RETROFITTING LEGACY CODE
FOR AUTHORIZATION POLICY ENFORCEMENT

by

Vinod Ganapathy

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2007

In fond remembrance of my thatha Shri. V. Lakshminarayanan,
my paati Smt. Sharada Seshan, kollu paati, and Bruno.

ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to:

- my teachers and peers from IIT Bombay, for being a never-ending source of inspiration and for giving my career the best start that I could have hoped for.
- Somesh Jha, for six years of patient advice and support, for ensuring me a productive research environment, and for shaping me as a researcher.
- Trent Jaeger, for being a wonderful mentor and for being unconditionally supportive of all my efforts.
- Trishul Chilimbi, Thomas Reps, Sanjit Seshia, Michael Swift, and all my collaborators over the years for giving me the opportunity to work with and learn from them.
- Trent Jaeger, Susan Horwitz, Marvin Solomon, Michael Swift and Parameswaran Ramanathan, for serving on my thesis committee.
- my academic siblings and other members of the security research group at UW-Madison, for making the group a productive and enjoyable one to do research in.
- all my friends over the years, both at UW-Madison and at IIT Bombay, for support, encouragement, and for keeping me sane.

Thank you very much!

My biggest source of advice, encouragement, motivation and support are undoubtedly Amma and Appa, and mere words will do injustice to express my gratitude to them. Their guidance and sacrifice made this possible, and their affection and belief in me made it all worthwhile. My success is also theirs. To them, I dedicate this work.

Funding. My graduate studies were funded by ONR grants N00014-01-1-0708 and N00014-01-1-0796 and NSF grant CNS-0448476. The Secure Systems Department at IBM Thomas J. Watson Research Center (Hawthorne, NY) and the Runtime Analysis and Design Group at Microsoft Research (Redmond, WA) employed me as a summer intern in Summer 2005 and Summer 2004, respectively. Their support is gratefully acknowledged.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

ABSTRACT x

1 Introduction 1

1.1 Motivation 1
1.2 Retrofitting authorization policy enforcement mechanisms 3
1.3 Current practice 5
1.4 Contributions 10
1.5 Structure of this dissertation 11
1.6 Bibliographic attributions 12

2 Overview 13

2.1 Goal 13
2.2 Assumptions 13
2.3 A note about the trusted computing base 15
2.4 Basic tools 16
2.4.1 The enforcer 17
2.4.2 The reference monitor 17
2.5 Our approach 18
2.5.1 An example: Retrofitting the X server 18
2.5.2 Step 1: Find security-sensitive operations to be protected 19
2.5.3 Step 2: Find fingerprints of security-sensitive operations 20
2.5.4 Step 3: Find all locations that are security-sensitive 23
2.5.5 Step 4: Instrument the server 25
2.5.6 Step 5: Generate the reference monitor 26
2.5.7 Step 6: Link the modified server and reference monitor 26
2.6 Discussion I: Security analysis 27

Page

2.7 Discussion II: Why retrofit the server? 28

3 Fingerprints 29

3.1 Syntax 29
3.2 Interpretation 33

4 Mining Fingerprints using Dynamic Program Analysis 34

4.1 Problem statement 34
4.2 Identifying fingerprints using analysis of program traces 35
4.3 Implementation 38
4.4 Evaluation of the dynamic fingerprint mining algorithm 41
4.4.1 How effective is A at locating fingerprints? 41
4.4.2 How precise are the fingerprints found? 43
4.4.3 How much effort is involved in manual labeling of traces? 43
4.4.4 How effective is manual labeling of traces? 43
4.5 Limitations 44
4.6 Using the dyamic fingerprint mining tool 45
4.7 Summary of key ideas 45

5 Mining Fingerprints using Static Program Analysis 47

5.1 Problem statement 47
5.2 Identifying fingerprints using static program analysis and concept analysis 48
5.2.1 Running example 48
5.2.2 Step A: From source code to building blocks 49
5.2.3 Step B: Refining building blocks 51
5.3 Extracting building blocks from code 54
5.3.1 Static analysis 55
5.3.2 Background on concept analysis 55
5.3.3 Using concept analysis 58
5.4 Refinement with constraints 59
5.5 Evaluation of the static fingerprint mining algorithm 62
5.5.1 The ext2 file system 63
5.5.2 The X11 server 65
5.5.3 The PennMUSH server 67
5.6 Limitations 69
5.7 Static fingerprint mining versus dynamic fingerprint mining 70
5.8 Using the static fingerprint mining tool 70

Page

5.9 Summary of key ideas 71

6 Using Fingerprints to Retrofit Legacy Code 73

6.1 Problem statement 73
6.2 Identifying security-sensitive locations 74
6.3 Evaluation of the matching algorithm 78
6.3.1 How precise are the security-sensitive locations found? 78
6.3.2 How easy is it to identify subjects and objects? 79
6.4 Synthesizing a reference monitor implementation 79
6.5 Example: Retrofitting the X server to enforce authorization policies 82
6.5.1 Example I: Setting window properties 82
6.5.2 Example II: Secure cut-and-paste 83
6.6 Performance of the retrofitted X server 84
6.7 Analyzing a reference monitor implementation 84
6.8 Using the matching tool 91
6.9 Summary of key ideas 92

7 Related Work 93

7.1 Foundations of authorization 93
7.2 Authorization policy enforcement systems 94
7.3 Code retrofitting and refactoring systems 96
7.4 Aspect-oriented programming 97
7.5 Authorization policy formulation and analysis 98
7.6 Other related work 99
7.6.1 Root-cause analysis 99
7.6.2 X Window system security 100

8 Conclusions and Future Work 101

LIST OF REFERENCES 104

DISCARD THIS PAGE

LIST OF TABLES

Table Page

4.1 Examples of labeled traces obtained from the X server. A "✓" entry in (*row*, *column*) denotes that the trace represented by *column* performs the security-sensitive operation represented by *row*. A "$\times_{fn}$" or a "$\times_{fp}$" entry denotes a mistake in manual labeling. . 39

