4 Solving real-world information flow errors

In this section, we show how we used the graph cut approach to solve real-world instances of information flow errors. In order to apply this approach to real world problems we need to create an instance of the information flow error problem. The four steps are described below and are largely domain and problem specific.

- First, we need to generate a specification of the security property that we want to enforce. This may be any arbitrary property that can be represented as a lattice. Common security properties include confidentiality, integrity, authorization etc. The lattice generated is typically application specific as our experiments show.
- Next, we need to generate an information flow graph of the system/program. We need to identify what entities the nodes in the graph represent. The edges will represent

information flow between the node entities. For programs, these may be variables, statements, commands, etc. For system level MAC policies these will typically be the kernel and system software. For a network, these will be the individual hosts in the network.

- Third, we need to generate a mapping from the labels in the lattice to the nodes in the graph. This will depend both on the semantics of nodes in the graph and the security policy that the lattice represents.

- Finally, we need to specify the optional mediation constraints in the form of RAISELABELS and edge capacities.

Given this input, the technique will return a) the set of nodes such that by placing mediators at these nodes the information flow errors will be completely resolved, and b) mapping from each mediator to the lattice labels for which it performs mediation. The actual task of placing the mediators is upto the programmer.

First, we discuss how we applied the approach to solve information flow errors in user-space programs and discuss some of the results. Next, we describe how we applied this approach to resolve information flow errors in composing policies from different components in a distributed system.

3.4.1 Resolving Information flow errors in programs

To run the automated mediation placement tool, it is necessary to assign security semantics to program elements. The programmer labels security-relevant data structures with security labels. 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`. However, the programmer must manually associate a security label with certain program locations such as protected data structures.

Next, an interprocedural analysis infers labels for unlabeled program elements. This analysis builds a set of *information-flow constraints* that captures how code interacts. This set of information-flow constraints is then transformed into an *information-flow graph*. Third, the programmer then needs to determine the set of *security* and *placement* requirements that need to be satisfied.

3.4.1.1 Specifying Security Requirements:

The programmer needs to identify the security property to enforce in the program. To make this more concrete, let us assume that program variables are assigned labels at runtime. Suppose that a program variable is assigned a security label l_v at runtime (e.g., the file from which the variable's data originated) and the current client is assigned a security label l_c (e.g., the authenticated identity of the client). There are several security decisions that may be made in the program.

- **Secrecy:** if l_v is of higher secrecy than l_c is allowed to access, then information labeled l_v should not be released to the client without employing a declassification process. Declassifiers permit controlled leakage of some l_v data to clients at l_c. An example of a declassifier is a function that encrypts data before sending it on a public output channel.

- **Integrity:** if l_c is of lower integrity than l_v, then information from the client should not be inserted into l_v without employing a filtering process. Filters (or endorsers) perform sanitization operations that either discard data or upgrade its label. Endorsers are the dual of declassifiers, as integrity is a dual to secrecy.

- **Runtime Authorization:** a security policy can specify which actions clients of label l_c are allowed to perform on a variable with label l_v. For example, a client might be allowed to read from a variable, but not delete entries from it. Since the labels are not known until runtime, a runtime authorization query to the reference monitor is necessary.

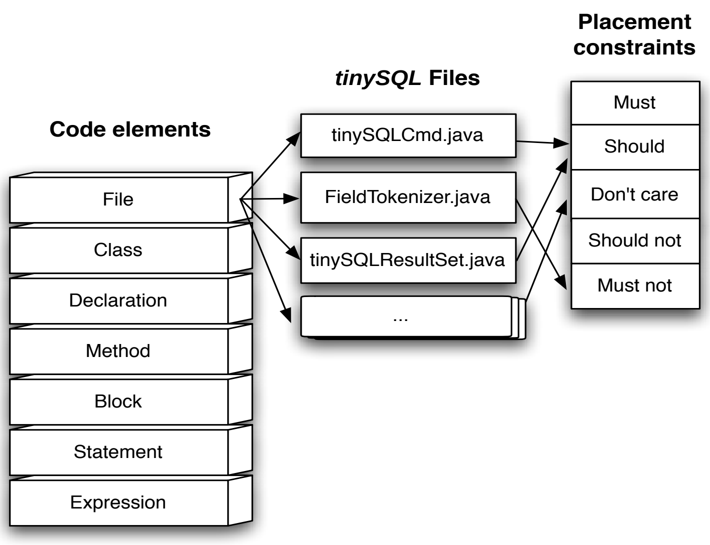

Figure 3.3: Model for specifying placement requirements

3.4.1.2 Specifying Placement Requirements

The enforcement of security properties must also satisfy *placement requirements*. We envision that this is either used to capture *programmer knowledge* about the program or comes from some *external analysis* of the program. These requirements are encoded as constraints to guide the approach towards placing more usable mediators and will affect the capacities assigned to edges in the information flow graph.

First, programmers may have a strong intuition about some parts of the application; so they may choose to put a subset of mediators at locations they think best. The approach must work around such specification. Next, there may be some obvious locations in the code where mediators must not be placed. For example, it is probably not feasible to place mediators inside library routines and therefore files or functions that come from the library must not have any mediatable edges. Finally, additional external analyses such as a static dominator analyses or runtime code coverage analyses may be used to extract should or should not mediate constraints for vertices in the information flow graph.

Figure 3.3 shows the model for representing program-specific placement requirements. 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.

3.4.1.3 Placing mediators in programs

Once properly annotated, the programmer next uses the automated approach described in Section 3.3 to select mediation points constrained on the placement policy generated in the last two steps. After being run, the tool returns a set of suggestions, each consisting of individual mediation points. If the programmer places mediation statements at each of the points contained in a suggestion, the resulting program will contain no programmatic flows that violate the chosen security policy at runtime. If the selected points are not satisfactory, the programmer can iterate this procedure by modifying the placement policy and re-running the placement tool.

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. For example, for an integrity policy the programmer must specify the filtering mechanism that transforms low-integrity data to high-integrity data. Mediation points may also be clustered together to correspond with a higher-level policy check. 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 [73]. We elaborate on this in the discussion section.

Our tool presently places mediation points at the level of expressions and statements. If the results of automatic placement indicate that several expressions in the same block require mediation, it may be beneficial to mediate the entire block instead. Similarly, if many blocks in the same method require mediation, it may be better simply to mediate at the method level. In this way we expect to *hoist* several mediation points that occur at the granularity of one low-level code element to a higher-level code element.The programmer may write custom mediators for each error or choose to use appropriate library functions. The programmer could also use aspect-oriented programming to insert the mediators. Our solution essentially produces information for specifying point-cuts.

3.4.1.4 Evaluation of graph cut approach in programs

We now show how the framework applies to `tinySQL`, an SQL server written in Java. `tinySQL` supports a subset of the SQL language: clients make requests and the database returns information or is modified based on these requests.

Generate Security and Placement constraints If each table is given a security label l_t and each client is associated with a security label l_c, then we created secrecy, integrity and authorization in accordance with the description in Section 3.4.1.1. For tinySQL, the inputs and outputs of the system are handled by the Java IO `System.in` and `System.out` objects. For secrecy, we would assign these a public security label p. For integrity, we would assign `System.in` with a tainted integrity label t. For an authorization check, we would give these data structures client label c, and also mark the `main` method as having client level c, as this is where the client enters the system. The protected data structure in tinySQL is a table, stored in the classes `tinySQLTable` and `dbfFileTable`: for secrecy and authorization policies this class would have a secret security label s; for integrity it would have an untainted integrity label u.

It is important to enforce an integrity policy that filters tainted information as soon as it enters the system. Here an integrity policy would add the placement constraint that before use, data from the user must be filtered and endorsed. For an authorization policy, we must ensure that we do not add an authorization check for a security-sensitive operation that will not be performed. This requires an analysis to compute the dominators [98] of the control-flow graph. If a security sensitive operation o does not dominate a code element e, meaning that there are runtime paths through e that do not execute o, then a mediation statement for o should not be placed at e. Running a dominator analysis on the program will eliminate checks from occurring at positions that would cause the existing program to stop functioning.

The method `displayResults` in `tinySQL` outputs the results from a query to the screen. As this function is part of the communication framework in `tinySQL`, we consider this an inappropriate place for a mediation statement, since it is difficult for the programmer to tell exactly what data is being released to the client. As a result, we would specify that the method `displayResults` has a *must not* constraint.

Placement Mechanism and Results We ran our automatic placement analysis on tinySQL for secrecy and integrity; we have not yet implemented the functional constraints required to select points for an authorization policy, although this would be straightforward. For secrecy, four mediation points were selected, corresponding to where the name of a table is accessed, where the result of a query is sent to the user, where a container class containing a set of results is pass through a method, and where the number of columns is indirectly returned from a query by the number of results sent to the user. For integrity, one mediation point was selected in accordance with the security placement policy: on immediately entering the system, programmer input should be mediated.

3.4.2 Resolving Information flow errors in distributed system deployments

To produce a mediator placement system-wide, we need to construct a system-wide information flow problem and compute a placement that resolves all the information flow errors in that problem. A modern system deployment now consists of reference monitors at several components (e.g.,firewall, OS, program) independently enforcing access control policies. When enforcing mandatory access control, these policies represent the possible data flows among subjects and objects governed by their respective reference monitors.

3.4.2.1 Creating a system-wide information flow graph

We automatically construct a hierarchical data flow graph consisting of components where network is at the highest level, followed by hosts and programs. Each component represents an entity such as an operating system or program by its set of entrypoints, a module containing internal authorized data flows, and exitpoints. A program entrypoint is a program instruction that receives input from the operating system, such as the caller of the library function that invoke a system call. For operating systems, the MAC policy they enforce creates a subgraph within the module. For programs, if the program is information-flow aware, its internal information flows can be represented the same way as those of the operating system MAC policy, otherwise it is a single node.

Mapping firewall rules to applications: Firewall policies limit how adversaries may access the host by port, but the specific host processes using those ports are not identified explicitly. We find though that such connections are either well-known or can be derived automatically, so administrator specification is unnecessary. In the web application, many subjects can be inferred by the use of privileged ports. Also, given the emergence of purpose-specific VMs, the subjects that can possibly use network resources can be easily identified (e.g., included with its specification).

Connect Program Entrypoints to subject and object labels: We use runtime analysis to collect these data flows in a manner analogous to the construction of MAC policies from the permissions programs request [99, 33]. Thus, to construct a system-wide data flow graph for the web application, no administrator specification is necessary, if the distributors of purpose-specific VMs include connections between the network and VM processes (manual) and between program entrypoints and MAC labels (runtime analysis).

3.4.2.2 Creating an integrity lattice

In order to construct an information flow problem, administrators must produce an integrity lattice and label mapping function to the appropriate subjects and objects in each of the security policies in the distributed system. Typically, once a mapping is created for a reference monitor, it can be reused for different deployments. Therefore, creating mappings requires a one-time effort by the administrator. We have defined an integrity lattice and label mappings for the web server, database, and privileged VM, and have been able to reuse them unchanged for various web application deployments.

3.4.2.3 Placing mediators in distributed system deployments

Once the information flow problem is constructed, we must examine the possible placements for mediators. In the past, researchers have considered mediators at the level of processes (e.g., for system MAC policies [94, 95, 96]) and program expressions (e.g., for information flow languages [59]). As discussed in the previous section, using an entire process as a mediator requires risk, as nearly all programs are not formally-assured. On the other hand, assuming that any program expression in a distributed system may be equally qualified to mediate information flow errors, likely makes the information flow problem intractable.

As a guide for where to place mediators, we turn to the Clark-Wilson integrity model [100], which consists of rules that define the high integrity operation of a system. Of particular interest are the rules that define how high integrity data is processed securely, where: (1) high integrity data (CDIs) must satisfy *integrity verification procedures* (IVPs) (Clark-Wilson rule C1); (2) only approved programs called *transformation procedures* (TPs) may modify high integrity data (C2, E1); and (3) TPs may only receive low integrity data (UDIs) if that data is upgraded or discarded (C5). That is, TPs protect data integrity, but they require IVPs to validate that the data an application depends upon is high integrity and TPs must be capable of mediating low integrity inputs to upgrade or discard such data. Thus, Clark-Wilson identifies two types of mediators, IVPs and the program entrypoints of TPs. Thus, we focus on using IVPs to block some local threats (e.g., where the legal values can be identified) and program entrypoints to mediate the remaining threats. We note that the resulting information flow policy is only an approximation of Clark-Wilson integrity, however, because in the model definition IVPs and TPs must be formally-assured, where commodity programs are not. Where available, we still want to leverage information-flow aware programs [59], but where not available, we can still produce a valuable mediator placement that approximates Clark-Wilson integrity.

3.4.2.4 Evaluation of graph cut approach in distributed system deployments

The experimental system includes two hosts, for the web server and web client, which run Xen VM system 4.1. The web server host runs three VMs: a privileged VM (domain 0), a web server VM including an Apache web server and web application, and MySQL database VM. The client host runs two VMs, each configured to run a web browser. Each VM runs the Linux 2.6.31-23-generic kernel. The Xen hypervisor enforces XSM/Flask policies, and each of the VMs enforce SELinux policies. While all of the OS policies are SELinux, they are independent in the sense that each policy supports a distinct set of applications and these policies do not refer to the interactions among VMs. Also, each VM runs an iptables firewall to govern network communications. We assume that secure communication (e.g., IPsec or SSL) is used to protect any channel that carries application data between hosts. Unprotected channels are given the `External` level (i.e., system-low)

In addition to separating the functionality of different security levels using VMs, we also use features of the SELinux MAC system to protect the web application (on the server) and web browser (on the clients) processes further. For the web application, we use the mod_selinux module for Apache to generate separate web application processes to communicate with protected VMs, regular VMs, and other external parties. For clients, we use SELinux policy booleans to set more restrictive permissions. For web browsers, the base permissions (i.e., booleans are include access to system files (e.g., /etc and /var) and the user's home directory (e.g., for plugins).

In our evaluation, we look at two types of mediation: a) Infrastructure mediation, which shows how much mediation is required for the VM independent of its deployment, and b) additional mediation required for the specific deployment characterized as local and remote threats.

VMs	Default	Remote	Local
Database Server	281	15	-
Web Server	217	24	56
Privileged VM	335	0	-
Protected Browser	372	15	47
Regular Browser	371	15	-
Total	1576	69	103
Unique Mediators	525	27	85

Table 3.1: The number of mediators needed for: (1) infrastructure mediation per VM (default); (2) remote threats when application is deployed; and (3) local threats to deployment. For "unique mediators" we count each mediator only once even if it is used in a different VM.

Infrastructure Mediation: Table 3.1 (Default) shows our prototype's estimate of the minimal mediation provided by each infrastructure component. For example, the web server VM requires that at least 217 program entrypoints provide mediation to protect the web server process and kernel integrity from remote threats based on the default firewall and MAC policies and given the runtime trace. The estimates for other VMs are somewhat higher, indicating that the web server may be easier to defend than the others. We also found that many of the mediators are common across VMs. There are 525 unique mediator entrypoints across all programs in these VMs (i.e., a program may appear in multiple VMs). In particular, 161 entrypoints are common across all VMs, which is approximately 50-75 percent of the mediator entrypoints in each of the infrastructure VMs.

Remote Threats to Applications: Table 1 (Remote) shows the entrypoint mediation required to block remote threats when the web application is deployed on this infrastructure. While new mediation is required in most VMs, only 27 new, unique entrypoints need to be protected due to the application deployment. 11 entrypoints are necessary for the new web application programs themselves to receive untrusted input.

Local Threats to Applications: We identify local threats as those objects (MAC object labels) that may be modified by subjects (MAC subject labels) that are not trusted by the target (web browser or web server). These untrusted subjects are mainly user subjects for running ad hoc programs on these VMs. We only consider local threats for the web server and browser. For the web server, 56 more mediators are required, whereas for the browser 47 more mediators are required. As can be seen, most of these mediators are unique (85 out of 103), which differs from the mediators selected for infrastructure and remote threat mediation.

3.5 Discussion

In this section, we discuss some of the limitations and lessons learned from applying this approach to real world problems.

Inferring Specification: In order to apply our graph cut approach, the programmer first needs to identify domain/application-specific data in order to create the information flow problem. The completeness guarantee that our application can provide is contingent on the completeness of the programmer's specification. Inferring program specification in order to run the analysis is an orthogonal problem that needs to first be solved in order to apply this technique more effectively. In the next chapter, we will discuss how we can approach this problem in a systematic manner. We focus on the problem of authorization hook placement and show that it is possible to use a principled approach to infer specification of security-sensitive objects and operations in programs in a mostly automated fashion.

Semantics of Authorization Checks: During the course of our experiments, we also found that authorization checks are a fundamentally different type of mediator compared to declassifiers and endorsers since they block rather than transform information flows. They seek to answer the question "Is this information flow permitted for a given subject"? To allow/disallow the information flow in blocking mediator, it is not necessary to mediate at the information flow. We can in fact mediate as soon as we know that the information flow is going to occur, whereas in the case of sanitizers and endorsers which transform the information flow each individual flow must be mediated.

'Hoisting' mediators can achieve several purposes. First, it allows the program to determine as early as possible if the requested information flow can even succeed. Second, if two instances of the same information flow may occur together, it is not necessary to have a separate mediator for each of them. A single access check will suffice. Third, in programs access control is rarely enforced at the granularity of information flows of variables or expressions, but rather at the granularity of security-sensitive operations that are a set of reads and writes of security-sensitive objects. Therefore, it often makes sense to hoist mediators to mediate entire blocks/methods.

On the other hand, hoisting is not always possible. If a program location, when executed, can perform one of two security-sensitive operations o and o', then a mediation statement to check whether or not the client can perform o should not be performed at that code element. Otherwise, a client attempting to perform action o' may not be authorized to perform o, and so the mediation statement alters the semantics of the original program. 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.

In the next two chapters, we focus entirely on the authorization placement problem and seek to deal with these issues and come up with a technique to place authorization checks automatically.

Chapter 4

Leveraging ‘Choice’ to automate authorization hook placement

4.1 Introduction

The main contribution of this chapter is a novel automated method for placing authorization hooks in server code that significantly reduces the burden on developers. The technique can identify optimal hook placements, in a manner that both minimizes the number of authorization hooks placed, as well as the number of authorization queries generated at runtime, while providing complete mediation. To develop the technique, we rely on a key observation that we gleaned by studying server code. In a server, clients make requests, which identifies the objects manipulated and the security-sensitive operations performed on them. *We observe that when a client makes a* ***deliberate choice*** *of an object from a collection of objects managed by the server, that automatically signals the need for authorization.* What security-sensitive operations are performed on the retrieved object(s) is determined by the code path that the server takes, which is also an upshot of the user’s choice.