4.2 Set equations for security-sensitive operations computed using the annotations in Table 4.1, and the sizes of the resulting sets. 40

5.1 Steps to statically mine fingerprints of security-sensitive operations, and the techniques used in each step. Static analysis is first used in conjunction with concept analysis to extract building blocks. These building blocks are refined/composed to yield fingerprints. 49

5.2 Results for each of our case studies. The sixth column denotes the number of building blocks mined, while the seventh column shows their average size, in terms of the number of code patterns per building block. The table also shows the number of building blocks that had to be refined with precision constraints. Figure 5.6, Figure 5.7 and Figure 5.8 depict the concept lattices produced for each of these case studies. . . . 62

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

1.1 Retrofitting legacy software with a reference monitor interface. Lines with underlined comments must be retrofitted. 6

2.1 Steps involved in retrofitting a server for authorization policy enforcement. 19

2.2 X server function MapSubWindows 24

3.1 BNF grammar defining fingerprints. The symbol OP in the definition of F denotes the name of a security-sensitive operation. Note that an abstract syntax tree (AST) representing a resource access is represented using data types, denoting the data type of the resource being accessed. 30

4.1 Fingerprints obtained by analyzing set equations in Figure 4.1 by applying Algorithm 1 to the labeled traces from Table 4.1. 42

5.1 One of the building blocks that concept analysis identifies for ext2. 51

5.2 Example showing the need for precision constraints. 53

5.3 Fingerprints obtained after refinement with precision constraints. 54

5.4 Concept analysis example. 57

5.5 BNF grammar for constraints. 60

5.6 Concept lattice for ext2. The shaded nodes represent those marked by Algorithm 3, and the concepts represented by these node contain building blocks. This concept lattice has 21 nodes and 32 edges. Algorithm 3 identified 18 building blocks. 64

5.7 Concept lattice for the X server. This concept lattice has 329 nodes and 978 edges. Algorithm 3 identified 115 building blocks in this concept lattice. 66

Figure Page

5.8 Concept lattice for PennMUSH. This concept lattice has 127 nodes and 310 edges. Algorithm 3 identified 38 building blocks in this concept lattice. 68

6.1 Interprocedural fingerprints for four security-sensitive operations for the ext2 file system 77

6.2 A portion of the call-graph of the ext2 file system, rooted at the function `ext2_rmdir`. Code snippets relevant to the example are shown in boxes near the functions that they appear in. 77

6.3 Results of matching the interprocedural fingerprints shown in Figure 6.1. 78

6.4 Code fragment showing the implementation of `QueryRefmon` for Window_Create. . . 80

6.5 Code for the SELinux authorization query `selinux_inode_permission` (borrowed from the Linux-2.4.21 kernel). 85

RETROFITTING LEGACY CODE
FOR AUTHORIZATION POLICY ENFORCEMENT

Vinod Ganapathy

Under the supervision of Associate Professor Somesh Jha
At the University of Wisconsin-Madison

Research in computer security has historically advocated Design for Security, the principle that security must be proactively integrated into the design of a system. While examples exist in the research literature of systems that have been designed for security, there are few examples of such systems deployed in the real world. Economic and practical considerations force developers to abandon security and focus instead on functionality and performance, which are more tangible than security. As a result, large bodies of legacy code often have inadequate security mechanisms. Security mechanisms are added to legacy code on-demand using ad hoc and manual techniques, and the resulting systems are often insecure.

This dissertation advocates the need for techniques to retrofit systems with security mechanisms. In particular, it focuses on the problem of retrofitting legacy code with mechanisms for authorization policy enforcement. It introduces a new formalism, called fingerprints, to represent security-sensitive operations. Fingerprints are code templates that represent accesses to security-critical resources, and denote key steps needed to perform operations on these resources. This dissertation develops both fingerprint mining and fingerprint matching algorithms.

Fingerprint mining algorithms discover fingerprints of security-sensitive operations by analyzing source code. This dissertation presents two novel algorithms that use dynamic program analysis and static program analysis, respectively, to mine fingerprints. The fingerprints so mined are used by the fingerprint matching algorithm to statically locate security-sensitive operations. Program transformation is then employed to statically modify source code by adding authorization policy lookups at each location that performs a security-sensitive operation.

The techniques developed in this dissertation have been applied to three real-world systems. These case studies demonstrate that techniques based upon program analysis and transformation offer a principled and automated alternative to the ad hoc and manual techniques that are currently used to retrofit legacy software with security mechanisms.

Somesh Jha

ABSTRACT

Research in computer security has historically advocated Design for Security, the principle that security must be proactively integrated into the design of a system. While examples exist in the research literature of systems that have been designed for security, there are few examples of such systems deployed in the real world. Economic and practical considerations force developers to abandon security and focus instead on functionality and performance, which are more tangible than security. As a result, large bodies of legacy code often have inadequate security mechanisms. Security mechanisms are added to legacy code on-demand using ad hoc and manual techniques, and the resulting systems are often insecure.

This dissertation advocates the need for techniques to retrofit systems with security mechanisms. In particular, it focuses on the problem of retrofitting legacy code with mechanisms for authorization policy enforcement. It introduces a new formalism, called fingerprints, to represent security-sensitive operations. Fingerprints are code templates that represent accesses to security-critical resources, and denote key steps needed to perform operations on these resources. This dissertation develops both fingerprint mining and fingerprint matching algorithms.

Fingerprint mining algorithms discover fingerprints of security-sensitive operations by analyzing source code. This dissertation presents two novel algorithms that use dynamic program analysis and static program analysis, respectively, to mine fingerprints. The fingerprints so mined are used by the fingerprint matching algorithm to statically locate security-sensitive operations. Program transformation is then employed to statically modify source code by adding authorization policy lookups at each location that performs a security-sensitive operation.