Based upon this observation, we design a static program analysis that tracks user choice to identify both security-sensitive objects and the operations that the server performs on them. Our analysis only requires a specification of the statements from which client input may be obtained (e.g., socket reads), and a language-specific definition of object containers (e.g., arrays, lists), to generate a complete authorization hook placement. It uses context-sensitive, flow-insensitive data flow analysis to track how client input influences the selection of objects from containers: these are marked security-sensitive objects. The analysis also tracks how control flow decisions in code influence how the objects are manipulated: these manipulations are security-sensitive operations. The output of this analysis is a set of program locations where mediation is necessary. However, placing hooks at all these locations may be sub-optimal, both in terms of the number of hooks placed (e.g., a large number of hooks complicates code maintenance and understanding) and the number of authorization queries generated at runtime. We therefore use the control structure of the program to further optimize the placement of authorization hooks.

We have implemented a prototype tool that applies this method to C programs using analyses built on the CIL tool chain [101]. We have evaluated the tool on programs that have manual mediation for comparison, such as the X server and *postgresql*. We also evaluated the tool in terms of the minimization of programmer effort it entails and the accuracy of identification of objects and security sensitive operations in programs.

To summarize, our main contributions are:

- An approach to identifying security-sensitive objects and operations by leveraging a novel observation — that a deliberate choice by the client of an object from a collection managed by the server signals the need for mediation.

- The design and implementation of a static analysis tool that leverages the above observation to automate authorization hook placement in legacy server applications. This tool also identifies optimization opportunities, i.e., cases where hooks can be hoisted, thereby reducing the number of hooks in the source code, and by eliding redundant hook placements that would otherwise result in extra authorization queries at runtime.

- Evaluation with four significant server applications, namely, the X server, postgresql,

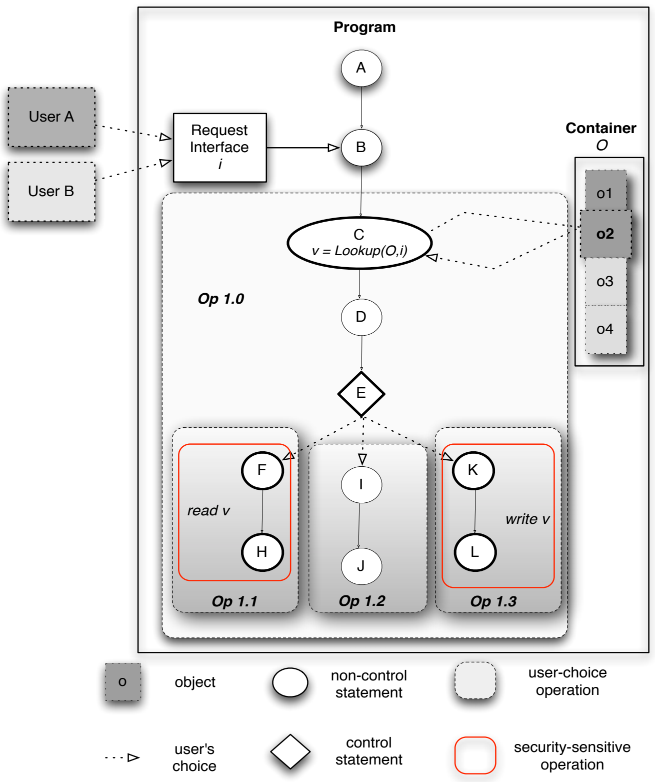

Figure 4.1: Program model of the authorization hook placement problem

PennMush, and memcached, demonstrating that our approach can significantly reduce the manual burden on developers in placing authorization hooks. In case of two of these servers, the X server and postgresql, there have been efforts spanning multiple years to place authorization hooks. We show that for these servers, our approach can automatically infer hook placements that are comparable to those placed manually with few false positives and negatives.

4.2 Approach Overview

Figure 4.1 shows the insights used to motivate our solution approach. Authorization is needed when only a subset of subjects should be allowed to access particular program objects or perform particular accesses on those objects. Figure 4.1 shows that objects $o1$

and $o2$ are accessible to a subject User A, but $o3$ and $o4$ are not. Further, the subject may not be allowed to perform all operations on her accessible objects. For example, User A may only be allowed to perform a read operation on object $o1$, while she can both read and write object $o2$. The choice of authorization hook placement must ensure that the program can only perform an operation after it is mediated for that operation, while ensuring that a subject is authorized only for the operation she has requested to perform. Thus, the statements F and H should only be mediated for read operations, whereas the statements K and L should only be mediated for write operations.The statements I and J do not require mediation as they perform no security-sensitive operations.

The program's behaviors on behalf of subjects are determined by the subject's *user requests*. We find that by tracking user requests we can identify the set of objects that require mediation because such inputs guide the selection of objects for processing. Also, by tracking user requests, we can identify operations the program performs because such inputs choose the program statements that manipulate objects. We detail the method to track user request input in Section 4.3.1.

Programs that manage objects on behalf of multiple users typically store them in *containers*. When a subject makes a request, the program may use the request input to choose the objects to retrieve from some containers, potentially resulting in access to a data used by another subject. As the retrieved values are assigned to program variables, these variables represent the program's security-sensitive objects. In Figure 4.1, variable v in statement C is security-sensitive because the user request input i is used to retrieve an object from the container. By tracking the dataflow from user requests to the selection of objects in containers, we can identify the variables that hold these objects in the program. We detail the method to identify security-sensitive objects in Section 4.3.2.

The program executes statements chosen by its *control statements*. If a user request affects the values of the variables used in a control statement's predicate, then the subject can choose the program statements that may access security-sensitive objects. We call the sets of program statements that may be chosen by subjects *user-choice operations*.

As shown in Figure 4.1, statement E is a control statement. It is shown that three user-choice operations result if E's predicate is dependent on the user request. In addition, the choice of an object from a container is also a user-choice operation. Only the user-choice operations that contain accesses to variables that hold security-sensitive objects represent security-sensitive operations, which are just operations `read` v and `write` v. These are the operations that require mediation via authorization hooks. We detail the method to identify user-choice operations in Section 4.3.3.

Using a naive placement of an authorization hook per security-sensitive operation may lead to sub-optimal hook placement. For instance, if all three user-choice operations at E perform the same security-sensitive operation, then we could place a single hook at E to the same effect as placing a separate hook at each branch. Also, if the same security-sensitive operation is performed twice as part of a single request, we only need to authorize it only once. Our solution to the authorization hook placement problem optimizes hook placement by removing redundant mediation in two ways. First, we remove hooks from sibling operations (i.e., user-choice operations that result from the same control statement) if they mediate the same operations, which we call hoisting common operations. Second, we remove any mediation that is already performed by existing hooks that dominate the operation, which we call removing redundant mediation. We detail these optimizations in Section 4.3.4.

4.3 Design

Figure 4.2 shows the design steps in our approach. The primary input is the source code of the program along with a manual specification identifying the program variables where user request enters the program. We discuss the other inputs in the subsequent sections.

In Stage 1, we identify the set of program variables tainted by the user request inputs, called *tainted variables*, using an inter-procedural static taint analysis [55]. The results of this analysis are used in the subsequent stages to identify objects and operations chosen by

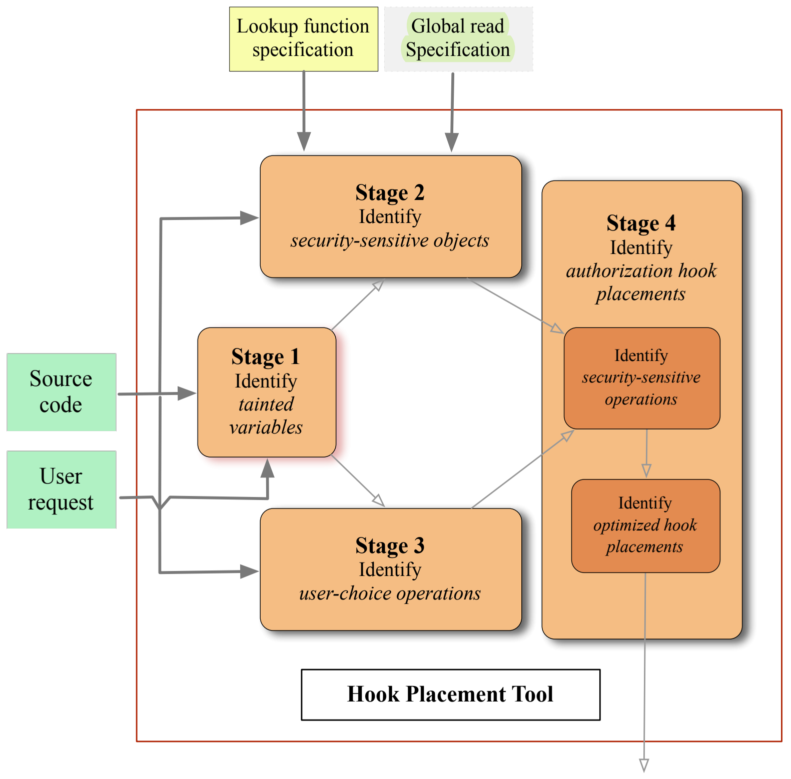
Lookup function specification
Global read Specification
Stage 2
Identify
security-sensitive objects
Stage 4
Identify
authorization hook placements
Source code
Stage 1
Identify
tainted variables
Identify
security-sensitive operations
User request
Identify
optimized hook placements
Stage 3
Identify
user-choice operations
Hook Placement Tool

Figure 4.2: Shows the sequence of stages in automated authorization hook placement. The light shaded boxes shows the required programmer specification for every program, the dark shaded box shows language-specific specification, and the dashed box shows optional specification.

the user.

In Stage 2, we identify the set of program variables that represent *security-sensitive objects*. We use the tainted variables identified in Stage 1 to find variables whose assigned values are retrieved from container data structures using tainted variables.

In Stage 3, we identify the set of *user-choice operations*. We use the tainted variables from Stage 1 to identify control statements whose predicates include tainted variables. A user-choice operation is created for each conditional branch that is dependent on the control statement. Each user-choice operation is represented by a subgraph of the program's control dependence graph [102] for that conditional branch.

In Stage 4, we combine the security-sensitive objects identified in Stage 2 and the user-choice operations identified in Stage 3 to identify *security-sensitive operations*. A user-choice operation is security-sensitive if its statements access variables that represent security-sensitive objects. The actual accesses that take place in a security-sensitive operation determine the authorization requirements that must be approved using the authorization hook. We then propose two techniques to remove unnecessary and redundant authorization hook placements.

4.3.1 Stage 1: Tracking User Requests

The goal of this stage is to identify the set of all variables in the program that are dependent on the user request variables; we call this the set of *tainted variables*. To identify which program variables are dependent on the user request input, we first need to identify the user request variables in the program. We obtain the user request variables from the programmer as a manual specification of the variables that signify where user requests are read by the program. For example, in X server the READREQUESTFROMCLIENT function performs a socket read to obtain the request from the client and initializes a variable representing the request. We provide a specification that this variable represents the user request. We discuss the identification of user requests in Section 4.4.

We compute the set of tainted variables as follows. Let P be the program we wish to analyze, $S(P)$ be the set of all statements of P, and $V(P)$ be the set of all variables in P. We want to identify the set $\mathbf{V_T(P)}$ of all *tainted variables* in the program. Let $V_I(P) \subseteq V(P)$ be the set of variables that represent the manual specification of the user request variables. Let $\mathcal{D}(v, v')$ be a relation which is true if variable v is data-flow dependent on variable v'.

Definition 1. *The set of tainted variables* $\mathbf{V_T(P)}$ *is the transitive closure of the relation* $\mathcal{D}$ *from the user request variables* $V_I(P)$.

Taint analysis is used extensively in program analysis, frequently to detect security vulnerabilities. It can identify the set of program variables that are data-dependent on any input variables. Static taint analyses can approximate this information although factors such as aliasing and polymorphic types lead to imprecision. Nevertheless, they have been successfully used in various projects to identify security vulnerabilities [103, 52, 104].

Chang *et al* [55] have used static interprocedural context sensitive taint tracking to compute the set of program values data-dependent on network input. We adopt the same approach, but the taintedness that we track at each variable indicates whether it is data-dependent on user request. This technique uses procedure summarization to avoid the exponential blow-up that can occur during context-sensitive analyses. A procedure summary is a succinct representation of some procedure behavior of interest as a function parameterized by the input variables of the procedure. Through the procedure summary we want to capture if the procedure enables the taint to be propagated at its output variables given information about whether its input variables are tainted. Therefore, we the summary of a procedure is the taintedness of the output variables of the procedure (those that escape the procedure scope such as returns, pointers, globals, etc.) represented as a function of the taintedness of its input variables.

The taint-propagation involves two passes through the call graph. The bottom up summarization phase visits each node of the call-graph in the reverse topological sort order and creates procedure summaries. Recursive procedures are treated context-insensitively. The top-down propagation phase visits the call graph in topological order, computing the actual taint at each variable, using procedure summaries to resolve procedure calls. The result of this analysis is the set of tainted variables $V_T(P)$ which can be queried during the next two stages to check if the variables are dependent on user request input.

This stage yielded 2795 variables in $V_T(P)$ for X server which is 38% of all variables in the portion of the X server program that we analyzed. Section 5.5 shows the size of $V_T(P)$ for the different programs analyzed.

4.3.2 Stage 2: Finding Security-Sensitive Objects

The goal of this stage is to identify the set of program variables that reference security-sensitive objects (i.e., the set of objects that need authorization). As we described in Section 4.2, security-sensitive objects are stored in containers. A *container* is an instance of a container data type or variable that holds multiple instances of the same type. Examples of containers include arrays, lists, queues, stacks, hash tables, etc. We claim that whenever the program stores a collection of objects in a container and the user has the ability to choose specific object(s) from this container, the chosen object becomes security-sensitive. Without additional manual input, it is impossible to determine whether all objects in a container are uniformly accessible to all users. We therefore make the conservative assumption that whenever user request input determines the choice of an object from a container, this object needs authorization by default. The variables to which such objects are assigned after retrieval thus represent the *security-sensitive objects* in the program.

The user can request an object from the container by specifying an identifier for the object. For example, X server has a global array (called `clientTable`) to store the set of resources (windows, cursors, fonts, etc.) belonging to all subjects. The user requests supply an identifier which is used to index into this array to retrieve the corresponding object.

We find containers may be arranged hierarchically, such that an object retrieved from a container may also have a field with another container. For example, in the following Listing 2 for the function DIXLOOKUPPROPERTY, the `Window` object `win` (which was previously retrieved from the global container `ClientTable`) has a field `win->opt->userProps` which is a list that stores properties associated with that window. The property `prop` retrieved from this container is needs to be authorized since user request provides the input parameter `pName` that specifies the property to be looked up.

In the identification of objects retrieved from containers, we focus on instances of programmer-defined data structures. Programmers typically define custom data structures to manage resources that are uniquely tied to the functionality of the application. For example, the X server has data structures for windows, devices, screens, fonts, cursors, etc.

Listing 2 Procedure to look up a specific property of a specific window in X server.

```c
/*** property.c ***/
int dixLookupProperty(PropertyPtr *p,
   Window * win, Atom pName,Client * c)
{
  PropertyPtr prop;
  for (prop=win->opt->userProps;
           prop; prop = prop->next)
      {
       if (prop->name == pName)
        break;
      }
  *p = prop;
}
```

Past efforts [72, 73, 37] have also focused on custom data structures.

Since containers are language-specific abstractions, we require a language-specific way of identifying the retrieval of objects from containers. We use the term *lookup function* to refer to any routine that retrieves objects from containers. Our approach depends on a language-dependent specification of lookup functions as shown in Figure 4.2. Since our analysis deals with C programs and the C language does not have standard lookup functions, we provide specifications in the form of code patterns. For example, in the above code snippet, the DIXLOOKUPPROPERTY function uses a standard code-level idiom in the C language, `next` pointer, to iterate through the list, comparing each item in the list with the identifier to determine if it satisfies the criteria. This code pattern is an example of a lookup function specification that the programmer needs to specify. Many object-oriented programming languages provide container classes which export well-defined methods to create a new container, insert, delete and provide access to objects in the container. In such cases, it is straightforward to identify lookup functions. We discuss the lookup function identification in more detail in Section 4.4.

Given such lookup functions, we can now formally define the set of all security-sensitive variables $\mathbf{V_S(P)}$ in the program.

Definition 2. *A variable $v \in V_S(P)$ if any following are true: a) If it is assigned a value* *from a container via a lookup function using a variable* $v' \in V_T(P)$. *b) If* $\mathcal{D}(v', v)$ *is true for some* $v' \in V_S(P)$. *c) If it is a global variable and in the set* $V_T(P)$.

First, any variable retrieved from a container using a tainted variable is security-sensitive. Second, any variable data-dependent on security-sensitive variables is also security-sensitive. Finally, any globals that can potentially be modified based on user request are also security-sensitive. This prevents trivial data flows between subjects using globals that may be modified based on user requests. If a global variable contains secret data that must be authorized before being read by a subject, these variables must be identified manually, as shown in Figure 4.2. We have found these variables to be rare in the programs that we have examined.

4.3.3 Stage 3: Finding User-Choice Operations

In this stage, our goal is to identify the set of operations that the user can choose to execute by modifying their user request input; we call these the *user-choice operations*. Control statements such as `if`, `switch` and function pointers are choice points in the program, where different program statements may be executed based on the value of the predicate evaluated in these control statements or the choice of function pointer values. If a tainted variable is used in a control statement's predicate or as a function pointer, then the subject can choose among different program functionality based on the values in the user request. Thus, the user request input provides the subject with a means to choose among sets of program statements to execute. These sets of program statements are the user-choice operations.