The techniques developed in this dissertation have been applied to three real-world systems. These case studies demonstrate that techniques based upon program analysis and transformation

offer a principled and automated alternative to the ad hoc and manual techniques that are currently used to retrofit legacy software with security mechanisms.

Chapter 1

Introduction

This dissertation presents program analysis and transformation techniques to retrofit legacy software with mechanisms for authorization policy enforcement. Using case studies with real-world software systems, it demonstrates that these techniques offer a principled and automated alternative to the ad hoc and manual techniques that are currently used to retrofit legacy software.

1.1 Motivation

Design for Security, the principle that security must be a key design consideration in the construction of a secure system, has long been a mantra of the security community. Indeed, systems such as Multics [CV65] and Hydra [WCC+74], which provided strong security guarantees, were constructed with security as a principle design consideration.

While proactively designing for security will undoubtedly create more robust and secure systems, doing so is often difficult in practice because of two reasons.

First, because application functionality and performance are often more tangible features to a customer, software producers focus on these aspects, with security typically being an afterthought. Indeed, excluding the operating system, the number of applications that have been proactively designed for security is far outnumbered by applications that are not. Some examples of software that have proactively been designed for security include the Postfix mail program [Pos] and database servers [Ora, SQL]. There are thus large bodies of legacy software with inadequate or non-existent security mechanisms. It is impractical to require that these systems be redesigned and rebuilt for security. It is instead advisable to *retrofit* security mechanisms into these systems. However, because of the time and cost involved, even these security retrofits are currently undertaken only rarely for large legacy systems. For example, it took almost two years to modify the Linux kernel to add mechanisms that enforce mandatory access control (MAC) policies [1]. Similarly, the privilege-separated version of OpenSSH required a new system architecture, and the addition of a significant amount of new code [PFH03].

Second, even if a system is designed for security, additional security mechanisms may have to be added in the future. For example, while the Linux kernel has historically had mechanisms to enforce discretionary access control (DAC) policies, mechanisms to enforce MAC policies were added only in 2002 [WCS+02]. Moreover, prior experience shows that even if a system is designed for security, functional and performance enhancements that are added in the future may break security assumptions, thus calling for a reevaluation of the system's security. For example, Karger and Schell [KS74] showed that modifications to improve the usability of Multics broke several of its design assumptions, which resulted in potentially exploitable vulnerabilities.

These reasons motivate the need for retroactive techniques to secure software—automatic and semi-automatic techniques to reason about security properties of legacy software, and retrofit it with security mechanisms. Indeed, the need for such techniques has also been realized by others. For example, the CCured tool [NCH+05, NMW02] automatically retrofits legacy C programs into a type-safe variant by inserting runtime checks that enforce type safety. Similarly, the Privtrans tool [BS04] semi-automatically refactors legacy C programs for privilege separation using program partitioning.

[1] The Orange Book [TCS85] defines Mandatory access control as "a means of restricting access to objects based on the sensitivity (as represented by a security label) of the information contained in the objects and the formal authorization (*i.e.,* clearance) of subjects to access information of such sensitivity". In contrast, Discretionary access control (DAC) is defined as "a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject". Thus MAC denies users full control over access to resources, including those that they create. In contrast, in DAC, ownership of a resource allows a subject full control over access to the resource, including delegation of access rights to the resource. Examples of MAC policies include the Bell-LaPadula policy [BL76] and the Biba policy [Bib77], while the Graham-Denning model [GD72] is an example of a DAC policy. The specifics of MAC and DAC are not central to the techniques developed in this dissertation and will not be presented in further detail. For a good overview of MAC and DAC, see McLean [McL90].

This dissertation continues this line of research into retroactive techniques to secure legacy software, and focuses on the problem of retrofitting legacy software with mechanisms for authorization policy enforcement.

1.2 Retrofitting authorization policy enforcement mechanisms

Software systems that manage shared resources must protect these resources from unauthorized access. This is achieved by formulating and enforcing an appropriate authorization policy (also called an access control policy). This policy specifies the set of *security-sensitive operations* that a *subject* (typically a user of the system) can perform on an *object* (typically a resource managed by the system). For example, operating systems manage shared resources such as files, network connections and memory, and typically enforce policies that determine how users of the system can access these resources. A popular example of such a policy on UNIX-like systems allows only the `root` user to perform the operation Write on the `/etc/passwd` file (the resource).

Authorization policies are typically enforced using a security mechanism called a *Reference Monitor*. Introduced by Anderson in 1972 [And72], a reference monitor is an entity that satisfies three key properties.

1. **Complete mediation.** The reference monitor must be invoked at each access to a shared resource, *i.e.,* all security-sensitive operations performed on a shared resource must be mediated. Saltzer and Schroeder also call this property the *Principle of Complete Mediation* [SS75].

2. **Tamper resistance.** The reference monitor mechanism must be tamperproof, *i.e.,* an attacker must not be able to circumvent the mechanism, *e.g.,* by rewriting the code of the reference monitor, so that an access check is not performed (and the authorization policy not enforced) before a shared resource is accessed.

3. **Verifiability.** The reference monitor must be a small-enough entity so as to allow for thorough verification.

A reference monitor thus mediates each security-sensitive operation on a shared resource, and ensures that a subject is allowed to perform the operation on a resource only if it is allowed by the authorization policy. This dissertation develops a suite of techniques to retrofit a reference monitor into a legacy software system that lacks this security mechanism.

We consider two concrete examples to motivate the need to retrofit a reference monitor into legacy code: (1) the integration of the Linux Security Modules (LSM) framework [WCS+02] to the Linux operating system, and (2) the integration of a reference monitor with the X server [X11a].

While Linux has long had the ability to enforce DAC policies, the need was felt to extend these mechanisms to enforce MAC policies. This was achieved using the Linux security modules (LSM) framework [WCS+02], by retrofitting the Linux kernel. The creation of the LSM framework was motivated by the proliferation of different Linux patches that aimed to improve the default Linux access control mechanism through finer grained enforcement of MAC policies [Arg, GRS, RSB, SEL, App, LID]. The LSM framework generalizes this work to define a reference monitor interface for mediating all accesses to security-sensitive operations via loadable kernel modules in a policy-independent way. While there were differences between these patches in scope, policy models, and ancillary features, a single reference monitor interface can be defined that subsumes all approaches since the goal of a reference monitor is complete mediation of all security-sensitive operations. The development of the LSM framework from the multitude of Linux patches described above was a manual process of collecting authorization decision points (*i.e.,* calls to the reference monitor, or *hooks*) from each patch and resolving inconsistencies between these choices.

The X server, like many other server applications, enables multiple X clients to access shared resources (*e.g.,* windows, fonts) that it manages. However, the X server was historically developed to promote cooperation between X clients, and security (*e.g.,* isolation) of X clients was not built into the design of the server. In the absence of isolation within the X server, malicious clients can compromise the integrity and privacy of other X clients handled by the X server—well-documented instances of such attacks abound in the literature (*e.g.,* [EP91, KSV03, Kle04, Wig96a]). For example, in the X server, a malicious X client can easily compromise the privacy of other X clients by snooping on their input, or by retrieving bitmaps of their windows [KSV03]. Similarly, it is easy to program a Trojan horse X client that registers with the X server to receive keystrokes sent to other X clients connected to the X server (*e.g.,* the xkey application [Gia] does so in just 100 lines of C source code).

As was the case with Linux, several security mechanisms have been developed to secure the X server (*e.g.,* the X security extension [Wig96b], and several solutions at the level of the X protocol [DRU05, Wig96a]). The goal of the X11/SELinux project [KSV03, Wal07] is to add a reference monitor interface to the X server that can be used to enforce policies on how X clients access resources managed by the X server. This reference monitor is designed to interface to the security-enhanced Linux (SELinux) policy server [KSV03].

The central theme in both cases is to *retrofit legacy software with a reference monitor*. Other recent efforts with similar objectives include retrofitting the Java Virtual Machine [Fle06], the IBM Websphere software [HMS06, Shi07] and IBM DB2 [Shi07] with mechanisms to enforce SELinux authorization policies. In addition to these examples, a tremendous amount of other legacy code exists that likely requires similar retrofitting, ranging from server applications (*e.g.,* middleware [JDB, ODB], web servers [Apa, IIS], Samba [Sam], game systems [Pen], proxy and cache servers [SQU]), to client applications that manage multiple information flows (*e.g.,* email clients [HAM06], browsers and chat servers).

1.3 Current practice

In current practice, legacy software is manually retrofitted with a reference monitor. We illustrate the steps involved in this process using the example in Figure 1.1.

This example shows a fragment of code from an API function RequestAPI of a hypothetical server application. This API function accepts two arguments, a `client` and a `target`, and performs security-sensitive operations on an internal resource `obj`, that is derived from `target`. This is reminiscent of, for instance, a user requesting a security-sensitive operation (*e.g.,* Read, Write, Append) on a file via a system call—much as the system call internally translates the file into an inode before performing the operation on the actual data blocks representing the file, this API function translates `target` into `obj` by calling the function GetData. The lines with underlined comments must be retrofitted to ensure that security-sensitive operations are mediated by reference monitor calls. Retrofitting proceeds in four steps, as discussed below.

```
101 /* API function to process a request by client on target */
102 void RequestAPI (client, target) {
103     struct object obj;                  /* Denotes the resource to protect accesses to */
104     Others x, y, z, subjlabel;                                  /* Variable declarations */
105     ...
106     subjlabel = GetSubjLabel(client);              /** Find subject label for client */
107     obj = GetData(target);                                        /* Get obj from target */
108     obj.label = GetObjLabel(target);                /** Find object label for target */
109     x = obj.innocuous;                      /* Non-security-sensitive operation on obj */
110     if (AuthHook(subjlabel, obj.label)) {                 /** Reference monitor call */
111         y = obj.secret;            /* Security-sensitive operation to read secret data */
112         obj.integrity = z;             /* Security-sensitive operation to write data */
113     }
114     return;
115 }

201 /* Two reference monitor calls (CheckPolicy) in one authorization query */
202 int AuthHook (sublabel, objlabel) {
203     QueryResults q1, q2;                                        /* Variable declarations */
204 
205     q1 = CheckPolicy(sublabel, objlabel, SecrecyOp);             /** Query for line 111 */
206     q2 = CheckPolicy(sublabel, objlabel, IntegrityOp);           /** Query for line 112 */
207     return q1 && q2;
208 }
```

Figure 1.1 Retrofitting legacy software with a reference monitor interface. Lines with underlined comments must be retrofitted.

1. **Subject/Object identification and labeling.** Each subject requesting a security-sensitive operation and each object that may be affected by the security-sensitive operation must be identified and labeled with a security identifier. Subjects and objects are represented in the authorization policy using their labels, and the reference monitor uses these labels to determine whether a requested operation is permitted.

In current practice, the subjects and objects that are affected by a security-sensitive operation are determined manually. Their labels are bootstrapped using operating system support and are typically bound to the variables representing the subject and object, *e.g.,* stored as a field in the C `struct` representing the subject and object variables [KSV03, HMS06, HRJM07, Fle06]. For example, the SELinux operating system [LS01a, McC04] maintains security labels for each user and resource (*e.g.,* files, sockets) that it manages. Thus, the functions shown in lines 106 and 108, that fetch the subject and object label, respectively, rely on the operating system to supply labels. In the example in Figure 1.1, the object label is stored in the field `label` of `struct object`. Note that the application itself may create new objects, in which case it must also label these objects appropriately. In current practice, the function calls to determine subject and object labels (lines 106 and 108) are placed manually.