Consider the code snippet in Listing 3 from X server which shows the function CHANGEWINDOWPROPERTY. Here the user request inputs are `stuff`, `rc` and `mode` and statements predicated on these variables lead to user choice. First, `stuff` determines the choice of window and property being extracted. The lookup of an object from a container itself becomes the start of an operation that includes all statements in the program that can be executed following the lookup of the object. Next, the three control statements

Listing 3 The code snippet showing the procedure to change a specific property of a window of X server.

```
// ''stuff'' stores a formatted version
//  of the client request
int ChangeWindowProperty(ClientPtr *c,
    WindowPtr * w, int mode)
{
  WindowPtr * win;
  PropertyPtr * pProp;
  err = LookupWin(&win,stuff->window, c);
  rc =  LookupProperty(&pProp, win, stuff->property, c);
  if (rc == BadMatch)
    {/* Op 1*/
     pProp->name = property;
     pProp->format = format;
     pProp->data = data;
     pProp->size = len;
    }
  else
    { /* Op 2 */}
      if (mode == REPLACE)
       { /* Op 2.1 */
         pProp->data = data;
         pProp->size = len;
         pProp->format = format;
       }
     else if (mode == APPEND)
       {/* Op 2.2 */
         pProp->data = data;
         pProp->size += len;
       }
    }
}
```

that are predicated on `rc` and `mode` also cause user-choice operations. The first, `if (rc == BadMatch)` leads to two branches, `Op 1` and `Op 2` that represent two conceptually different operations, namely, adding a new property and changing an existing one. Furthermore, the latter operation has additional control statements `if (mode == REPLACE)` and `if (mode == APPEND)` through which the subject has the option of choosing between replacing an existing property (`Op 2.1`) and appending toit (`Op 2.2`).

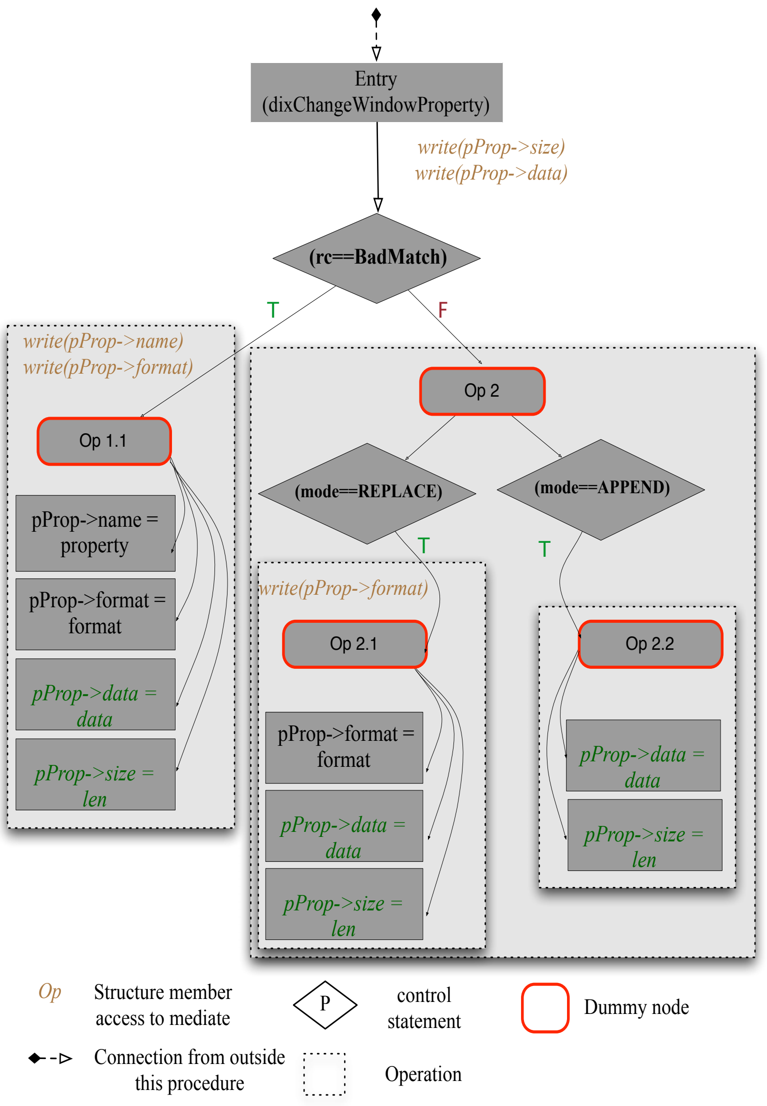

Figure 4.3: Shows the CDG with the user-choice operations for the DIXCHANGEWINDOWPROPERTY function

We use a standard program representation called the Control Dependence Graph [102] (CDG) to characterize operations. A CDG of a program $CDG(P) = (S(P), E)$ consists of a set of program statements $S(P)$ and the edges are represent the control dependence relationship between two statements. The control dependence property is typically defined in terms of control dominance relationships that exist between nodes in the Control Flow Graph (CFG) of the program. A statement Y is control-dependent on X if Y post-dominates a successor of X in the CFG but does not post dominate all successors of X. We make a minor adjustment to the traditional CDG representation by adding a dummy node for each choice (e.g., branch or possible function pointer value) of a control statement to group the program statements associated with each possible choice. We now define operations in terms of the CDG of the program.

Definition 3. *An* operation *is a subgraph of the CDG rooted at dummy node. If a dummy node's control statement is predicated on a variable in $V_T(P)$, then the operation rooted at the dummy node is a* user-choice operation.

Figure 4.3 shows the CDG constructed for the procedure DIXCHANGEWINDOWPROPERTY from listing 3. It also shows the four user-choice operations `Op 1.1`, `Op 2`, `Op2.1` and `Op 2.2` identified for the procedure DIXCHANGEWINDOWPROPERTY in listing 3. The statements in the diamond-shaped figures are the control statements, whose children are dummy nodes representing the start nodes for operations.

For the X server code the tool returned a total of 4760 user-choice operations. The results for the other programs are shown in Section 5.5.

4.3.4 Stage 4: Placing Authorization Hooks

Our final stage is the placement of authorization hooks. In order to do this, we first need to determine which of the user-choice operations computed in Stage 3 are actually security-sensitive operations. We then need to find placements of authorization hooks to mediate these operations. Our placement mechanism ensures complete mediation of all security-sensitive operations while ensuring that a user will never be authorized of any operations other than the ones requested.

We now have a the set of user-choice operations any of which can be requested by the user and the set of security-sensitive objects can of which can also be requested by the user. We now define security-sensitive operations as follows:

Definition 4. *A operation is* security-sensitive *if it is a user-choice operation and it allows the user to access a variable in* $V_S(P)$.

A security-sensitive operation must be mediated by at least one authorization hook. In order to place only necessary authorization hooks, we need to determine the authorization requirements of each security-sensitive operation. Such authorization requirements of an operation consist of the set of *security-sensitive accesses* performed by the operation's statements. Past efforts [73, 72, 37] have used structure member accesses of variables to represent the permissions required in programs. However, as accesses to variables directly may need to be protected, we generalize the definition slightly. In this work, each security-sensitive access consists of a variable that holds a security-sensitive object and the accesses made upon that variable (i.e., read or write).

We now have all the information we need to generate a placement of authorization hooks. A naive placement would simply associate an authorization hook with each security-sensitive operation. However, a naive placement would also be suboptimal. There are two problems. First, an authorization hook may be redundant if a dominating operation already authorizes all of the security-sensitive accesses performed in this operation. Second, an authorization hook may be hoisted if all the security-sensitive operations for the same control statement perform the same security-sensitive accesses. We must be careful to ensure that each authorization hook only authorizes security-sensitive accesses performed by the associated security-sensitive operation, which limits the amount of hoisting of authorization hook placement possible. Under this constraint, we discuss the two optimizations below. Each optimization relies heavily on the CDG of the program.

4.3.4.0.1 Hoisting common operations If all security-sensitive operations associated with the same control statement perform the same security-sensitive accesses, then it implies that irrespective of the choice the user makes, the security implication is the same. So we can hoist the authorization hook to operations that dominate those operations. This will enable us to place a single hook instead of a hook at each branch. Figure 4.3 shows how the structure member accesses `write(pProp->data)` and `write(pProp->size)` have been hoisted above control statement `rc==BadMatch` presumably at the operation that contains a call to the function DIXCHANGEWINDOWPROPERTY. In practice though, we restrict the hoisting of the structure member accesses to the entry point of the procedure if the procedure has more than one caller.

Algorithm 1 shows how we perform this optimization. For a node s_i, let $AS[s_i]$ represent the set of all security-sensitive accesses occurring in all operations dominated by s_i. For a control statement predicated on the user request, $AS[s_i]$ is the intersection of security-sensitive accesses occurring in all its successors. For all other statements, it is the union of accesses on their successors. We perform this optimization by processing the CDG bottom-up in reverse topological sort order, computing $AS[s_i]$ at each node s_i.

Algorithm 1 Bottom Up Operation Accumulation

```
$top' = TopoSortRev(G_d(P))$
**while** $top' \neq \emptyset$ **do**
    $s_i = top'.pop()$
    **if** $isControl(s_i, V_T(P))$ **then**
        $AS[s_i] = \bigcap_j AS[s_j] | (s_i, s_j) \in CDG(P)$
    **else**
        $AS[s_i] = \bigcup_j AS[s_j] \cup acc(s_j) | (s_i, s_j) \in CDG(P)$
    **end if**
**end while**
```

4.3.4.0.2 Avoiding redundant mediation Our second optimization is aimed at avoiding redundant authorizations thereby minimizing runtime performance impact of authorization hooks. We take advantage of the fact that it is sufficient to perform a specific authorization check only once along any control flow path. For a particular path $s_1 \rightarrow s_2 \rightarrow^* s_k$ in the CDG, the set of authorizations at s_k is the union of all authorizations on that path. However, there may be multiple distinct paths that reach s_k in the program. For example, a function may be called by two callers, one which has authorization hooks and one which does not. Let $AP[s_i]$ store the set of all security-sensitive accesses authorized along all paths leading to s_i in the CDG. Let $AT[s_i]$ be the set of security-sensitive accesses to be authorized at a node s_i. Algorithm 2 shows how we compute $AT[s_i]$. For this step, we process the CDG in the topological sort order.

In Figure 4.3 we can see that the security-sensitive accesses `write(pProp->data)` and `write(pProp->size)` that have already been authorized first control before the first control statement and therefore do not have to be mediated again at any operation that this statement dominates.

Algorithm 2 Control Flow Dominance

```
$top = TopoSort(G_d(P))$
**while** $top \neq \emptyset$ **do**
    $s_i = top.pop()$
    $AP[s_i] = AS[s_i] \cup \bigcup_j AP[s_j], (s_j, s_i) \in CDG(P)$
    $AT[s_i] = AS[s_i] - \bigcup_j AP[s_j], (s_j, s_i) \in CDG(P)$
**end while**
```

The set of hook placement locations is now defined as

$$A(P) = \{s_i | AT[s_i] \neq \emptyset\}$$

For X server we found that the there were 1382 placement locations before optimization and 532 after optimization. The total number of hook placement locations for the other test programs is shown in Section 5.5.

4.4 Implementation

We used the CIL framework to implement out tool. We now provide a brief overview of CIL.

4.4.1 The CIL Infrastructure

CIL is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs. CIL is a highly-structured, clean subset of C and features a reduced number of syntactic and conceptual form The main functionality of CIL is building an easy to use intermediate representation of the source code by performing transformations on the input file in order to obtain an AST(abstract syntax tree) which uses a few core constructs and a very clean semantics.

The main advantage of running CIL on a source file is that it organizes the imperative features of C into expressions, instructions and statements based on the presence or absence of side-effects and control-flow. Every statement will be annotated with successor and predecessor information and will consist of one or more instructions (without control- ow effects), thus providing a program representation that can be used with routines that require an AST (for instance, type based analyses), as well with routines that require a control flow graph (for instance, data flow analyses).

Our implementation consists of the following modules, all written in OCaml:

- A call graph constructor: It consists of 237 lines of code and uses a simple function pointer analysis. Any function whose signature matches that of a function pointer and whose address is taken is considered a potential callee. This analysis is conservative in the absence of typecasts.

- Static Taint tracker: It consists of 438 line of code. We currently do not use an alias analysis. We found that the alias analyses provided with the CIL distribution did not terminate within reasonable time period for some of our larger code bases such as the X server and `postgres`. Not having precise alias analysis may currently lead to false negatives in the results of static taint tracking. However, we found that our tool gives fairly accurate results even in the absence of alias analysis, presumably since alias analysis does not greatly affect user-request propagation. We hope to improve the precision of our tool using alias analysis as part of the future work. We also make conservative assumptions about library functions which are not defined inside the code we analyze. We assume that when we encounter such function calls, all actual call parameters affect the result. We also need to consider the case where sensitive-objects are passed as parameters to such function calls.

- Automatic hook placement: It consists of a control dependence graph generator and the code to identify security-sensitive operations and the bottom up and top down algorithms for placing authorization hooks, written in 1196 lines of code.

4.4.2 Building a Control Dependence Graph

One of the most useful tools exported by the CIL API is an implementation of the visitor pattern for CIL programs. The visiting engine scans depth-first the structure of a CIL program and at each node is queries a user-provided visitor structure whether it should visit the children, replace the subtree rooted at the current node with another subtree, rebuild the node after visiting all the children. Writing visitors is the way to customize the program traversal and transformation. Several analysis such as control flow graph graphs, liveness analysis, dominators, points-to Analysis, stack guard, partial evaluation, constant folding, reaching definitions, available expressions, liveness analysis, dead code elimination are provided with the current distribution.

We first build the intra-procedural control dependence graph for each procedure in the program after using the built in control flow graph in CIL and the visiting engine to compute postdominators. We then connect each procedure call site to the entry of the procedure. Note we do not do an context-sensitive inlined version of the control dependence graph since an operation is considered security-sensitive in all possible contexts. There can be multiple incoming edges to each procedure entry. But during the *bottom up operation accumulation* step of the hook placement, we do not propagate operations beyond the function entry point if it has multiple callers and in the *top down control flow dominance* step we do not propagate operations to an entry node unless it is performed at all the callers.

4.4.3 Input specifications

Our approach requires a manual specification of variables that represent user-requests and a representation of lookup functions. The procedure we followed to identify user requests was to identify instances of read-like library calls. We then manually reason about which subset of these calls truly represent client-inputs and annotate the variables that store the results of these calls as representing user requests. This typically takes less than 30 minutes of work by an experienced programmer since read-like library calls can be identified with automated tools. Identifying lookup functions for most object-oriented languages would be a straightforward language-level specification of standard container abstractions. Since the C language does not provide such standard abstractions, we provided specification at the level of common code patterns in C. We specify patterns for retrieval of objects from static and dynamic containers using indexing and pointer arithmetic and retrieval from recursive data structures by identifying instances where the next pointer is used inside loops to iterate through the container. In larger programs such as `postgres`, programmers typically implement their own containers. Therefore for postgres we created a program-level specification of lookup functions.

4.5 Evaluation

We tested the tool on two types of user-space server programs: (1) two programs with manually-placed authorization hooks, namely, X server and *postgres*, and (2) two programs without hooks, viz. *memcached* and *pennpush*. We evaluated our approach using two metrics: reduction in programmer effort and accuracy relative to manual placements.

The key results of the analysis are: (1) programmer effort was reduced by 80-90% for the tasks of identifying security-sensitive variables (objects) and data structures and operations requiring mediation when compared to manual placement and (2) the method placed hundreds of authorization hooks for X server and *postgres* with almost no false negatives (3 and 0, respectively) and few false positives (19 and 17, respectively), where the even these false positives were indicative of operations that could potentially be security-sensitive (except for one case due to analysis imprecision).

4.5.1 Reduction in Programmer Effort

We show how this tool aids the progammer by showing the reduction in the problem space that the different stages provide in Table 4.1. We show how the tool helps in highlighting variables, data structures, statements, and operations that are of significance in placing authorization hooks using the following metrics.

1. *Lines of code:* The total number of line of code *LOC* the programmer would have to examine to analyze the program.

2. *Variables:* Total number of variables in the program (*All*), the number of tainted variables(*Tainted*), identified in Stage 1, the number of security-sensitive variables (*Sensitive*) identified in Stage 2.

3. *Data structures:* The total number of data structures defined in the program (*All*), the number of data structures that correspond to security-sensitive objects identified (*Sensitive*).

4. *Control Statements:* The total number of control statements in the program (*All*), the number of control statements predicated on user-choice (*User-choice*).

5. *Operations:* The number of operations incident on all user-choice control statements (*User-choice*), the number of security-sensitive operations (*Sensitive*) and the number of operations where hooks are placed after optimization (*Hook*).

Table 4.1 shows these metrics for the programs that we evaluated. We can see that less than 50% of all variables are tainted by user-request, and less than 10% are security-sensitive which correspond to less than 20% of all data structures in the program. The number of security-sensitive operations is less than 30% of all user-choice operations and the number of operations where hook placement is suggested is typically around 11% of user-choice operations. Since our hook placement is conservative and fine-grained the programmer would only need to examine the final hook placement operations to determine

Source		Variables			Data Structures		Control Statements		Operations		
Program	LOC	All	Tainted	Sensitive	All	Sensitive	All	User-choice	User-choice	Sensitive	Hooks
X server	28658	7795	2975 (38%)	823 (10%)	404	61 (15%)	4297	3170 (73%)	4760	1382 (29%)	532 (11%)
postgres	49042	12350	5100 (41%)	402 (3%)	278	30 (10%)	5821	3289 (56%)	5063	1378 (27%)	579 (11%)
memcached	8947	2350	490 (20%)	82 (3%)	41	7 (17%)	982	647 (65%)	996	203 (20%)	56 (5%)
pennmush	78738	24372	4168 (17%)	1573 (6%)	311	38 (12%)	20202	4135 (20%)	6485	1382 (21%)	714 (11%)

Table 4.1: Table showing the reduction in programmer effort that the automated approach provides.

which ones to incorporate into the program. We therefore significantly reduce the amount of programmer effort needed to place authorization hooks.

Below, we examine the reduction in program effort for two programs that have no authorization hook placement.

4.5.1.0.3 Memcached This is a distributed memory object caching system for speeding up dynamic web applications. It is an in-memory key-value store of arbitrary chunks of data (strings, objects) from results of database calls, API calls, or page rendering. Our automated tool identifies seven security-sensitive data structures out of 41, each of which appeared to be critical to the correctness of the program. Two data structures covered the key-value pairs and statistics collected over their use. Another data structure was associated with a comment stating that modifying any variable of this type can result in undefined behavior. Also, security-sensitive global variables were identified that store program settings. By mediating access to variables of these data structures only, only 5% of the user-choice operations need to be examined by programmers.

4.5.1.0.4 pennmush *PennMush* is a server of textual virtual reality used for social and role-playing activities which maintains maintains a world database containing players, objects, rooms, exits, and programs. Clients can connect to the server and take on characters in the virtual world and interact with other players. Our tool found 38 security-sensitive data structures of the 311 data structures in the program. We were able to confirm that at least 9 data structures are security-sensitive. Among them are data structure representing the object stored in the database and its attributes, communication channels and locks in addition to the cache, mail messages exchanged between players, and the player's descriptor data structures. These 714 hooks reduce the programmer effort by nearly 90% relative to the user-choice operations in the program.

4.5.2 Accuracy Relative to Manual

For programs with hooks manually placed, we verify the accuracy of our approach by comparing manual and automated hook placements.

4.5.2.0.5 X Server There were 209 manually-placed hooks in the `dix` module that we analyzed. We found that our automated placements covered all but three of the manually placed hooks. Two of the false negatives were cases where the object being mediated by the manual hook was the client object which provides the user-request. The reason this is being mediated is because changing this parameter can have some implications on the global resource management done by the X server, but we do not consider resource management in authorization hook placement. The remaining case occurred because the manual hook was mediating reads of some global variables that are security-sensitive. As mentioned in Section 5.4, the user must identify global variables with secret values manually in an optional specification.

Our tool automatically chose 532 placements in X server, which while significantly greater than the 209 hooks manually placed largely corresponded to those placements. We found that the difference in number of hooks is due to domain-knowledge used by the programmers to aggregate hooks over operations that have same security semantics and the finer granularity that is possible using automated placement. Only 26 of the manually-placed hooks are dominated by a hook placed automatically. The remaining 183 manual hooks correspond to 487 automated hooks (the remainder are false positives, see below). One extreme case of this one-to-many mapping between manual and automated hook placements occurs in the procedure CopyGC in X server, which copies some information from a security-sensitive source object to a target security-sensitive object of type GC. User request specifies which field of the source object should be copied to the target object. This leads to a `switch` statement with 23 `case` statements where each case copies a different field of the source object. Therefore, we place 23 hooks, one at each case statement in contrast to the single manual hook placed to dominate the switch statement. However, it is not clear that the finer granularity of the automated placements are *wrong*, as the X server developers admit that the number of hooks will continue to increase. Thus, we believe that automated placement greatly reduced the effort and also keeps developers honest by showing them cases that may actually have different security implications.

Finally, we found that our tool produced 19 false positives. We found that the false positives were caused because of the following reasons:

1. 14 of the false positives deal with a data structure called emph_Node which stores property names. There is a central container called `nodetable` that stores the set of all property names. It is unclear if names of properties are themselves security-sensitive.

2. A false placement mediates an operation that reads the field *key→xkbInfo→desc*, which reads the description the keyboard that anyone can do. We found that manual hooks mediate other fields of the keyboard object such as *key→xkbInfo→state* and *key→down*.

3. Authorization hooks are placed for both the grab and ungrab operations on devices, whereas the manual placement covers only the grab. Ungrab is tantamount to releasing the device that was grabbed, therefore not really security-sensitive.

4. One placement was due to imprecision of function pointer analysis (InitClientResources) has same signature as the function pointer.

5. One placement was due to setting some global variables that were not considered security-sensitive by the manual analysis.

We find that all but one of these false placements was due to a lack of domain knowledge regarding the intent of the program. We believe that these other cases are reasonable for programmers to consider in designing an authorization hook placement.

4.5.2.0.6 postgres We analyzed the `tcop` module of Postgres that performs command dispatching from user requests. Our labeling of lookup operations is described in the Section 4.4. Postgres has 325 authorization hooks, including those provided by the SEPostgreSQL [21] project and with existing role-based access control hooks. Our tool identified 33 data structures corresponding to the security-sensitive operations, of which 22 were data types corresponding to catalog, tuples and relations. Our automated hook placement resulted in 579 hooks being placed for similar reasons to those discussed above for the X server. The placement generated no false negatives and 17 false positives. We found that several of these false positives were due to global variables that are not used globally, but we have not yet completed identification of causes of all the false positives.

Finally, we found that using container lookup alone to find security-sensitive objects does not identify all the security-sensitive operations to access containers, but a simple extension covers these cases as well. When an object is created and inserted into a container, mediation is often necessary to ensure that the object is assigned a security label and to verify that the creating subject is authorized to create objects assigned that label. However, no lookup is necessary to insert an object into a container, so this object (and the variable to which it is assigned) would not be marked security-sensitive using lookup. The extension that we propose is as follows. If a container holds any security-sensitive objects (i.e., any container in which a lookup is used to retrieve objects), then all objects in the container are security-sensitive. Thus, a variable upon which a security-sensitive container object is data-dependent is also a security-sensitive. This heuristic covers object creation as well as operations that modify all the objects in a container without retrieving one. In the programs we examined, all containers either were global variables or secondary containers within objects extracted from global containers. Therefore, all these write operations were marked as being to global variables, which are mediated in our approach. However, object creation for a local variable container would not be covered by our current approach, requiring this proposed extension.

Chapter 5

Generating the Least Mediation for Least Privilege

5.1 Introduction

Many programs manage resources on behalf of mutually distrusting clients, so they must have the ability to control the operations performed when processing client inputs. For example, the X server receives requests from multiple client programs to access resources (e.g., windows, cursors, etc.) managed by the X Window Server. Other programs that must control access to resource managed on behalf of external parties include databases, web servers, middleware, and browsers. To control access to security-sensitive operations when executing such requests, programmers place *authorization hooks* in their programs to mediate access to such operations[1]. Each authorization hook enables the program to decide whether to allow the operation or not, often by invoking a reference validation mechanism commonly called a *reference monitor* [11].

There are two main steps programmers perform when placing authorization hooks in programs. First, they must locate the program operations that are security-sensitive. In

[1]There are several projects specifically aimed at adding authorization hooks to legacy programs of these kinds [105, 106, 107, 20, 23, 22].

general, any program variable may contain security-sensitive data, so any instruction that operates on variables containing security-sensitive data may perform a security-sensitive operation. Given the number of variables in a program (e.g., 7,800 in the X server), identifying those that contain security-sensitive data and distinguishing those instructions that perform security-sensitive operations is a tedious and error-prone task. Second, our experience is that programmers want to minimize the effort of writing authorization policies. They often aim for a *minimal* placement of hooks that still satisfies the *complete mediation* [11] property, which states that every security-sensitive operation must be first checked by a reference validation mechanism. However, naive approaches (e.g., place a single authorization hook at the beginning of the program requesting all permissions) violate the principle of *least privilege* [108] by requiring that clients have all the request's permissions to process any specific request. At present, programmers make these trade-offs between complete mediation and least privilege manually, which delays the introduction of needed security measures [106] yet still requires many refinements after release. For example, authorization hooks were added to the X server mainline over a period of nearly four years [109], but we observed several subsequent modifications resulting in the addition of over 30 new hooks after upstreaming.

To lessen the burden on programmers, researchers have begun to examine techniques [71, 72, 73, 74, 110] to produce authorization hook placements automatically. However, these techniques currently only focus on the first step above, identifying security-sensitive operations in programs. The problem is that when methods do not aspire to minimality they either produce more hooks than necessary or require significantly more manual input from programmers. A recent approach proposes an automated authorization hook placement method that tracks the "choices" made by client request inputs, such as tainting of variables used in conditional predicates [110]. This method requires the least input of any known method (e.g., the identification of variable assignments from request input) and eliminates redundant authorization hooks, but still produces approximately twice the number of authorization hooks compared to manual placements produced by experts for the same programs.

After examination, we found the difference between automatic placements and expert manual placements is because experts optimize the placement of checks using domain knowledge. For example, an expert may treat two distinct security-sensitive operations as equivalent and insert one hook to mediate both operations, while an automatic placement system inserts one hook before each operation. Given this, we believe the only way to minimize the number of authorization hooks (while still satisfying complete mediation and least privilege) is to make the domain knowledge explicit. One approach is to ask programmers/experts write down a complete authorization policy, which states explicitly the set of operations allowed by a pair of subjects and objects. From the policy, a system can infer information such as what operations are equivalent in terms of access control to optimize hook placements. However, the difficulty is that writing down the complete authorization policy for a realistic program is a tedious task and programmers are often unwilling to spend the time.

We propose an approach that is based on *constraints on authorization policies*, which we call *authorization constraints*. In this case, the authorization policy is not provided explicitly. Instead, constraints narrow down the space of allowed authorization policies. When no constraints are provided, any policy is possible. When some constraints are provided, the underlying policy has to satisfy those constraints. In our system, we use two kinds of constraints. The first kind is *operation-equivalence* constraints, meaning two distinct security-sensitive operations on parallel code paths are authorized for the same set of subjects, which enables a single authorization hook to mediate both. The second kind is *operation-subsumption* constraints, meaning that if a subject can perform one operation then the same subject can perform the second operation. Therefore, if the second operation follows the first in a code path, then an authorization hook for the second operation is unnecessary.

For this approach to be automatic, we have to overcome two challenges. First, given a set of authorization constraints, a method needs to be developed to use these constraints to minimize the number of authorization hooks placed such that the resulting placement can enforce any policy that satisfies those constraints. Second, there should be a method to help programmers specify or discover authorization constraints. In this paper, we provide a solution to both of these challenges.

We address the first challenge by developing an algorithm that uses authorization constraints to eliminate hooks that are unnecessary. Specifically, we show that: (1) when operations on parallel code paths are *equivalent*, the hook placement can be *hoisted*, and (2) when operations that *subsume* each other also occur on the same control-flow path, we can *remove* the redundant hook. This method is shown to produce a minimal-sized hook placement automatically that satisfies the authorization constraints.

We have two insights to overcome the second challenge to help programmers generate the equivalence and subsumption constraints. First, given a hook placement that programmers experiment with, we can show a set of equivalence and subsumption constraints implied by that placement. This will make the authorization constraints explicit for the programmers and help them decide whether to keep their placement or modify the constraints. Second, we introduce the notion of *constraint selectors*. These selectors are able to make a set of constraint choices on behalf of the programmer based on higher-level goals. For example, suppose the programmer's goal is for her program to enforce multi-level security (MLS) policies, such as those expressed using the Bell-La Padula model [111]. The Bell-La Padula model only reasons about read-like and write-like operations, so any two security-sensitive operations that only perform reads (or writes) of the same object are equivalent. Thus, a constraint selector for MLS guides the method to create equivalence constraints automatically for such operations.

We have designed and implemented a source code analysis tool for producing authorization hook placements that reduce manual decision making while producing placements that are minimial for a given set of authorization constraints. The tool requires only the program source code and a set of security-sensitive operations associated with that code, which can be supplied by one of a variety of prior methods [71, 72, 73, 74, 110]. Using the tool, the programmer can choose a combination of constraint selectors, proposed hook placements, and/or hoisting/removal choices to build a set of authorization constraints. Once constraints are established, the tool computes a complete authorization hook placement that is minimal with respect to the constraints. We find that using our tool reduces programmer effort significantly. Importantly, this method removes many unnecessary hooks from fine-grained, default placements. For example, simply using the MLS constraint selector removes 124 hooks from the default fine-grained placement.

In this paper, we demonstrate the following contributions:

- We introduce an automated method to produce a minimal authorization hook placement that can enforce any access control policy that satisfies a set of authorization constraints.

- We explore the relationship between default placements, candidate placements, and the authorization constraints. Given a default placement and authorization constraints we are able to derive a *minimal* placement capable of enforcing those constraints. Given a default placement and a candidate placement we are able to extract the authorization constraints implied by the candidate placement.

- We simplify the task of eliciting authorization constraints from programmers, by providing them useful choices based on the program structure and by explaining constraints implied by sample placements.

- We evaluate a static source code analysis that implements the above methods on multiple programs, demonstrating that this reduces manual programmer effort by as much as 67% and produce placements that reduce the number of hooks by as much as 36% (reducing the gap between manual and automated placements by as much as 70%). We believe this is the first work that utilizes authorization constraints to reduce programmer effort to produce effective authorization hook placements in legacy code.

The paper is structured as follows: Section 5.2 lays out the motivation with concrete code examples and background information, Section 5.3 introduces the insight that guides our approach, Section 5.4 shows the design of our approach and how we envision helping programmers, Section 5.5 discusses our evaluation of the approach on four real-world programs.

5.2 Motivation

5.2.1 Background on Hook Placement

Authorization is the process of permitting a subject (e.g., user) to perform an operation (e.g., read or write) on an object (e.g., program variable or channel), which is necessary to control which subjects may perform security-sensitive operations. An authorization hook is a program statement that submits a query for the request (subject, object, operation) to an authorization mechanism, which evaluates whether this request is permitted (e.g., by a policy). The program statements guarded by the authorization hook may only be executed following an authorized request (control-flow dominance). Otherwise, the authorization hook will cause some remedial action to take place.

Related Work: The two main steps in the placement of authorization hooks are the identification of security-sensitive operations and the identification of locations in the code where authorization hooks must be placed in order to mediate such operations. Past efforts focused mainly on the first problem, defining techniques to identify security-sensitive operations [36, 70, 71, 72, 73, 37, 74, 75, 76, 110]. Initially, such techniques required programmers to specify code patterns and/or security-sensitive data structures manually, which are complex and error-prone tasks. However, over time the amount of information that programmers must specify has been reduced. In recent work, researchers infer security-sensitive operations only using the sources of untrusted inputs and language-specific lookup functions [110].

When it comes to the placement of authorization hooks, prior efforts typically suggest

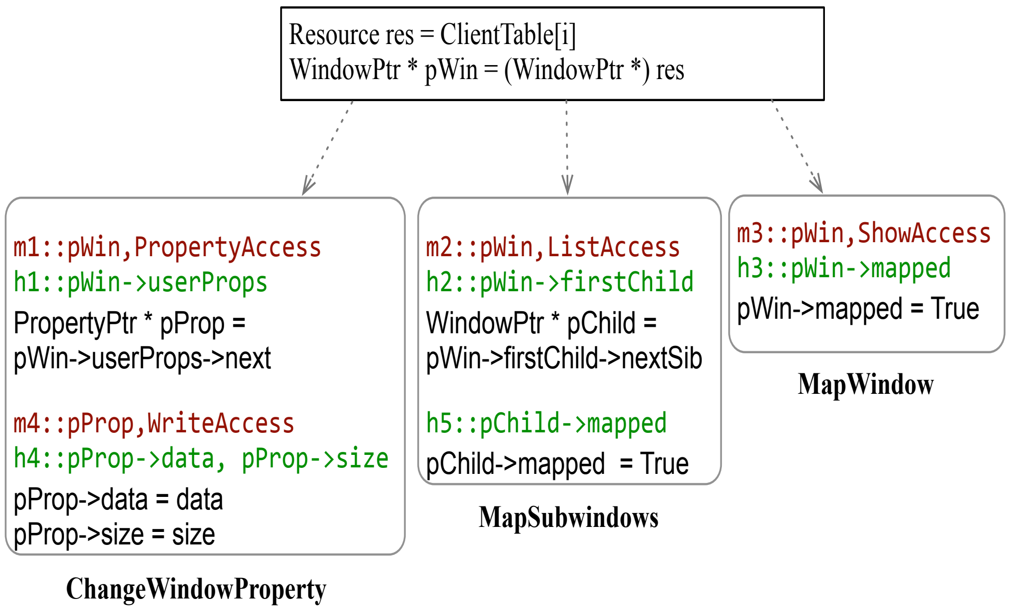

Figure 5.1: Hook Placement for functions MAPWINDOW, MAPSUBWINDOWS, and CHANGEWINDOWPROPERTY

placing a hook before every security-sensitive operation in order to ensure complete mediation. There are two problems with this simple approach. First, automated techniques often use low-level representations of security-sensitive operations, such as individual structure member accesses, which might result in many hooks scattered throughout the program. More authorization hooks mean more work for programmers in maintaining authorization hooks and updating them when the policy changes. Second, such placements might lead to redundant authorization, as one hook may already perform the same authorization as another hook that it dominates. Recently, researchers have suggested techniques to remove hooks that authorize structure member accesses redundantly [110]. This approach still does not result in a placement that has a one-to-one correspondence with hooks placed manually by domain experts. In X server, it was found that while the experts had placed 200 hooks, the automated technique suggested 500 hooks. In the following subsections we discuss some reasons for this discrepancy.

5.2.2 How Manual Placements Differ

We find that there are typically two kinds of optimizations that domain experts perform during hook placement. We follow with examples of both cases from the X Server.

- First, assume there are two automatically placed hooks D_1 and D_2 such that the former dominates the latter in the program's control flow. The placement by the domain expert has a matching hook for D_1 but not for D_2. We can interpret this as the expert having *removed* (or otherwise omitted) a finer-grained hook because the access check performed by D_1 makes D_2 redundant.

- The automated tools place hooks $(D_1, ..., D_n)$ at the branches of a control statement. The domain expert has not placed hooks that map to any of these hooks, but instead, has placed a hook E_1 that dominates all of these hooks[2]. The expert has *hoisted* the mediation of operations at the branches to a common mediation point above the control statement.

First, examine the code snippet in Figure 5.1. In the figure, hooks placed by the programmer have prefixes such as `m1::` and hooks placed by our previous automated tool [110] have prefixes such as `h1::`. The function MAPWINDOW performs the operation `write(pWin→mapped)` on the window, which makes the window viewable. We see that the programmer has placed a hook `m3::(pWin,ShowAccess)` that specifically authorizes a subject to perform this operation on object represented by `pWin`. Access mode `ShowAccess` identifies the operation. The requirement of consistency in hook placement dictates that an instance of hook `ShowAccess` should precede any instance of writing to the `mapped` field of a Window object that is security-sensitive. MAPSUBWINDOWS performs the same operation on the child windows `pChild` of a window `pWin`. While our previous automated tool prescribes a hook at MAPSUBWINDOWS, we find that the domain expert has chosen not to place a corresponding hook. MAPSUBWINDOWS is preceded by the manual hook `m2::(pWin,ListAccess)` for the subject to be able to

[2]There is no hook in the automated placement that matches E_1.

list the child windows of window `pWin`, but there is no hook to authorize the operation `write(pChild->mapped)`.