2. **Identifying security-sensitive operations.** Lines 109, 111 and 112 show accesses to members of the variable `obj` of type `struct object`. In this example, `struct object` denotes the type of a resource that is managed by the application. One or more of these accesses may represent a *security-sensitive operation*. Intuitively, a security-sensitive operation is a conceptual operation on a resource, such as a combination of structure-member accesses (*e.g.,* a set of structure-member accesses), that achieves a high-level objective. For example, a combination of accesses to the inode structure in the Linux kernel that performs a file read will be classified as a security-sensitive operation.

   Whether a combination of structure member accesses indeed represents a security-sensitive operation or not is determined by site-specific security requirements. For this example, we assume that the structure member accesses in lines 111 and 112 represent distinct security-sensitive operations (namely SecrecyOp and IntegrityOp), while the structure member access in line 109 is not representative of any security-sensitive operation.

   In current practice, security-sensitive operations are determined manually. Typically, a team of security analysts reasons about the kinds of security policies that must be enforced by the software, and determines a set of resources and security-sensitive operations. For instance, the LSM project identified 504 distinct security-sensitive operations (such as File_Read, File_Write, Dir_Mkdir, Dir_Rmdir) on different resources, such as files, directories, sockets and shared memory, that the Linux kernel (version 2.4.21) manages [LS01a, SVS01, Sma03]. Similarly, the X11/SELinux project identified 59 distinct security-sensitive operations (such as Window_Create, Window_Map) on resources such as windows, fonts and other resources that the X server manages [KSV03].

It is important to note that security-sensitive operations are currently identified using ad hoc reasoning, *e.g.,* by considering different security policies to be enforced, and *not* by analyzing source code. As a result, the relationship between the security-sensitive operations so identified and the code that implements them is *not* identified (*i.e.,* the combination of structure member accesses that represents a security-sensitive operation is not identified). Thus, identifying where a security-sensitive operation happens in source code is currently a manual and ad hoc process.

3. **Placing authorization queries.** Because the structure member access on line 111 represents the security-sensitive operation SecrecyOp, it must be mediated by the reference monitor call shown on line 205 (in this case, the keyword `SecrecyOp` shown on line 205 is a constant that denotes a security-sensitive operation). Similarly, the structure member access on line 112 must be mediated by the reference monitor call on line 206.

   Both these objectives are achieved by calling the function AuthHook on line 110 with the subject `client` and object `obj` that are involved in the security-sensitive operation. The implementation of AuthHook is provided as part of the retrofitting process. In this case, the function AuthHook consults the authorization policy (via the CheckPolicy function calls on lines 205 and 206) to check that both the security-sensitive operations SecrecyOp and IntegrityOp are allowed. They are thus either performed together, or are not performed at all. Finer-grained placement of authorization checks, that will allow individual checking of permissions for each of these security-sensitive operations is also possible, *e.g.,* by placing the function call on line 205 guarding line 111, and the function call on line 206 guarding line 112, respectively. In this case, the programmer likely combined the checks for both these security-sensitive operations because of one of three reasons:

- The site-specific policy demands that both SecrecyOp and IntegrityOp be performed together, or not at all. In this case, it can be argued that these security-sensitive operations instead be combined into a single security-sensitive operation, called SecrecyAndIntegrityOp, that is identified in code by a read of the `secret` field and a write of the `integrity` field of a variable of type `struct object`. The site-specific policy must also be rewritten to express policies for this security-sensitive operation.
- SecrecyOp and IntegrityOp are performed together at several locations in code, and the programmer combined the checks for these operations as an optimization (at the cost of a conservative authorization check).
- This was unintended, and the programmer instead meant to check for SecrecyOp and IntegrityOp separately.

In current practice, locating security-sensitive operations in source code, and placing authorization queries to mediate them is a manual and ad hoc process.

4. **Writing an authorization policy.** The final component of the retrofitting process is to write an appropriate authorization policy that satisfies site-specific security goals. Abstractly, an authorization policy is a set of triples ⟨Subject-label, Object-label, Operation⟩ that determines the set of security-sensitive operations that a subject with label Subject-label can perform on on an object with label Object-label. The CheckPolicy function calls on lines 205 and 206 consult the authorization policy and determine whether the requested operation should be permitted (and return a non-zero value if permitted).

   Because policies are determined by site-specific security goals, in current practice, they are written and maintained manually. Several off-the-shelf tools, such as the Tresys SELinux policy management toolkit [Trea, Treb], are now available to manage and interface with authorization policies.

As this example illustrates, retrofitting legacy code with a reference monitor is currently an ad hoc, manual process. Thus, it is both *time-consuming* and *error-prone*. For example, in spite of all the prior work put into several security patches for the Linux kernel [Arg, GRS, RSB, SEL, App, LID], it still took over two years to integrate the LSM framework into the mainline Linux kernel, and a number of bugs (*e.g.,* showing violation of the Principle of Complete Mediation [SS75]) were found and fixed along the way [JEZ04, ZEJ02]. Similarly, despite an initial implementation in 2003, the work of integrating a reference monitor interface into the X server is still ongoing. Part of this is due to changes in developers and the challenges of implementing a trusted path for the user (as outlined in several prior papers on secure windowing systems [BPWC90, Eps90, EMO+93, EP91, MPR06]), but recent work is still addressing fundamental issues, such as subject/object labeling and design of the query interface to the reference monitor [Wal07].