Second, look at the example shown in Listing 4. The function COPYGC in X server accepts a source and target object of type `GC` and a `mask` that is determined by the user request and, depending on the mask, one of 23 different fields of the source are copied to the target via a `switch-case` control statement. Since each branch results in a different set of structure member accesses, our previous tool considers each branch to perform a different security-sensitive operation. Therefore, it suggests placing a different hook at each of the branches. On the contrary, there is a single manually placed hook that dominates the control statement, which checks if the client has the permission `DixGetAttrAccess` on the source object. Therefore one manually placed hook replaces 23 automated hooks in this example.

5.2.3 Balancing Minimality with Least Privilege

We have in X Server a mature codebase, which has been examined over several years by programmers in order to reach a consensus on hook placement. We are convinced that a deliberate choice being made by the experts about where to place hooks and which hooks to avoid on a case-by-case basis. For example, in CHANGEWINDOWPROPERTY in Figure 5.1, a property `pProp` of `pWin` is retrieved and accessed. Programmers have placed a finer-grained hook `m4::(pProp,WriteAccess)` in addition to the hook `m1::(pWin,SetPropertyAccess)`. Constrast this with the MAPSUBWINDOWS example where they decided not to mediate the access of a child object.

The fundamental difference between a manually written hook placement and an automatically generated one is in the granularity at which security-sensitive operations are defined. When the automated tool chooses to place a hook at each of the branches of a control statement, it implicitly identifies security-sensitive operations at a finer granularity than experts. The choice of granularity of security-sensitive operations is an exercise in balancing the number of hooks placed and least privilege. A fine-grained placement allows

Listing 4 Example of manual hoisting in the COPYGC function in the X server.

```
/*** gc.c ***/
int CopyGC(GC *pgcSrc, GC *pgcDst, BITS32 mask){
 switch (index2)
  {
      result = dixLookupGC(&pGC, stuff->srcGC,
                           client,DixGetAttrAccess);
      if (result != Success)
         return BadMatch;
      case GCFunction:
         /* Hook(pgcSrc, [read(GC->alu)]) */
         pgcDst->alu = pgcSrc->alu;
         break;
      case GCPlaneMask:
         /* Hook(pgcSrc, [read(GC->planemask)]) */
         pgcDst->planemask = pgcSrc->planemask;
         break;
      case GCForeground:
         /*  Hook(pgcSrc, [read(GC->fgPixel)]) */
         pgcDst->fgPixel = pgcSrc->fgPixel;
         break;
      case GCBackground:
         /* Hook(pgcSrc, [read(GC->bgPixel)]) */
         pgcDst->bgPixel = pgcSrc->bgPixel;
         break;
        /* .... More similar cases */
     }
}
```

more precision in controlling what a subject can do, but this granularity may be overkill if programmers decide that subjects are authorized to access operations in an all-or-nothing fashion. For the switch statement with 23 branches, having 23 separate hooks will lead to a cumbersome policy because the policy will have 23 separate entries for each subject. Since the programmers decided that all subjects either can perform all 23 operations for an object or none, it is preferable to have a single hook to mediate the 23 branches.

We have also seen that even with manual hook placement multiple iterations may be necessary to settle on the granularity that balances least privilege and minimality in hook placement. For example, the X server version of 2007 had only four operation modes, namely, `read`, `write`, `create` and `destroy`. But during the subsequent release, the programmers replaced these with 28 access modes that was necessary to specify and enforce policies with finer granularity. Since the first release of X server with the XACE hooks in 2007, the hooks have undergone several changes. Over 30 hooks were added to the X server code base, but some existing hooks were also removed, moved or combined with other hooks [112]. Some of these changes are documented in the XACE specification [88]. We believe that observing typical policy specifications at runtime enabled the programmers to add and remove hooks in subsequent versions of the application. We want to understand iterative refinement and build methods to automate some tasks in the process, making key decisions explicit.

5.3 Authorization Hook Placement Problem

In this section, we discuss the two main steps in authorization hook placement to satisfy least privilege: a) finding security-sensitive operations (SSOs) in the program and, b) consolidating hook placements.

5.3.1 Identifying security-sensitive operations

In this section, we provide some background on prior research [110] in automatically identifying the set O of security-sensitive operations (SSOs) in programs using static analysis. Each SSO is represented using a variable v and a set of read and write structure member accesses on the variable. There may be multiple instances of an SSO in a program. Each instance is represented using the tuple (o, l) where o is the SSO and l is the location (statement) in the code where the instance occurs. Let O_L be the set of all instances of SSOs in the program. Our goal is to place authorization hooks to mediate all the elements of O_L.

Definition 5. *An authorization hook is a tuple* (O_h, l_h) *where* l *is a statement that contains the hook and* $O_h \subseteq O$ *is a set of security-sensitive operations mediated by the hook.*

A set of authorization hooks is called an *Authorization Hook Placement*. Our approach in [110] used a Control Dependence Graph (CDG) of the program as part of the analysis. A CDG of a program $CDG = (L, E)$ consists of a set of program statements L and edges E, which represent the control dependence relationship between two statements; this exposes the statements that a given statement depends upon for execution. We add an explicit dummy node to represent each branch of a control statement since security-sensitive operations are identified at the granularity of control-statement branches. We will continue to use the CDG in this work for refining hook placements.

5.3.2 Consolidating Hook Placements

We note from our experience that there are three scenarios that result from how permissions are typically assigned to subjects to perform security-sensitive operations. Let U be the set of all subjects for a hypothetical access control policy $\mathcal{M}$ for the given program. Let $Allowed$ be a function that maps each security-sensitive operation in O to subjects U that are allowed to perform the operation according to policy $\mathcal{M}$. We identify three cases that are relevant to the placement of authorization hooks:

- **Invariant I**: First, if it is true that $Allowed(o) = U$ for some $o \in O$ (i.e., every subject is assigned the permission to perform o), it is not necessary to have an authorization hook to mediate o. In this case, the hook for o can be removed.
- **Invariant II**: Second, given any two operations o_1 and o_2, if permissions are always assigned in such a way that $Allowed(o_1) = Allowed(o_2)$ then o_1 and o_2 are equivalent for the purpose of authorization. This means that any hook that mediates o_1 and dominates o_2 in the code automatically authorizes o_2 and vice versa.
- **Invariant III**: Finally, given two operations o_1 and o_2, if permissions are assigned in such a way that $Allowed(o_1) \subset Allowed(o_2)$ then operation o_1 'subsumes' o_2 for the purpose of authorization. This means that a hook that mediates o_1 and dominates o_2 can also mediate o_2 but not vice-versa.

These observations lead us to believe that in order to enable hook placement optimizations we need to impose equivalence and partial-order relationships between the elements in O. Therefore, we define a *set of authorization constraints* as follows:

Definition 12. *A* set of authorization constraints $\mathcal{P}$ *is a pair* (S, Q) *of relationships between SSOs in the program, where* Q *stands for* **equivalence** *and* S *stands for* **subsumption**.

We can see that the equivalence relationship Q results in a partitioning of the set O of security-sensitive operations. Let O_Q be the set of partitions produced by Q. The subsumption relationship S imposes a partial order between the elements in O_Q. We can use these two relationships to perform hoisting and removal operations. We describe the technique for this in the following section.

There are two questions related to authorization constraints. First, how to get those constraints. We will present dynamic and static analysis that help programmers generate those constraints (semi-)automatically from source code in later sections.

The second question is how to use authorization constraints to consolidate hook placements. Our system uses an algorithm that, given a program and its set of authorization constraints, generates a minimal authorization hook placement that satisfies complete mediation and least privilege. **Complete mediation** states that every SSO instance in a program should be dominated by an authorization hook that mediates access to it. A placement that enforces **least privilege** ensures that during an execution of a program, a user of the program (subject) is only granted the set of permissions to perform the SSOs requested as part of that execution. This effectively puts a constraint on how high a hook can be hoisted. We define the two properties more formally in Appendix 5.4.3 where we show that our approach does indeed result in a placement that satisfies these properties.

We observe that even though automated placement methods may be capable of producing placements that can enforce any policy, and thus can enforce least privilege, programmers will not accept a hook placement unless there is a justified need for that hook. Specifically, programmers only want a hook placed if that hook will actually be necessary to prevent at least one subject from accessing a security-sensitive operation. That is, while

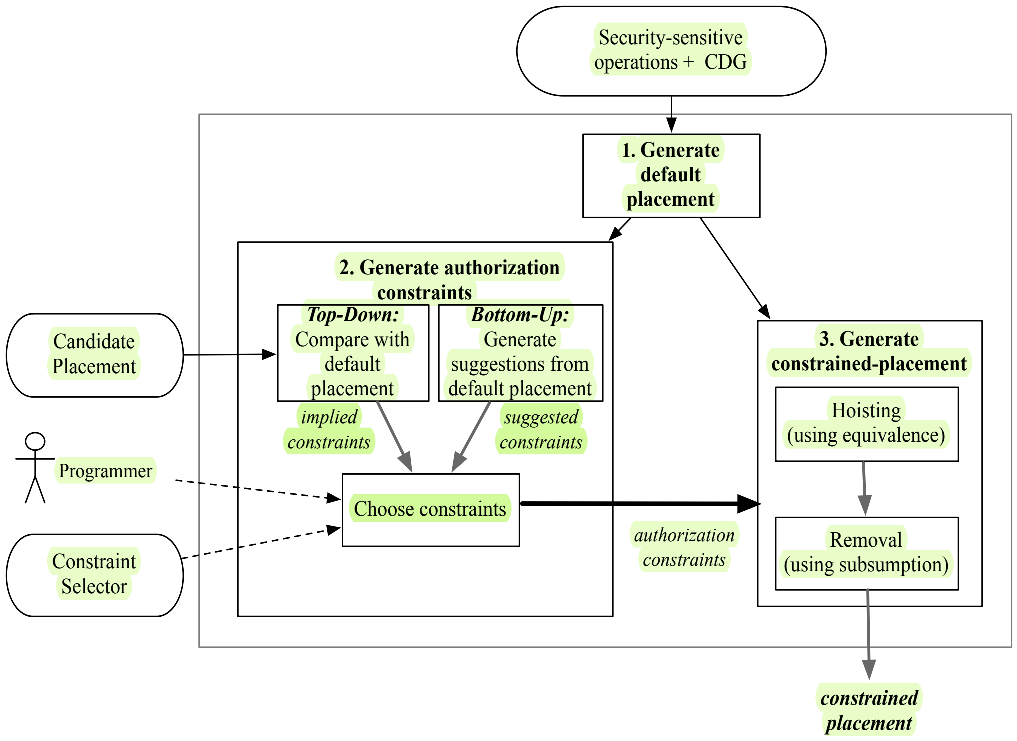
Security-sensitive operations + CDG
1. Generate default placement
2. Generate authorization constraints
Top-Down: Compare with default placement
Bottom-Up: Generate suggestions from default placement
implied constraints
suggested constraints
Candidate Placement
Programmer
Constraint Selector
Choose constraints
authorization constraints
3. Generate constrained-placement
Hoisting (using equivalence)
Removal (using subsumption)
constrained placement

Figure 5.2: Authorization hook placement method

programmers agree that they should give subjects the minimal rights, they also require *that a program should have only the minimal number of hooks necessary to enforce complete mediation and least privilege*.

5.4 Design

We propose the following solution, shown in Figure 5.2. There are two main inputs to our approach: a) the set O of SSOs and their instances O_L in the program, and b) the control dependence graph CDG of the program. We have defined these inputs in Section 5.3.1 and discussed prior work where we use static analysis to infer them automatically.

Step 1: Generating a default placement: First, using O, O_L and CDG, we generate a default placement. Since the elements in O_L are in terms of code locations, and the nodes in the CDG have location information, we can create a map $C2O$ from each node n in the

CDG to the set of SSO instances in O_L that occur in the same location as n. The default placement $\mathcal{D}$ has a hook at each node n where $C2O[n] \neq \emptyset$. Once the default placement is generated, there are two orthogonal steps in our approach as shown in Figure 5.2.

Step 2: Generating authorization constraints: If a set of authorization constraints is not available, programmers are guided in step 2 to generate one. The task of building and modifying a set of authorization constraints $\mathcal{P}_i$ is orthogonal to generating hook placements. Using external static and dynamic analyses we can help the programmer specify $\mathcal{P}_i$. We help programmers generate constraints either by giving suggestions for constraints (*bottom-up approach*) or by explaining the implied constraints of a candidate placement (*top-down approach*). Finally, we introduce the notion of constraint selectors to make the choices of authorization constraints on behalf of programmers. This step is discussed in Section 5.4.2.

Step 3: Generating constrained hook placement: If a set of authorization constraints $\mathcal{P}_i$ is already available, step 3 of our tool is able to automatically generate a minimal placement[3] $\mathcal{E}_i$ that can enforce any policy that satisfies the constraints. Our approach uses the *equivalence* constraints to perform *hoisting* and *subsumption* constraints to perform *removal*, thereby minimizing the number of hooks. This procedure is described in Section 5.4.1.

5.4.1 Deriving a constrained placement

Given default placement $\mathcal{D}$ and set of authorization constraints $\mathcal{P}$, our system can derive a candidate constrained placement $\mathcal{E}$ that satisfies complete mediation and least privilege enforcement with the minimum number of hooks (used in step 3 of our design). The subsumption S and equivalence Q relationships in $\mathcal{P}$ enable us to perform two different hook refinements on $\mathcal{D}$ to derive the $\mathcal{E}$. We present the algorithm next and the proof why it has

[3]Henceforth referred to as constrained placement.

Algorithm 3 Algorithm for hoisting

```
$top' = TopoSortRev(CDG)$
**while** $top' \neq \emptyset$ **do**
    $s_i = top'.pop()$
    **if** $isControl(s_i)$ **then**
        $\alpha[s_i] = C2O[s_i] \cup \bigcap_j^Q \{\alpha[s_j] \mid (s_i, s_j) \in CDG\}$
    **else**
        $\alpha[s_i] = C2O[s_i] \cup \bigcup_j \{\alpha[s_j] \mid (s_i, s_j) \in CDG\}$
    **end if**
**end while**
```

desired properties is in Appendix 5.4.3.

5.4.1.1 Hoisting

The first refinement is called *hoisting* and it aims to consolidate the hooks for mediation of equivalent operation instances that are siblings (appear on all branches of control statements). This lifts hook placements higher up in the CDG based on Q. Given a node n in the CDG, if each path originating from that node n contains SSOs that are in the same equivalence class in O_Q, then we can replace the hooks at each of these paths with a single hook at n. This relates to the example in Listing 4, where if the operations along all the 23 branches of the $Switch$ statement in line 3 are equivalent, then we can replace the 23 automatically generated hooks at those branches with a single hook that dominates all of them.

Algorithm 3 shows how hoisting is done. It uses the CDG and the $C2O$ map as inputs. Accumulator α gathers the set of SSOs at each node n by combining the $C2O$ of n with the α mapping from the child nodes. The algorithm traverses the CDG in reverse topological sort order and makes hoisting decisions at each node in the CDG. We partition the set of nodes in the CDG into two types - control and non-control nodes. Control nodes represent control statements (such as if, switch etc) where hoisting can be performed. At control nodes[4], we perform the intersection operation $\bigcap^Q$ which uses the equivalence relation Q to perform set intersection in order to consolidate equivalent SSOs. Note that this intersection

[4]Each control node has dummy nodes as children each representing a branch of the control node

Algorithm 4 Algorithm for removal

```
$top = TopoSort(CDG)$
while $top \neq \emptyset$ do
    $s_i = top.pop()$
    $O_D = \bigcap_j^{QS} \{\phi[s_j] \mid (s_j, s_i) \in CDG\}$
    $\phi[s_i] = \alpha[s_i] \cup O_D$
    $O_R = \emptyset$
    for all $o_m \in \alpha[s_i]$ do
        if $\exists o_n \in O_D, (o_n\, S\, o_m)$ or $(o_n\, Q\, o_m)$ then
            $O_R = O_R \cup \{o_m\}$
        end if
    end for
    $\beta[s_i] = \alpha[s_i] - O_R$
end while
```

operation limits how high hooks may be hoisted in the program. At non-control nodes, we accumulate SSOs from children using a union operation.

Note that this algorithm does not remove any hooks. It places new hooks that dominate the control statements where hoisting occurs. For example, given Listing 4, the algorithm would place a new hook before the Switch statement. The removal operation which we discuss next will eliminate the 23 hooks along the different branches because of the new hook that was placed by this algorithm.

5.4.1.2 Removal

The second refinement is called redundancy removal and aims to eliminate superfluous hooks from CDG using S. Whenever a statement s_1 that performs SSO o_1 dominates statement s_2 that performs SSO o_2 and o_1 either subsumes or is equivalent to o_2 according to $\mathcal{P}$, then a hook at s_1 for o_1 automatically checks permissions necessary to permit o_2 at s_2. Therefore, we may safely remove the hook at s_2 without violating complete mediation.

In the example in Figure 5.1, if we had authorization constraints specify that operation $read(pWin \rightarrow firstChild)$ subsumes (or is equivalent to) operation $write(pChild \rightarrow mapped)$, then we do not need the suggested hook `h5`.

Algorithm 4 shows how the removal operation is performed. The algorithm takes as input the CDG and the map α computed by the hoisting phase. It traverses the CDG in topological sort order (top-down) and at each node n makes a removal decision based on the set of operations checked by all hooks that dominate n. The accumulator ϕ stores for each node n the set of operations checked at n and all nodes that dominate n. While processing each node, the algorithm computes O_D, which is the set of operations checked at dominators to node n. Note that because of the way we construct the CDG interprocedurally (refer to Section 5.3.1), a node can have multiple parents only at function calls. The $\bigcap^{QS}$ that combines authorized operations in case of multiple parents is shown in Algorithm 5.

Algorithm 5 Algorithm for performing $\bigcap^{QS}$ on two sets O_i and O_j. Returns result in O_T.

```
$O_T = \emptyset$
**for all** $o_i \in O_i$ **do**
    **for all** $o_j \in O_j$ **do**
        **if** $o_i \ Q \ o_j$ **then**
            $O_T = O_T \cup \{o_i\}$
        **end if**
        **if** $o_i \ S \ o_j$ **then**
            $O_T = O_T \cup \{o_j\}$
        **end if**
        **if** $o_j \ S \ o_i$ **then**
            $O_T = O_T \cup \{o_i\}$
        **end if**
    **end for**
**end for**
```

Next, Algorithm 4 creates the set O_R which is the set of operations that do not have to be mediated at n since they are either subsumed by or equivalent to operations that have been mediated at dominators to n. The resulting map β from nodes to the set of operations that need to be mediated at the node gives the final placement. The constrained placement $\mathcal{E}$ suggests a hook at each node n such that $\beta[n] \neq \emptyset$.

Note that both the bottom-up hoisting and top-down removal must be performed in sequence to get the final mapping from nodes to the set of SSOs that need mediation.

5.4.2 Deriving a set of authorization constraints

In this section we describe the computation necessary to produce the *suggested constraints* and *implied constraints*.

Once the default hooks are generated, there are three ways in which we can help programmers generate authorization constraints.

- *Making constraint suggestions visible:* First, whenever our automated approach produces a placement, it also produces the next set of choices for hoisting and removal of hooks, giving programmers constraint suggestions that are useful from the point of placing authorization hooks. This is described in Section 5.4.2.1.

- *Making implied constraints visible:* Second, given a candidate placement, we can produce a set of authorization constraints implied by that placement by comparing the candidate placement against the default placement or any derivative of the default placement (called the 'baseline' placement). The implied constraints capture the difference between the two placements (i.e., constraints that when applied to the baseline placement would produce the candidate placement). This is described in Section 5.4.2.2.

- *Automating constraint choice:* The programmers could pick from the implied and suggested constraints manually, but we go one step further to provide constraint selectors that automatically make constraint choices on behalf of programmers based on authorization goals. This is described in Section 5.4.2.3.

After our approach generates the default placement, the programmer has two options.

- **Bottom-Up:** The bottom-up approach helps programmers choose from a set of *suggested constraints*. First, along with the *default placement*, our approach also provides the programmer with a set of *suggested constraints* in the form of hoisting and removal choices. The programmer can then apply specific constraint selectors that will make these choices on their behalf in order to produce constrained placements automatically. The new placement may generate additional suggestions that programmers can either address manually or by using other constraint selectors to deal them automatically, making this an iterative process.

- **Top-down:** Whenever programmers have a candidate placement in mind, the top-down approach make the implied constraint choices they have made visible to them. This will give them a way to explain their placement choices. The programmer may guess at a candidate placement (based on their limited understanding of the program). Then they may use constraint selectors as a way to reason about this candidate placement. The will apply constraint selectors to automatically generate a 'baseline' placement. They can then use our approach to compare their candidate placement with the 'baseline' placement. This will then generate a set of *implied constraints* that the programmers can use to reason about their placement.

5.4.2.1 Deriving suggested constraints bottom-up

We note that any set of authorization constraints is useful only when it allows us to perform hoisting or removal. So whenever our approach produces a placement automatically, we generate a set of suggestions for additional authorization constraints expressed as the next set of *hoisting* and *removal* choices available.

- Hooks that correspond to branches of the same control statement is reduced to one *hoisting choice* the programmer has to make. For the `switch-case` example in Figure 4, our 23 hooks along the branches would produce one hoisting choice for the programmer. This is the same as suggesting that the operations performed on all 23 branches are equivalent.

- Second, any hook may be a candidate for removal if it has a dominating hook. We show the hook and its dominating hook as one *removal choice*. For example, in

Figure 5.1 hook `h5` has a dominating hook `h2` and so we will show this as a removal choice.

5.4.2.2 Deriving implied constraints top-down

We claim that given a candidate constrained placement $\mathcal{E}$ we can infer the *authorization constraints* implied by it. The key to solving this problem is determining a) which hooks in $\mathcal{D}$ have corresponding hooks in $\mathcal{E}$; we call these *matched hooks* and create a mapping $D2E$ from hooks in D to hooks in E; and b) for the remaining unmatched hooks in $\mathcal{D}$, identify the *matched hooks* in D that immediately dominate them. For example, in Figure 5.1 hook `h2` corresponds to the constrained hook `m2` and is therefore a *matched hook*. Hook `h5` does not have a corresponding constrained hook; so we can map it to `h2` which is the default matched hook that immediately dominates it. We create a mapping $D2D$ to represent this. Then based on these mappings we can analyze the program and gather the subsumption and equivalence constraints implied by the constrained placement.

We begin by first mapping hooks in D and E to the nodes in the CDG based on their location. We generate maps called $D2G$ and $E2G$ for this.

- $D2E$: A hook $h_d \in \mathcal{D}$ is mapped to a hook $h_e \in \mathcal{E}$ in $D2E$ only if a) h_e and h_d mediate operations on the same or related variables, b) given $n_e = E2G(h_e)$ and $n_d = D2G(h_d)$ where there exists a path in CDG between n_e and n_d, and c) there exists no other $h'_e \in \mathcal{E}$ or $h'_d \in \mathcal{D}$ mapped to any of the nodes along the path between n_e and n_d in the CDG.

- $D2D$: A hook h_{d1} is mapped to a hook h_{d2} in $D2D$ if given $n_1 = D2G(h_{d1})$ and $n_2 = D2G(h_{d2})$, a) n_1 dominates n_2 in CDG, and b) h_{d1} has a mapping in $D2E$ and c) there exists no other hook h_{d3} mapped to any of the nodes in the path from n_1 to n_2 such that h_{d3} has a mapping in $D2E$.

Once we have the two hook maps, we use the following intuition for identifying the Q and S implied by the placement $\mathcal{E}$.

- Whenever multiple matched hooks in $\mathcal{D}$ have the same mapping in $D2E$ then it signifies the possibility of hook equivalence. Let $HD = \{h_{d1}, h_{d2}, ..., h_{dn}\}$ be the set of hooks that are all mapped to the same hook h_e in $D2E$. This means that there was no dominance relationship between any of the elements in HD; so they would have to appear along different paths originating from h_e. Since the operations mediated by all the hooks in HD are instead being mediated by a single hook $h_e \in \mathcal{E}$, this indicates that these operations belong to the same equivalence class in O_Q. For example, in Listing 4, the hooks suggested in $\mathcal{D}$ for each of the 23 branches would all be mapped to the same manually placed hook in line 5 and so the operations along those branches would all be made equivalent.

- For each hook h_d that has a mapping in $D2E$ we identify the set $HD = \{h_{d1}, h_{d2}, ..., h_{dn}\}$ of hooks that are mapped to it in $D2D$. This implies that the choice was made not to place expert hooks at any of the nodes in HD which leads us to infer that the operation mediated by h_d 'subsumes' the operations mediated by each of the hooks in HD and we can add this to S. For example in Figure 5.1, `h2` is mapped to `m2` in $D2E$ and to `h5` in $D2D$. Therefore, the operation $read(pWin \rightarrow firstChild)$ mediated by `h2` subsumes operation $write(pChild \rightarrow mapped)$ that is mediated by `h5`.

5.4.2.3 Using constraint selectors to make choices

Given a set of constraint suggestions $\mathcal{P}_\alpha = (S_\alpha, Q_\alpha)$ based on the set of hoisting and removal opportunities in the program, our framework enables the use of *constraint selectors*, where each constraint selector f extracts a corresponding set of authorization constraints $\mathcal{P}_f$ from these relations. A constraint selector f essentially picks a subset of S_α and Q_α based on some authorization goals. We envision that a constraint selector would have some simple rules to decide which subset of the candidates in $\mathcal{P}_\alpha$ should be picked in order to create a set of authorization constraints $\mathcal{P}_f$ that is then used to generate a corresponding constrained placement.

This framework allows us to explore different types of constraint selectors. First, we may aim to enforce well-known security policies, such as Multi-Level Security [113]. In MLS, subjects are assigned permissions at the granularity of read and write accesses. Typically, a subject that is permitted to read a field of a variable is permitted to read all fields of that variable; a similar case holds for writes. This means that all read-like (write-like) accesses of a variable can be treated as equivalent. To derive a set of authorization constraints in this case, we apply a constraint selector that checks all equivalence suggestions in Q_α to determine if the corresponding operations are equivalent (i.e., reads of the same variable/writes of the same variable). If yes, then we add an equivalence relationship to the set of authorization constraints. In Section 5.5, we will explore some candidate constraint selectors to investigate how authorization constraints can be generated automatically.

5.4.3 Hook Placement Properties

First, we want to prove that the authorization hook placement mechanism satisfies two goals: *least privilege* enforcement and *complete mediation*. Least Privilege requires that at any point in the execution of a program a subject must only be authorized for operations that are requested as part of that execution.

Definition 6. *In a least privilege placement a hook* (O_h, l_h) *placed at location* l_h *authorizing a set of SSOs* O_h *implies that for each* $o_i \in O_h$*, on each path in the program originating from* l_h*, there must be an operation instance* (o_j, l_j) *such that* $(o_i \, S \, o_j) \vee (o_i \, Q \, o_j)$ *.*

To show that our placement guarantees the enforcement of least privilege, we need to show that our initial placement satisfies this property and the subsequent hoisting and removal phases preserve this property.*Complete mediation* stipulates that every SSO instance must be control flow dominated by an authorization hook that authorizes that operation.

Definition 7. *Complete Mediation requires that for every operation instance* (o_i, l_i)*, there exists a hook* (O_h, l_h) *such that* l_h *control flow dominates* l_i *and there exists* $o_h \in O_h$ *such that* $(o_i \, Q \, o_h) \vee (o_h \, S \, o_i)$*.*

To show that our placement guarantees the enforcement of complete mediation, we need to show that our initial placement satisfies this property and the subsequent hoisting and removal phases preserve this property. Our approach depends on two inputs a) The set of all SSOs in the program b) The authorization constraints that determine all possible optimizations (hoisting and removal) in hook placement. Our proof assumes that both of these specifications are complete.

Our approach starts by placing a hook at every instance of every SSO in the program. First, it is trivial to show that this results in a placement that adheres to complete mediation since every SSO instance has a corresponding hook. Second, this approach guarantees least privilege since every hook is post-dominated by the SSO instance for which it was placed.

First, the hoisting in Algorithm 3 hoists the hooks pertaining to equivalent SSOs on all branches of a control statement in a bottom-up fashion in the CDG. The properties are discussed:

1. The hoisting operation introduces a new hook which dominates the control statement. This new hook preserves least privilege since all the branches of the control statement (therefore all paths originating from the new hook) must contain instances of operations that are equivalent to the one mediated by the new hook.

2. This stage does not remove any hooks so complete mediation is preserved.

Second, the redundancy removal stage in Algorithm 4 propagates information about hooks placed in a top-down fashion in the CDG. The properties are discussed:

1. The removal operation does nothing to violate least privilege since it does not add additional hooks.

2. When each node n of the CDG is processed, the set of propagated hooks that reach n represent the hooks that control dominate n. Therefore, if a hook h placed at node n is subsumed by or equivalent to any hook in the set of propagated hooks, then h can be safely removed without violating the complete-mediation guarantee.

Additionally given sound and complete alias analysis we can also guarantee a minimality in hook placement (constrained by complete mediation and least privilege). The construction of our algorithm guarantees that both hoisting and removal at each node are performed transitively in the context of all successors and predecessors respectively. Therefore, using an oracle-based argument similar to the proof in [110] we can show that with respect to a given set of authorization constraints after using our technique to remove hooks, no additional hoisting or removal can be performed, resulting in a minimal placement.

5.5 Evaluation

Our goal with the evaluation was to answer two questions:

- Does the approach reduce programmer effort necessary to place authorization hooks?
- Does the approach produce placements that are closer to manually placed hooks?

We evaluated our approach to determine how our approach can help reduce programmer effort by either making hoisting and removal choices on their behalf. Our results show that we are able to reduce the amount of manual programmer effort by as much as 67% and produce placements that reduce the number of hooks by as much as 36%, reducing the gap between manual and automated placements by as much as 70%.

Measuring Programmer Effort and Hook Count: The main effort for the programmer is in reasoning about the suggested constraints in the *bottom-up* approach and implied constraints in the *top-down* approach. Therefore, we measure programmer effort by counting the number of *hoisting* and *removal* choices that the bottom-up approach makes automatically when compared to the default placement. The results for this experiment are shown in Table 5.2. We also show the results of the top-down approach in Table 5.5. Placements generated using constraint selectors also reduce the total number of hooks generated and the gap between manual and automated placements as shown in Table 5.1.

We ran our experiments on four programs:

- **X Server 1.13**: X Server accepts requests for graphical output (windows) and sends back user input from keyboard, mouse, or touchscreen to clients. It has been manually retrofitted with XACE [88] hooks since 2007. Our results show that we are able to reduce the amount of programmer effort by 67% using the bottom up approach and by 40% using the top-down approach. The number of hooks generated is reduced by 36%(reducing the gap between manual and automated placements by 70%).

- **Postgres 9.1.9:** It is an open-source object-relational database management system. It has discretionary access control hooks. The version 9.1.9 also has sepgsql [114] hooks, but these are incomplete according to the documentation of the module [114]. Therefore, we do not consider these currently. Our experiments show that we are able to reduce the amount of programmer effort by 32% using the bottom-up approach and by 22% using the top-down approach. The number of hooks generated is reduced by 15%(reducing the gap between manual and automated placements by 46%).

- **Linux kernel 2.6.38.8 virtual file system (VFS):** The VFS allows clients to access different file systems in a uniform manner. The Linux VFS has been retrofitted with mandatory access control hooks in addition to the discretionary hooks. Our results show that there is an 17% reduction in programmer effort using the bottom-up approach and 3% using the top-down approach. The number of hooks is reduced by 4.

- **memcached:** It is a general-purpose distributed memory caching system used to speed up dynamic database-driven websites. It does not currently have any hooks. Our experiments show that constraint selectors are able to reduce the amount of programmer effort by 33% using the bottom-up approach and the number of hooks by 22%.

Program	**LOC**	**MAN**	**DEF**	**MLS**			**RUN (0.80)**			**MLS+RUN**		
				Total	*RO*	*PDR*	*Total*	*RO*	*PDR*	*Total*	*RO*	*PDR*
X Server 1.13	28K	207	420	296	30	58	393	6	13	270	36	70
Postgres 9.1.9	49K	243	360	326	9	29	344	4	14	306	15	46
Linux VFS 2.6.38.8	40K	55	139	135	2	5	139	0	0	135	2	5
Memcached	8K	0	32	30	6	n/a	31	3	n/a	25	22	n/a

Table 5.1: Table showing the lines of code (LOC), number of manual hooks (MAN), default hooks (DEF), and number of hooks generated using the MLS, RUN and MLS+RUN constraint selectors for hook placement.

Program	**DEF**		**MLS**		**RUN**		**MLS+RUN**	
	REM	*HST*	*REM*	*HST*	*REM*	*HST*	*REM*	*HST*
X Server 1.13	237	55	113	10	216	39	92	4
Postgres 9.1.9	208	42	146	21	182	22	161	9
Linux VFS 2.6.38.8	53	4	49	3	53	4	49	3
Memcached	8	1	6	0	12	1	10	1

Table 5.2: Table showing the hoisting (HST) and removal (REM) suggestions in the bottom-up approach for the default placement and placements generated using the constraint selectors.

5.5.1 Using constraint selectors to automate hook placement

In our experiments we use three constraint selectors.

- **MLS** enables the enforcement of the MLS security property (discussed in Section 5.4.2.3).
- **RUN** that suggests hoists based on run-time code coverage information gathered using the gcov [115] tool. Whenever a particular branch of a control statement is executed more than a threshold fraction of the total number of executions of the dominating control statement, then the SSOs along that branch are hoisted. The idea is that if a branch is executed predominantly (and the dominating branch has all permissions of any other branches), then all requesters should obtain the permissions for that branch for all executions that lead to the dominating control statement. Tables 5.1, 5.2 and 5.5 use the default threshold of 0.80, but in Section 5.5.1.2 we show Table 5.4 with results for two additional thresholds (0.7 and 0.9).

File	DEF		MLS		RUN		MLS+RUN	
	REM	*HST*	*REM*	*HST*	*REM*	*HST*	*REM*	*HST*
events.c	57	17	21	2	41	5	15	0
window.c	41	2	27	1	35	1	15	1
gc.c	48	3	23	1	48	3	23	1
colormap.c	40	17	17	2	38	16	14	0
dispatch.c	0	0	0	0	0	0	0	0
devices.c	13	4	5	0	13	4	5	0

Table 5.3: Distribution of hooks and hoisting and removal choices among files in X Server 1.13. These 6 files have 80% of all the hooks in X Server. Column 'REM' refers to removal choices, and 'HST' refers to hoisting choices.

- **MLS+RUN** We also show that constraint selectors may be combined in useful ways to generate placements. In this case, we combined them by applying the MLS after RUN at each hoist and only MLS at each removal. For 'RUN', we used the default threshold of 0.8. The results are shown under 'MLS+RUN' in all tables.

Evaluating the number of hooks: Table 5.1 shows the total number of hooks placed for each experiment. 'LOC' shows the number of lines of code that were analyzed, 'MAN' shows the number of hooks placed manually by domain experts and 'DEF' shows the number of hooks placement in the default case by the automated technique. Each of the remaining three columns - 'MLS', 'RUN' and 'MLS+RUN' shows how using constraint selectors affects the total number of hooks and how this compares with the number of hooks placed manually. In each of the constraint selector columns, 'Total' refers to the total number of hooks placed when using the constraint selector, 'RO' refers to the percentage reduction in number of hooks compared to the default placement and 'PDR' refers to percentage reduction in the gap between automated and manual placement when compared against the default placement. In the case of X Server we see that in the experiment we considered 28K lines of code, where programmers had placed 207 hooks manually, whereas the default placement suggested 420 hooks. When the 'MLS' constraint selector was used, the number of hooks suggested by the automated technique went down to 296 which is a 30% reduction in the number of hooks compared to manual placement and reduces the gap between automated and manual hook placements by 58% . With the 'RUN' constraint selector 393 hooks were suggested which is a 6% reduction in the number of hooks compared to default placements and reduces the gap between manual and automated placements by 13%. With the 'MLS+RUN' only 270 hooks were suggested by the automated tool which is a 36% reduction in the number of hooks compared to default placement and reduces the gap between manual and automated placements by 70%.

Evaluating programmer effort: Our results for the bottom-up approach using constraint selectors are shown in Table 5.2, showing the number of removal choices(REM) and hoisting choices(HST) for each experiment. For example, there were 237 removal choices and 55 hoisting choices left for the programmer after the default placement for X Server. After the bottom-up approach using MLS that automatically selects a subset of constraints, there are 113 removal and 10 hoisting choices which the programmer has to select from. This implies that using constraint selectors in the bottom-up approach has reduced the number of next-level choices that the programmer has to make. Making some set of hoisting and removal choices may expose additional choices due to newly introduced dominance and branch relationships. Therefore the number of choices shown in this table is not a measure of the total remaining programmer effort but only of the next set of choices available to the programmer. The hoisting choices combine those at `if` and `switch` statements. Since `switch` statements have a higher fan out compared to `if` statements, programmers will reason about more hooks per choice than with hoisting choices from `if` branches. In the following sections we discuss the programmer effort only the bottom-up approach. In Section 5.5.2 we will discuss the results from applying the top-down approach.