1.4 Contributions

This dissertation develops techniques to automate the hitherto ad hoc, manual process of retrofitting legacy software with a reference monitor. The thesis that this dissertation supports is the following:

Program analysis and transformation techniques offer a principled and automated way to retrofit legacy software with mechanisms for authorization policy enforcement

This dissertation supports the above thesis by making the following contributions:

1. **Fingerprints.** It introduces a new formalism, called *fingerprints*, to represent security-sensitive operations. A fingerprint represents a security-sensitive operation using a set of *code patterns* that represent how a resource must be accessed to perform that operation. Thus, a fingerprint implicitly embodies the relationship between a security-sensitive operation and the code that embodies this operation.

2. **Fingerprint mining algorithms.** It presents static and dynamic program analysis algorithms to automatically mine fingerprints for security-sensitive operations. The dynamic program analysis algorithm uses a novel *trace localization technique* that uses side-effects to localize

fingerprints in program traces. The static program analysis algorithm makes novel use of a hierarchical clustering technique called *concept analysis* to mine fingerprints.

Fingerprint mining algorithms remove the manual burden associated with Step (2) of the retrofitting process, discussed in Section 1.3.

3. **Fingerprint matching algorithms.** It presents a *fingerprint matching algorithm* that statically matches the fingerprint of a security-sensitive operation against source code and identifies all locations where the operation is performed. In conjunction with a program transformation tool that places authorization queries, this algorithm automatically retrofits legacy software with a reference monitor. It also presents heuristics to identify the subject and object involved in a security-sensitive operation.

   Fingerprint matching algorithms thus ameliorate the manual burden associated with Steps (1) and (3) of the retrofitting process from Section 1.3.

4. **Case studies on real-word software.** This dissertation presents case studies on three real-world software systems—the ext2 file system from Linux, the X server, and the PennMUSH multi-user dungeon [Pen]. The case studies on ext2 and X server directly compare the results of applying the algorithms developed in this dissertation against the results obtained by manually retrofitting these systems (as was done in the LSM and the X11/SELinux projects, respectively), while the case study on PennMUSH presents the applicability of these algorithms on a system that has as yet not been manually retrofitted with a reference monitor.

Techniques to help with Step (4) from Section 1.3, namely, writing authorization policies, are a subject of several past and ongoing research projects, and not considered in this dissertation.

1.5 Structure of this dissertation

This dissertation is organized as follows. Chapter 2 presents a high-level overview of our approach and states the assumptions underlying our work. Chapter 3 introduces fingerprints. Chapter 4 presents an algorithm to mine fingerprints from runtime execution traces of a program and shows its application to the X server. Chapter 5 presents a static program analysis-based fingerprint mining algorithm that overcomes several shortcomings of the dynamic program analysis-based algorithm. It also presents the application of this algorithm to the ext2 file system, the X server and PennMUSH. Chapter 6 presents an algorithm to match fingerprints against source code and place reference monitor checks. Chapter 7 presents related research and Chapter 8 concludes with directions for future work.

1.6 Bibliographic attributions

Most of the material presented in this dissertation has appeared in conference papers.

- Parts of Chapter 3 and Chapter 6 appeared in Proceedings of the 12^{th} ACM Conference on Computer and Communications Security (Alexandria, Virginia, November 2005) [GJJ05] as joint work with T. Jaeger and S. Jha. This paper introduced fingerprints, and presented an algorithm to match fingerprints against source code.

- Parts of Chapter 2, Chapter 4 and Chapter 6 appeared in Proceedings of the 27^{th} IEEE Symposium on Security and Privacy (Berkeley/Oakland, California, May 2006) [GJJ06] as joint work with T. Jaeger and S. Jha. This paper presented a dynamic fingerprint mining algorithm and applied it to the X server.

- Parts of Chapter 5 appeared in Proceedings of the 27^{th} International Conference on Software Engineering (Minneapolis, Minnesota, May 2007) [GKJJ07] as joint work with D. King, T. Jaeger and S. Jha. This paper presented a static fingerprint mining algorithm using concept analysis and applied it to the ext2 file system, the X server, and PennMUSH.

Chapter 2

Overview

This chapter discusses several assumptions upon which our approach is contingent, and presents the formal definition of a reference monitor. It then presents a high-level overview of our approach using the X server as an example. It shows how to retrofit the X server with mechanisms to enforce authorization policies on the security-sensitive operations requested by an X client that connects to the X server.

2.1 Goal

The main questions to be addressed when retrofitting a legacy server are *what are the security-sensitive operations to be mediated?*, *i.e.,* what are the primitive operations on critical server resources, and *where in the server's source code are these operations performed?* The idea is that once these locations are identified, authorization policy lookups can be added to the server code so as to completely mediate security-sensitive operations. The techniques developed in this dissertation assist with (1) the identification of resource accesses that constitute security-sensitive operations, (2) identification of locations in server code where these security-sensitive operations are performed, and (3) instrumentation of these locations, such that the operation is performed only if allowed by an authorization policy.

2.2 Assumptions

We assume the traditional client/server model, where the server manages resources on behalf of its clients. Clients connect to the server to request operations to be performed on these resources.

The server in turn must be equipped with mechanisms to mediate accesses to these resources and ensure that the requested operations are allowed only if they conform to an authorization policy. Our approach retrofits legacy servers with mechanisms for authorization policy enforcement. To ensure that our approach can securely enforce authorization policies, we make several assumptions about the server.

I: The server is not adversarial

We assume that the server itself is benign, *i.e.,* it is not written with adversarial intent, and does not actively try to defeat retroactive instrumentation. Thus, our approach assumes that the server does not remove or modify instrumentation. One way to ensure that a malicious user has not modified the server's code to defeat retroactive instrumentation is to have the operating system compare a hash of the server's executable against a precomputed value as it loads the server for execution. We also require that the server be non-self-modifying, to preclude the possibility that instrumentation is modified at runtime. One way to enforce this property is to make code pages write-protected.