It often happens that different programmers are in charge of different parts of an application. To aid in this, we can show the file-wise split of the bottom-up choices for applications. Table 5.3 shows six files (of a total of 17 files with hooks) where 80 percent of the hooks are concentrated. Only two hoisting choices map to 45 hooks in `gc.c` and a single hoisting choice maps to 14 hooks in `window.c`.

5.5.1.1 Hook Placement using MLS

Effect on programmer effort: Our results show that the reduction in programmer effort is between 25-100% for hoisting choices and between 8-52% for removal choices.

In the case of X Server, there are only 10 hoisting choices left after 'MLS' which is an 82% reduction from the 55 choices in the default case. Of the 10 choices, seven are due to `switch` constructs. One of the hoisting choices made automatically by the MLS approach was in COPYGC (shown in Figure 4) which has a fan-out of 23 where all the 23 hooks were reading and writing to the same variable. How easy is it to make the remaining choices? Among the remaining choices, is a `switch` statement in function CHANGEGC which has a similar fan-out as COPYGC except one hook which writes to a different variable, therefore MLS constraint selector does not hoist there. We show this information to the programmer and make the choice easier for them. The number of removal choices is reduced by 124 (52% reduction).

In the case of Postgres, the number of hoisting choices is reduced by 21 (50% reduction) and the number of removal choices is reduced by 62 (30% reduction) compared to the default case. Majority of the 21 hoisting choices still left are due to `Switch-Case` statements, but MLS is unable to make many hoisting choices since these `Switch-Case` branches are used to choose between fundamentally different database operations and therefore work on different data structures and variables.

In the case of the linux VFS, we see that 'MLS' eliminates one hoisting choice of the four (25% reduction) in the default placement and four removal choices (8% reduction). All the removal choices in Linux VFS fell into one of three categories:

- Interprocedural hook dominance relationships where the security-sensitive objects being guarded by the hooks were of different types. For example, the hook in function `do_rmdir` dominates the hook in `vfs_rmdir`, therefore there is a removal opportunity. But the hook in the former mediates an object of type `nameidata` and the latter mediates an object of type `struct dentry *`. Our approach currently only performs removal when the hooks mediate the same variable. Of the 49 removal choices left after MLS, 24 belong to this category. In cases like this, we would need programmer input to make the choice.

- Interprocedural hook dominance relationships where the object mediated is of the same type but because it is across procedure boundaries and our approach does not employ alias analysis, it conservatively assumes that they are different objects. Therefore, it cannot perform removal. Of the 49 removal choices left after MLS, 19 belong to this category. These cases can be dealt with by using a powerful alias analysis.

- Intraprocedural hooks on the same object but one mediates reads and the other mediates writes. The 'MLS' constraint selector forbids removal in these cases. There were 6 such cases.

In the case of memcached, the 'MLS' constraint selector removes all hoisting choices and reduces the number of removal choices by 2 (25% reduction).

File-wise split: Examining Table 5.3, we see that MLS has a significant effect on the hoisting choices in `events.c` and `colormap.c` (from 17 to only two in both cases) but has no effect on the hooks in `dispatch.c` since those in `dispatch.c` are part of the main dispatch routine and cannot be hoisted any further, and therefore there are no choices to be made. We see that there are at most 2 hoisting choices to be made per file.

Effect on number of hooks: Our results (in table 5.1) show that in the MLS constraint selector that there is as much as 30% reduction in the total number of hooks placed reducing the gap between manual and automated hook placements by as much as 58%. We see that compared to the default placement the 'MLS' constraint selector produced 127 fewer hooks (30% reduction) in X Server, 34 fewer hooks in the case of postgres (9% reduction), 4 fewer in the case of Linux VFS, and 2 fewer hooks in the case of memcached. In X Server, Postgres and Linux VFS, this number gets us closer to the number of manually placed hooks by 58%, 19% and 5% respectively. The programmers in these projects have stated that the manual hook are not complete, so the difference between manual and automated placements might provide a useful way address this incompleteness.

5.5.1.2 Hook Placement using RUN

This constraint selector is meant to encourage hoisting, therefore we should ideally see reduction in `hoisting` choices.

Effect on programmer effort: We see that there is a (0 - 50%) reduction in the number of hoisting choices and (0 - 13 %) reduction in the number of removal choices. In case of X Server, we see that there are 16 fewer hoisting choices (29% reduction) and 21 fewer removal choices (8% reduction). In Postgres, we see 20 fewer hoisting choices (50% reduction) and 26 fewer removal choices (13% reduction). In Linux VFS we see no changes in both hoisting and removal choices - the reasons being that the benchmark we used to generate the runtime code coverage data did not find a dominant branch that could be hoisted. In case of memcached, the hoisting choices remain the same but the removal choices increase by four. The reason is that the 'RUN' constraint selector ended up performing a hoist - but this introduced new dominance relationships and therefore four additional removal choices. In addition, the hoisting ended at a branch of a control statement and therefore introduced another hoisting choice.

File-wise Split: When we look at the file-wise split we see that while the number of hoisting choices in `events.c` is reduced by 12 (71% reduction), in case of `colormap.c`, it only reduces by one. More than half of the remaining hoisting choices are concentrated in this file whereas removals are more evenly spread out among the first four files.

Effect on number of hooks: Our results (in table 5.1) show that in the RUN constraint selector there is 0-6% reduction in the total number of hooks placed reducing the gap between manual and automated hook placements by 0-14%. We see that compared to the default placement the 'RUN' constraint selector produced 27 fewer hooks (6% reduction) in X Server, 16 fewer hooks in the case of postgres (4% reduction), no change in Linux VFS, and one less hook in the case of memcached. In X Server and Postgres, this number gets us closer to the number of manually placed hooks by 13%, 14% respectively. There

Program	**RUN (0.7)**	**RUN (0.8)**	**RUN (0.9)**
X Server 1.13	390 (7%)	393 (6%)	397 (5%)
Postgres 9.1.9	340 (6%)	341 (5%)	344 (4%)
Linux VFS 2.6.38.8	139 (0%)	139 (0%)	139 (0%)
Memcached	31 (3%)	31 (3%)	31 (3%)

Table 5.4: The number of hooks placed using the 'RUN' constraint selector with three different thresholds - 0.7, 0.8 and 0.9

is no change in the case of Linux VFS because no hoisting ended up being performed as discussed in the previous subsection.

In addition to the default threshold of 0.80 we also tried the automated hook placements using two additional thresholds - 0.7 and 0.9. The results for this experiment are shown in Table 5.4 which shows the total number of hooks placed for each experiment along with the percentage reduction in the number of hooks compared to the default placement (shown in parenthesis). We can see that in the case of X Server and Postgres the reduction over default placement goes up by one percentage when increasing the threshold by 0.1. In the Linux VFS case there is no change in the hook placement even at the lower threshold of 0.7 and in memcached the reduction remains the same at all three thresholds.

5.5.1.3 Hook Placement using 'MLS+RUN'

This experiment combines the 'MLS' and 'RUN' constraint selectors. In general, we would expect this combination of consselects to do at least as well as the one that performs best (i.e., results in more hoisting and removal). If the hoisting and removal choices made by 'MLS' and 'RUN' happen on different branches/dominance relationships then the cumulative effect will be better than each of them applied individually.

Effect on programmer effort: We see that there is a (25 - 93%) reduction in the number of hoisting choices and (0 - 61%) reduction in the number of removal choices. In case of X Server, we see that there are 51 fewer hoisting choices (93% reduction) and 145 fewer removal choices (61% reduction). In Postgres, we see 33 fewer hoisting choices (79% reduction) and 47 fewer removal choices (23% reduction). Therefore, in both X

Server and Postgres the combination of ‘MLS’ and ‘RUN’ constraint selectors does better than each of them applied separately. In Linux VFS, we see that the effect is similar to ‘MLS’ alone since we saw that ‘RUN’ produced no change. In case of memcached, the hoisting choices remain the same but the removal choices increase by two. The reason is that ‘MLS’ and ‘RUN’ constraint selectors end up hoisting at different branches/dominance relationships. Therefore we get the effect of ‘MLS’ applied separately which eliminates two removal choices combined with the ‘RUN’ constraint selector which introduces four removal choices to result in a total of two additional removal choices.

File-wise split: When we look at the file-wise split for X Server in table 5.3 we see that the number of hoisting choices is down to 0 for all but two files.

Effect on number of hooks: We see that combining filters produces fewer hooks than each constraint selector applied separately in three cases. Our results (in table 5.1) show that there is 0-36% reduction in the total number of hooks placed reducing the gap between manual and automated hook placements by 0-70%. We see that compared to the default placement the ‘MLS+RUN’ constraint selector produced 150 fewer hooks (36% reduction) in X Server, 54 fewer hooks in the case of Postgres (15% reduction), four fewer in Linux VFS (3% reduction), and seven fewer hooks (22% reduction) in the case of memcached. In X Server and Postgres, this number gets us closer to the number of manually placed hooks by 70% and 32% respectively.

5.5.2 Evaluating the top-down approach

In this experiment, we assumed that the manual placement was the programmers’ candidate placement. We then decided to compare the manual placement against the default placement and the placements generated by ‘MLS’, ‘RUN’ and ‘MLS+RUN’. Using the approach in Section 5.4.2.2, we generated a set of implied constraints for each case. Table 5.5 shows both the number of equivalence and subsumption constraints generated in each case. The difference between the hoisting and removal choices in the bottom-up experiment and the constraints in this experiment are that in the former, it is the next set of

Program	Constraints	DEF	MLS	RUN	MLS+RUN
X Server	Equivalence	114	72	101	70
	Subsumption	55	38	49	32
Postgres	Equivalence	101	77	82	72
	Subsumption	53	44	57	48
Linux-vfs	Equivalence	19	17	19	17
	Subsumption	38	38	38	38

Table 5.5: Table showing the equivalence and subsumption constraints implied by manual placements by comparing against the default placement and those generated using constraint selectors for X Server.

possible hoists or removals, whereas here it is the set of all locations where hoisting and removal is necessary to get to the manual placement from the placements generated using constraint selectors.

We find that there are fewer constraints generated when comparing manual placements with 'MLS' than with 'DEF' implying that 'MLS' gets us closer to manual placements. 'RUN' has higher implied constraints than 'MLS' but 'MLS+RUN' has fewer constraints than both 'MLS' and 'RUN'.

We see that in the case of X Server that MLS+RUN experiment results in 40% reduction in the programmer effort in reasoning about the difference between the candidate and automated hook placements. In the case of Postgres there is a 22% reduction in the number of constraints. In Linux VFS, we see a 11% reduction in equivalence constraints but no change in subsumption constraints.

5.5.3 Discussion

In this section, we first discuss how alias analysis affects hook placement and then try demonstrate other ways in which the techniques described in Section 5.4 may be used.

5.5.3.1 Effect of aliasing

Currently our approach does not use any pointer analysis to determine if objects can interprocedurally alias each other. Therefore, removal operations are not effective across procedure boundaries. We found that the CIL alias analyser returns a very large points-to sets for each variable and is therefore fairly imprecise. We believe that given an effective alias analysis module, our approach will produce fewer hooks and further reduce programmer effort.

5.5.3.2 Transferring hook placements between versions

We have examined whether we can extract an authorization constraints from one version of a codebase that already has hooks to produce a hook placement for a different version of the codebase. We were able to extract a set of authorization constraints from version 1.9 of the X Server and apply to version 1.13. This results in 276 hooks (76 more than the 200 manual hooks in 1.13). We found that of the 420 default hooks in version 1.13, the 249 were carried over to the new placement as is and 27 hooks were mediating a larger set of SSOs than the default (because of hoisting) and the remaining were removed. The 76 hooks additional to manual placements are caused because of changes to the codebase, such as changing variable names, and addition of code that changes the control flow or imprecision in the generation of placement policies. In future work, we will examine whether we can abstract authorization constraints such that they are tolerant to certain code changes in order to transfer hook placements automatically across versions.

Chapter 6

Conclusion

Researchers have been trying to tackle the problem of automatically placing authorization hooks in programs for many years, but these efforts require too much manual specification. In this thesis, we have shown that through static analysis we are able to successfully infer the set of security-sensitive operations in programs with very little programmer specification. Our approach to solving this problem is based on the notion that user-requests serve as the vehicle for choosing security -sensitive objects and operations in programs. We combine static analysis with graph theory to place authorization hooks that balance the need for minimizing redundancy while still guaranteeing complete mediation. We also show what drives the choices of hook placements by programmers and incorporate the notion of placement choices into our framework to make it more useful to programmers.

Future work arising from this work will address three aspects. First, the precision of the analysis can be improved by incorporating alias analysis into the technique for hook placements. Second, we need to find a way to display the information to the programmer in a manner that will be most useful to them. For instance, we can show hook placements for each file, incorporated into an IDE along with the set of structure member accesses that it mediates, the evidence for why they were considered security-sensitive, and the hoisting and removal choices that relate to the particular hook.

Bibliography

[1] (2006), "Implement keyboard and event security in X using XACE," https://dev.laptop.org/ticket/260.

[2] "The Postfix mail program," http://www.postfix.org.

[3] "Oracle Database," http://www.oracle.com/database.

[4] "Microsoft SQL Server," http://www.microsoft.com/sql.

[5] KARGER, P. A., U. ROGER, and R. SCHELL (1974) *Multics security evaluation: Vulnerability analysis*, *Tech. rep.*, HQ Electronic Systems Division: Hanscom AFB, MA. URL: http://csrc.nist.gov/publications/history/karg74.pdf.

[6] NECULA, G. C., S. MCPEAK, and W. WEIMER (2002) "CCured: Type-safe Retrofitting of Legacy Code," in *Proceedings of the ACM Conference on the Principles of Programming Languages*.

[7] BRUMLEY, D. and D. SONG (2004) "Privtrans: automatically partitioning programs for privilege separation," in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, USENIX Association, Berkeley, CA, USA, pp. 5–5.
URL http://dl.acm.org/citation.cfm?id=1251375.1251380

[8] KING, D., S. JHA, D. MUTHUKUMARAN, T. JAEGER, S. JHA, and S. A. SESHIA (2010) "Automating security mediation placement," in *Proceedings of the 19th European conference on Programming Languages and Systems*, ESOP'10, Springer-Verlag, Berlin, Heidelberg, pp. 327–344.
URL http://dx.doi.org/10.1007/978-3-642-11957-6_18

[9] HARRIS, W. R., S. JHA, and T. REPS (2010) "DIFC programs by automatic instrumentation," in *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, ACM, New York, NY, USA, pp. 284–296.
URL http://doi.acm.org/10.1145/1866307.1866340

[10] HARRIS, W. R., S. JHA, T. W. REPS, J. ANDERSON, and R. N. M. WATSON (2013) "Declarative, Temporal, and Practical Programming with Capabilities," in *IEEE Symposium on Security and Privacy*, pp. 18–32.

[11] ANDERSON, J. P. (1972) "Computer Security Technology Planning Study," *Physical Review E*, **Volume I**(ESD-TR-73-51), pp. 1–43.
URL http://seclab.cs.ucdavis.edu/projects/history/papers/ande72a.pdf

[12] GRAHAM, G. S. and P. J. DENNING (1972) "Protection — principles and practice," in *Proceedings of the AFIPS Spring Joint Computer Conference*, vol. 40, AFIPS Press, pp. 417–429.

[13] MCLEAN, J. (1990) "The Specification and Modeling of Computer Security," *Computer*, **23**, pp. 9–16.
URL http://dl.acm.org/citation.cfm?id=77577.77578

[14] LOSCOCCO, P. A., S. D. SMALLEY, P. A. MUCKELBAUER, R. C. T. AND̃ S. J. TURNER, and J. F. FARRELL (1998) "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments," in *Proceedings of the 21st National Information Systems Security Conferenc e*, pp. 303–314.

[15] WRIGHT, C., C. COWAN, and J. MORRIS (2002) "Linux security modules: General security support for the linux kernel," in *In Proceedings of the 11th USENIX Security Symposium*, pp. 17–31.

[16] EPSTEIN, J. and J. PICCIOTTO (1991) "Trusting X: Issues in Building Trusted X Window Systems -or- What's not Trusted About X?" in *Proceedings of the 14th Annual National Computer Security Conference*, Washington, DC, USA, a survey of the issues involved in building trusted X systems, especially of the multi-level secure variety.

[17] KILPATRICK, D., W. SALAMON, and C. VANCE (2003) *Securing the X Window system with SELinux*, *Tech. rep.*, NAI Labs.

[18] KLEIN, A. (2004), "Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning, and Related Topics," White Paper, Sanctum Inc.

[19] WIGGINS, D. (1996) *Analysis of the X Protocol for Security Concerns: Draft Version 2*, *Tech. rep.*, The X Consortium, Inc.

[20] D.WALSH, "SELinux/apache," http://fedoraproject.org/wiki/SELinux/apache.

[21] KOHEI, K., "Security Enhanced PostgreSQL," SEPostgreSQLIntroduction.

[22] LOVE, R. (2005), "Get on the D-BUS," http://www.linuxjournal.com/article/7744.

[23] CARTER, J. (2007) "Using GConf as an Example of How to Create an Userspace Object Manager," *2007 SELinux Symposium*.
URL http://www.nsa.gov/selinux/papers/gconf07-abs.cfm

[24] WATSON, R. N. M. (2001) "TrustedBSD: Adding Trusted Operating System Features to FreeBSD," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, USENIX Association, Berkeley, CA, USA, pp. 15–28.
URL http://dl.acm.org/citation.cfm?id=647054.715753

[25] VANCE, C., T. MILLER, and R. DEKELBAUM (2007) "Security-Enhanced Darwin: Porting SELinux to Mac OS X," *Proceedings of the Third Annual Security Enhanced Linux*.
URL http://selinux-symposium.org/2007/papers/01-SEDarwin.pdf

[26] FADEN, G. (2007) "Multilevel filesystems in solaris trusted extensions," in *Proceedings of the 12th ACM symposium on Access control models and technologies*, SACMAT '07, ACM, New York, NY, USA, pp. 121–126.
URL http://doi.acm.org/10.1145/1266840.1266859

[27] SAILER, R., T. JAEGER, E. VALDEZ, R. CACERES, R. NALD PEREZ, S. BERGER, J. L. GRIFFIN, and L. VAN DOORN (2005) "Building a MAC-Based Security Architecture for the Xen Open-Source Hyperviso r," in *Proceedings of the 2005 Annual Computer Security Applications Conference*, pp. 276–285.