II: The server can defend against control-hijacking exploits

Existing vulnerabilities, such as buffer-overflow vulnerabilities, could possibly be exploited by a malicious user to bypass our instrumentation. Because we cannot hope to eliminate these vulnerabilities statically, we assume that the server is protected using techniques such as CCured [NMW02], CFI [ABEL05] or other runtime execution monitoring and sandboxing techniques [FGH+04, FHSL96, LRB+05, SBBD01, WD01], which terminate execution when the behavior of the server differs from its expected behavior.

III: The server's running environment cooperates

The environment that the server runs in must cooperate with it to enforce authorization policies, and must not be malicious in intent. In particular, the server relies on the operating system to ensure that the authorization policy (stored on the file system) is tamper-proof. Moreover, because clients typically connect to the server via the operating system, the server relies on the operating system for tasks such as authentication and providing *security-labels* (*e.g.,* Top-Secret or Unclassified) associated with the clients.

IV: The server mediates all client communication

We assume that clients cannot communicate directly with each other, and that their communication is mediated by the server or the operating system. If client communication is mediated by the operating system, then the policy must be enforced by the operating system itself. Thus, we restrict ourselves to the case where communication is mediated by the server. We also note that if the clients communicate via the operating system, they cannot avail themselves of server-specific security-sensitive operations, such as cut and paste in the case of the X server. Thus our goal is to enforce authorization policies on server-specific security-sensitive operations requested by clients.

Finally, we assume that client-server communication is not altered by any intervening software layers. For example, most commercial deployments of the X server are accompanied by a *window manager*, (*e.g.,* `gnome` or `kde`). Because the window manager controls how clients connect to the X server, it can in theory, alter any information exchanged between the X server and its clients. However, because window managers are few in number (unlike X clients), we assume that they can be verified to satisfy the above assumption (though we have not done so). Further, the operating system can ensure that only certified window managers are allowed to run with the X server.

2.3 A note about the trusted computing base

The trusted computing base (TCB) [TCS85] of a computer system is defined as the set of all protection mechanisms, including hardware and software, that are needed to enforce a security policy. Researchers have historically advocated that TCB should be as small as possible to ensure that it is amenable to thorough verification and code audits. On most commercial systems, however, the TCB typically includes the hardware as well as the entire operating system.

The assumptions in Section 2.2 imply that in our approach the server to be retrofitted is also included in the TCB. This, unfortunately, is a drawback of our approach. The main reason that the application must be included in the TCB is to ensure that instrumentation added to enforce authorization policies is not bypassed. There are, however, several ways to reduce the size of the TCB.

One way to remove the retrofitted server from the TCB is to ensure protection against common vulnerabilities that can be exploited to bypass our instrumentation. While it would be unrealistic to assume that the server is vulnerability-free, additional protection mechanisms, *e.g.,* CCured or other sandboxing techniques, can ensure that the server is secure against most common control-hijacking exploits. In this case, it suffices to ensure that the operating system is in the TCB. The operating system bootstraps security by ensuring that the instrumentation inserted in the server is not tampered with. Clients need not be trusted, and could be malicious. Client security information, in particular, a client's security-label, is bootstrapped by the operating system during client connection, and is stored within the server. Clients thus cannot tamper with their security information after connection has been established.

Further reducing the size of the TCB is a topic for future investigation. For example, one approach is to leverage hardware support in modern commodity processors [MPP+07] to create a secure software stack via code attestation.

2.4 Basic tools

Our approach enforces authorization policies by retrofitting a server to ensure that security-sensitive operations requested by clients are mediated and approved by an authorization policy. The basic tools used to do so are a reference monitor and an enforcer [And72].

An authorization policy is defined as a set of triples $\langle sub, obj, op\rangle$, where each triple denotes that the subject *sub* is allowed to perform a security-sensitive operation *op* on an object *obj*. Subjects and objects are often associated with *security-labels*; for instance, all top-secret documents may have the security-label Top-Secret. Authorization policies are often represented using the security-labels of subjects and objects, rather than the subjects and objects themselves.

A reference monitor is defined as a quadruple $\langle\Sigma, \mathcal{S}, \mathcal{U}, \mathcal{R}\rangle$, and is parameterized by an authorization policy $\mathcal{A}$, where:

- Σ is a set of *security events*, where each security event is a triple $\langle sub, obj, op\rangle$;

- $\mathcal{S}$ is the *state* of the reference monitor, and is a set storing current associations of security-labels with subjects and objects;

- $\mathcal{U}$: $\Sigma \times \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is a *state update* function, which denotes how subject and object security-labels change in response to policy decisions;

- $\mathcal{R}$: $\Sigma \times \mathcal{S} \times \mathcal{A} \rightarrow \mathsf{Bool}$ is a *policy consulter*, which returns True if and only if a security event is permitted by the reference monitor.

An *enforcer* observes events in Σ generated in response to client requests, and passes them on to the reference monitor. Any violations of the policy, will result in $\mathcal{R}$ returning False, following which the enforcer will take appropriate action. Enforcing authorization policies entails implementing the enforcer and the reference monitor.

2.4.1 The enforcer

An implementation of the enforcer must satisfy two requirements:

1. It must monitor all security events generated in response to client requests. To do so, the enforcer must be able to infer the security-sensitive operation requested, the security-label of the subject that requests the operation (typically the client), and the object upon which the operation is to be performed.

2. It must take preventive action if a security event results in authorization failure. The action may be to terminate the client whose request resulted in the authorization failure. To do so, the enforcer must be able to control the execution of clients of the server, or audit the failure appropriately.

2.4.2 The reference monitor

An implementation of the reference monitor must ensure that the state of the reference monitor and the authorization policy are tamper-proof. In addition, the state of the reference monitor must be updated appropriately in response to security events, using $\mathcal{U}$. Implementing $\mathcal{R}$ entails looking up the policy, and can be achieved using off-the-shelf policy management libraries, such as the SELinux policy development toolkit [Trea, Treb].