[28] COKER, G. (2006) "Xen security modules (xsm)," *Xen Summit*, pp. 1–33.
URL http://xen.xensource.com/files/summit_3/coker-xsm-summit-090706.pdf

[29] "Argus PitBull," http://www.argus-systems.com/.

[30] "grsecurity," http://www.grsecurity.net/.

[31] "RSBAC: Rule Set Based Access Control," http://www.rsbac.org/.

[32] "Security-Enhanced Linux," http://www.nsa.gov/selinux.

[33] "AppArmor Application Security for Linux," http://www.novell.com/linux/security/apparmor/.

[34] "LIDS: Linux Intrusion Detection System," http://www.lids.org/.

[35] JAEGER, T., A. EDWARDS, and X. ZHANG (2004) "Consistency analysis of authorization hook placement in the Linux security modules framework," *ACM Transaction on Information and System Security*, **7**(2), pp. 175–205.

[36] ZHANG, X., A. EDWARDS, and T. JAEGER (2002) "Using CQUAL for Static Analysis of Authorization Hook Placement," in *Proceedings of the 11th USENIX Security Symposium*, pp. 33–48.

[37] TAN, L., X. ZHANG, X. MA, W. XIONG, and Y. ZHOU (2008) "AutoISES: automatically inferring security specifications and detecting violations," in *Proceedings of the 17th conference on Security symposium*, USENIX Association, Berkeley, CA, USA, pp. 379–394.
URL `http://dl.acm.org/citation.cfm?id=1496711.1496737`

[38] STAFFORD, J. A. (2000) *A formal, language-independent, and compositional approach to interprocedural control dependence analysis*, Ph.D. thesis, Boulder, CO, USA, aAI9979404.

[39] XU, J. and N. NAKKA (2005) "Defeating Memory Corruption Attacks via Pointer Taintedness Detection," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, IEEE Computer Society, Washington, DC, USA, pp. 378–387.
URL `http://dx.doi.org/10.1109/DSN.2005.36`

[40] NGUYEN-TUONG, A., S. GUARNIERI, D. GREENE, J. SHIRLEY, and D. EVANS (2005) "Automatically Hardening Web Applications Using Precise Tainting," in *In 20th IFIP International Information Security Conference*, pp. 372–382.

[41] SUH, G. E., J. W. LEE, D. ZHANG, and S. DEVADAS (2004) "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, ACM, New York, NY, USA, pp. 85–96.
URL `http://doi.acm.org/10.1145/1024393.1024404`

[42] KONG, J., C. C. ZOU, and H. ZHOU (2006) "Improving software security via runtime instruction-level taint checking," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, ACM, New York, NY, USA, pp. 18–24.
URL `http://doi.acm.org/10.1145/1181309.1181313`

[43] CRANDALL, J. R. and F. T. CHONG (2004) "Minos: Control Data Attack Prevention Orthogonal to Memory Model," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, IEEE Computer Society, Washington, DC, USA, pp. 221–232.
URL `http://dx.doi.org/10.1109/MICRO.2004.26`

[44] "Perl Programming Documentation," http://perldoc.perl.org/perlsec.html#Taint-mode.

[45] Song, D., D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena (2008) "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, Springer-Verlag, Berlin, Heidelberg, pp. 1–25.
URL http://dx.doi.org/10.1007/978-3-540-89862-7_1

[46] Ganesh, V., T. Leek, and M. Rinard (2009) "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, IEEE Computer Society, Washington, DC, USA, pp. 474–484.
URL http://dx.doi.org/10.1109/ICSE.2009.5070546

[47] Newsome, J. (2005) "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proceedings of NDSS '05*.

[48] Clause, J., W. Li, and A. Orso (2007) "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, ACM, New York, NY, USA, pp. 196–206.
URL http://doi.acm.org/10.1145/1273463.1273490

[49] Cifuentes, C. and B. Scholz (2008) "Parfait: designing a scalable bug checker," in *Proceedings of the 2008 workshop on Static analysis*, SAW '08, ACM, New York, NY, USA, pp. 4–11.
URL http://doi.acm.org/10.1145/1394504.1394505

[50] Shankar, U., K. Talwar, J. S. Foster, and D. Wagner (2001) "Detecting format string vulnerabilities with type qualifiers," in *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, USENIX Association, Berkeley, CA, USA, pp. 16–16.
URL http://dl.acm.org/citation.cfm?id=1251327.1251343

[51] Evans, D. and D. Larochelle (2002) "Improving Security Using Extensible Lightweight Static Analysis," *IEEE Softw.*, **19**(1), pp. 42–51.
URL http://dx.doi.org/10.1109/52.976940

[52] Jovanovic, N., C. Kruegel, and E. Kirda (2006) "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)," in *IN 2006 IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, pp. 258–263.

[53] Livshits, V. B. and M. S. Lam (2005) "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th conference on USENIX*

Security Symposium - Volume 14, SSYM'05, USENIX Association, Berkeley, CA, USA, pp. 18–18.
URL http://dl.acm.org/citation.cfm?id=1251398.1251416

[54] Wassermann, G. and Z. Su (2007) "Sound and precise analysis of web applications for injection vulnerabilities," *SIGPLAN Not.*, **42**(6), pp. 32–41.
URL http://doi.acm.org/10.1145/1273442.1250739

[55] Chang, R., G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov (2009) "Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities," in *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, IEEE Computer Society, Washington, DC, USA, pp. 186–199.
URL http://dl.acm.org/citation.cfm?id=1602936.1603628

[56] Sharir, M. and A. Pnueli (1981) "Two Approaches to Interprocedural Dataflow Analysis," in *Program Flow Analysis: Theory and Applications* (S. Muchnick and N. Jones, eds.), Prentice Hall, pp. 189–233.

[57] Flanagan, C., K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata (2002) "Extended static checking for Java," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, ACM, New York, NY, USA, pp. 234–245.
URL http://doi.acm.org/10.1145/512529.512558

[58] Yang, J., T. Kremenek, Y. Xie, and D. Engler (2003) "MECA: an extensible, expressive system and language for statically checking security properties," in *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, ACM, New York, NY, USA, pp. 321–334.
URL http://doi.acm.org/10.1145/948109.948153

[59] (2008), "Jif = Java + information flow," http://www.cs.cornell.edu/jif/.

[60] Vaughan, J. A. and S. Chong (2011) "Inference of Expressive Declassification Policies," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, IEEE Computer Society, Washington, DC, USA, pp. 180–195.
URL http://dx.doi.org/10.1109/SP.2011.20

[61] Askarov, A. and A. Sabelfeld (2005) "Security-typed languages for implementation of cryptographic protocols: a case study," in *Proceedings of the 10th European conference on Research in Computer Security*, ESORICS'05, Springer-Verlag, Berlin, Heidelberg, pp. 197–221.
URL http://dx.doi.org/10.1007/11555827_12

[62] KREMENEK, T., P. TWOHEY, G. BACK, A. NG, and D. ENGLER (2006) "From uncertainty to belief: inferring the specification within," in *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, USENIX Association, Berkeley, CA, USA, pp. 161–176.
URL http://dl.acm.org/citation.cfm?id=1298455.1298471

[63] LIVSHITS, B., A. V. NORI, S. K. RAJAMANI, and A. BANERJEE (2009) "Merlin: specification inference for explicit information flow problems," *SIGPLAN Not.*, **44**(6), pp. 75–86.
URL http://doi.acm.org/10.1145/1543135.1542485

[64] ERNST, M. D., J. H. PERKINS, P. J. GUO, S. MCCAMANT, C. PACHECO, M. S. TSCHANTZ, and C. XIAO (2006) "The Daikon system for dynamic detection of likely invariants," in *Science of Computer Programming*.

[65] HANGAL, S. and M. S. LAM (2002) "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, ACM, New York, NY, USA, pp. 291–301.
URL http://doi.acm.org/10.1145/581339.581377

[66] BALIGA, A., V. GANAPATHY, and L. IFTODE (2011) "Detecting Kernel-level Rootkits using Data Structure Invariants," *IEEE Transactions on Dependable and Secure Computing*, **8**(5), pp. 670–684.

[67] FELMETSGER, V., L. CAVEDON, C. KRUEGEL, and G. VIGNA (2010) "Toward automated detection of logic vulnerabilities in web applications," in *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, USENIX Association, Berkeley, CA, USA, pp. 10–10.
URL http://dl.acm.org/citation.cfm?id=1929820.1929834

[68] LAMPSON, B. W. (1971) "Protection," in *5th Princeton Conference on Information Sciences and Systems*.

[69] HARRISON, M., W. RUZZO, and J. D. ULLMAN (1976) "Protection in Operating Systems," *Communications of the ACM*.

[70] EDWARDS, A., T. JAEGER, and X. ZHANG (2002) "Runtime verification of authorization hook placement for the Linux security modules framework," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 225–234.

[71] SAILER, R., T. JAEGER, E. VALDEZ, R. CACERES, R. PEREZ, S. BERGER, J. L. GRIFFIN, and L. VAN DOORN (2005) "Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor," in *Proceedings of the 2005 Annual Computer Security Applications Conference*, pp. 276–285.

[72] GANAPATHY, V., T. JAEGER, and S. JHA (2006) "Retrofitting Legacy Code for Authorization Policy Enforcement," in *SP'06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Los Alamitos, California, USA, Oakland, California, USA, pp. 214–229.

[73] GANAPATHY, V., D. KING, T. JAEGER, and S. JHA (2007) "Mining Security Sensitive Operations in Legacy Code using Concept Analysis," in *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, California, USA, Minneapolis, Minnesota, USA, pp. 458–467.

[74] SON, S., K. S. MCKINLEY, and V. SHMATIKOV (2011) "RoleCast: finding missing security checks when you do not know what checks are," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, ACM, New York, NY, USA, pp. 1069–1084.
URL http://doi.acm.org/10.1145/2048066.2048146

[75] POLITZ, J. G., S. A. ELIOPOULOS, A. GUHA, and S. KRISHNAMURTHI (2011) "ADsafety: type-based verification of JavaScript Sandboxing," in *Proceedings of the 20th USENIX conference on Security*, SEC'11, USENIX Association, Berkeley, CA, USA, pp. 12–12.
URL http://dl.acm.org/citation.cfm?id=2028067.2028079

[76] SUN, F., L. XU, and Z. SU (2011) "Static detection of access control vulnerabilities in web applications," in *Proceedings of the 20th USENIX conference on Security*, SEC'11, USENIX Association, Berkeley, CA, USA, pp. 11–11.
URL http://dl.acm.org/citation.cfm?id=2028067.2028078

[77] ERLINGSSON, Ú. and F. B. SCHNEIDER (2000) "IRM Enforcement of Java Stack Inspection," in *IEEE Symposium on Security and Privacy*, pp. 246–255.

[78] "A utility for monitoring keystrokes on remote X servers," http://www.freshports.org/security/xspy.

[79] "Screenshots with the scrot command," http://en.kioskea.net/faq/1270-screenshots-with-the-scrot-command.

[80] "Trusted X Window System," http://docs.oracle.com/cd/E19963-01/html/821-1483/windowapi-1.html.

[81] WILLE, R. (2009) "RESTRUCTURING LATTICE THEORY: AN APPROACH BASED ON HIERARCHIES OF CONCEPTS," in *Proceedings of the 7th International Conference on Formal Concept Analysis*, ICFCA '09, Springer-Verlag, Berlin, Heidelberg, pp. 314–339.
URL http://dx.doi.org/10.1007/978-3-642-01815-2_23

[82] Srivastava, V., M. D. Bond, K. S. McKinley, and V. Shmatikov (2011) “A security policy oracle: detecting security holes using multiple API implementations,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI ’11, ACM, New York, NY, USA, pp. 343–354.
URL http://doi.acm.org/10.1145/1993498.1993539

[83] (2013), “Spring Security,” http://static.springsource.org/spring-security/site/reference.html.

[84] (2013), “Apache Shiro,” http://shiro.apache.org/.

[85] Son, S., K. S. McKinley, and V. Shmatikov (2013) “Fix Me Up: Repairing access-control bugs in web applications,” in *Network and Distributed System Security Symposium*.

[86] Smalley, S., C. Vance, and W. Salamon (2001) *Implementing SELinux as a Linux Security Module*, *Tech. Rep. 01-043*, NAI Labs, http://www.nsa.gov/selinux/papers/module.pdf.

[87] Microsystems, S., “Trusted Solaris Operating Environment - A Technical Overview,” http://www.sun.com.

[88] (2009), “X Access Control Extension Specification,” http://www.x.org/releases/X11R7.6/doc/xorg-docs/specs/Xserver/XACE-Spec.html.

[89] Morris, J., “New secmark-based network controls for SELinux,” http://james-morris.livejournal.com/11010.html.

[90] Jaeger, T., K. Butler, D. H. King, S. Hallyn, J. Latten, and X. Zhang (2006) “Leveraging IPsec for Mandatory Access Control Across Systems,” in *Proc. 2nd Intl. Conf. on Security and Privacy in Communication Networks*.

[91] Moore, P. A. P., “NetLabel - Explicit Labeled Networking for Linux,” http://netlabel.sourceforge.net.

[92] McCamant, S. and M. D. Ernst (2008) “Quantitative information flow as network flow capacity,” *SIGPLAN Not.*, **43**(6), pp. 193–205.
URL http://doi.acm.org/10.1145/1379022.1375606

[93] Myers, A. C. (1999) “JFlow: Practical Mostly-Static Information Flow Control,” in *POPL ’99*, pp. 228–241.
URL http://www.cs.cornell.edu/andru/papers/popl99/popl99.pdf

[94] JAEGER, T., R. SAILER, and X. ZHANG (2003) "Analyzing Integrity Protection in the SELinux Example Policy," in *Proceedings of the 12th USENIX Security Symposium*, pp. 59–74.

[95] SARNA-STAROSTA, B. and S. D. STOLLER (2004) "Policy Analysis for Security-Enhanced Linux," in *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pp. 1–12.

[96] TRESYS, "SETools - Policy Analysis Tools for SELinux. Available at http://oss.tresys.com/projects/setools," .
URL `http://oss.tresys.com/projects/setools`

[97] DAHLHAUS, E., D. S. JOHNSON, C. H. PAPADIMITRIOU, P. D. SEYMOUR, and M. YANNAKAKIS (1994) "The Complexity of Multiterminal Cuts," *SIAM J. Comput.*, **23**, pp. 864–894.
URL `http://portal.acm.org/citation.cfm?id=182366.182379`

[98] MUCHNICK, S. S. (1997) *Advanced Compiler Design and Implementation*, Morgan Kaufmann.

[99] PROVOS, N. (2003) "Improving Host Security With System Call Policies," in *Proceedings of the 2003 USENIX Security Symposium*.

[100] CLARK, D. D. and D. WILSON (1987) "A Comparison of Military and Commercial Security Policies," in *IEEE Symposium on Security and Privacy*.

[101] NECULA, G. C., S. MCPEAK, S. P. RAHUL, and W. WEIMER (2002) "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Compiler Construction, 11th International Conference, CC 2002*, Springer, pp. 213–228.

[102] STAFFORD, J. A. (2000) *A Formal, Language-Independent, and Compositional Approach to Interprocedural Control Dependence Analysis*, Ph.D. thesis, University of Colorado.

[103] LIVSHITS, V. B. and M. S. LAM (2005) "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, USENIX Association, Berkeley, CA, USA, pp. 18–18.
URL `http://dl.acm.org/citation.cfm?id=1251398.1251416`

[104] WASSERMANN, G. and Z. SU (2007) "Sound and precise analysis of web applications for injection vulnerabilities," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, ACM, New York, NY, USA, pp. 32–41.
URL `http://doi.acm.org/10.1145/1250734.1250739`

[105] (2006), "Implement keyboard and event security in X using XACE," https://dev.laptop.org/ticket/260.

[106] (2009), "SE-PostgreSQL?" http://archives.postgresql.org/message-id/20090718160600.GE5172@fetter.org.

[107] GONG, L. and R. SCHEMERS (1998) "Implementing Protection Domains in the Java™ Development Kit 1.2," in *NDSS*.

[108] SALTZER, J. H. and M. D. SCHROEDER (1975) "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, **63**(9), pp. 1278–1308.

[109] WALSH, E. (2007) "Application of the Flask Architecture to the X Window System Server," in *Proceedings of the 2007 SELinux Symposium*.

[110] MUTHUKUMARAN, D., T. JAEGER, and V. GANAPATHY (2012) "Leveraging "Choice" to Automate Authorization Hook Placement," in *CCS'12: Proceedings of the 19th ACM Conference on Computer and Communications Security*, ACM Press, p. TBD.

[111] BELL, D. E. and L. J. LAPADULA (1976) *Secure Computer System: Unified Exposition and Multics Interpretation*, *Tech. Rep. ESD-TR-75-306*, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA.

[112] (2008), "Xorg-Server Announcement," http://lists.x.org/archives/xorg-announce/2008-March/000458.html.

[113] (1995), "Multilevel Security in the Department Of Defense: The Basics," http://nsi.org/Library/Compsec/sec0.html.

[114] (2013), "F.38. sepgsql," http://www.postgresql.org/docs/9.1/static/sepgsql.html.

[115] (2013), "10 gcova Test Coverage Program," http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

Vita

Divya Muthukumaran

Education

- The Pennsylvania State University, Aug 2006 - Dec 2013
  Ph.D., Computer Science and Engineering, GPA - 3.81

- Anna University, Chennai, India, Aug 2001 - May 2005
  B.E. with Distinction, Computer Science and Engineering, 82 percent

Professional Experience

- **Senior Software Security Research Intern**, *HP Fortify*, Sunnyvale, CA, May 2013 - July 2013

- **Research Intern** *Hewlett Packard Labs*, Bristol, UK, July 2012 - Oct 2012

- **Assistant Systems Engineer** *Tata Consultancy Services (TCS)*, India, September 2005 - July 2006

- **Technical Intern** *Sify India Ltd* , India, May - Jul 2003