2.5 Our approach

This section presents a high-level, informal overview of our approach, and describes how we implement the enforcer and the reference monitor. Algorithm and system details omitted from this section appear in subsequent chapters. We use a running example, the X server, to illustrate the approach.

2.5.1 An example: Retrofitting the X server

The X server accepts connections from multiple X clients, and manages resources (*e.g.,* windows, buffers) that it offers to these clients. Thus, it is important for the X server to enforce authorization policies on its X clients. A manual effort to retrofit the X server with authorization policy enforcement mechanisms was initiated by the NSA in early 2003 [KSV03], and a retrofitted version of the X server was released in 2005 [Sma05b] (though work on this project is still ongoing, as of March 2007 [Wal07]).

We demonstrate that our techniques can assist with, and potentially reduce the turnaround time of efforts to retrofit legacy servers, such as the X server. Specifically, with our approach, we were able to identify security-sensitive locations in the X server, and add reference monitoring code, with a few hours of manual effort. We ran the retrofitted X server on a security-enhanced operating system (SELinux [LS01a]), so that X clients have associated *security-labels*, such as Top-secret and Unclassified. The retrofitted X server enforced mandatory authorization policies on security-sensitive window operations requested by X clients based upon their security-labels.

Our approach proceeds in six steps, as shown in Figure 2.1.

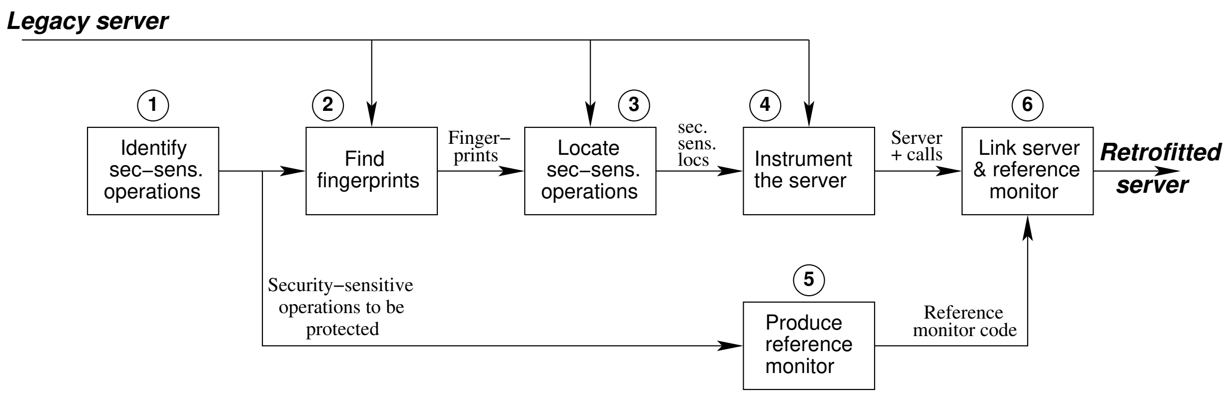


Figure 2.1 Steps involved in retrofitting a server for authorization policy enforcement.

2.5.2 Step 1: Find security-sensitive operations to be protected

The first step is to determine the security-sensitive operations to be protected. Typically, a design team considers security requirements for the server, and determines security-sensitive operations based upon these requirements. This approach was followed in the case of the LSM framework [WCS+02] and the X server [KSV03], where security-sensitive operations were identified for kernel resources, and X server resources, respectively. The design team typically considers a wide range of policies to be enforced by the server on resource accesses by clients. Because security-sensitive operations are typically the granularity at which authorization policies are written (a policy $\mathcal{A}$ is a set of triples of the form $\langle sub_i, obj_i, op_i \rangle$), the set of operations $\{op_i\}$ can be identified.

For the material presented in Chapter 4 (dynamic fingerprint mining), we assume that a description of security-sensitive operations is available. For instance, in the X server case study presented in Chapter 4, we used the set of security-sensitive operations that was identified manually by Kilpatrick *et al.* [KSV03]. This set of operations, 59 in number, considers security-sensitive operations on several key X server resources, including the `Client`, `Window`, `Font`, `Drawable`, `Input`, and `xEvent` data structures. Of these, 22 security-sensitive operations are for the `Window` data structure, such as Window_Create, Window_Map, and Window_Enumerate (we will denote security-sensitive operations in this dissertation using suggestive names, like the ones above). However, only an informal description of these security-sensitive operations is provided by Kilpatrick *et al.*, and a precise code-level description of these operations is needed for enforcement. Step 2 mines code-level descriptions of these operations; these code-level descriptions are called *fingerprints*.

However, a description of security-sensitive operations may not always be available, as for instance was the case with the PennMUSH [Pen] multi-user dungeon, one of the case studies considered in this dissertation. Indeed, it can be argued that identifying security-sensitive operations requires understanding the source code of the server being protected, which is a time-consuming exercise in itself. In such cases, the approach presented in Chapter 5 (static fingerprint mining) can be used.

This approach bypasses the need for a description of security-sensitive operations by directly analyzing the source code of the server and mining a set of resource accesses that describe how the server responds to client requests. In our experience, these resource accesses were also useful as code-level descriptions of security-sensitive operations (and are thus fingerprints themselves). Chapter 5 presents the details of a study where we correlated the fingerprints mined by the static approach against manually-identified security-sensitive operations for the ext2 file system and the X server. Thus, the static approach bypasses Step 1, and proceeds directly to Step 2.

2.5.3 Step 2: Find fingerprints of security-sensitive operations

The second step identifies fingerprints of security-sensitive operations. As described earlier, each security-sensitive operation is characterized by the set of resource accesses that are unique to the operation. These resource accesses are represented using code patterns (which are expressed as abstract syntax trees, or ASTs), and are the fingerprint of the security-sensitive operation (a formal definition of fingerprints appears in Chapter 3). The dynamic and static approach differ in their approach to fingerprint finding, as described next.

2.5.3.1 Dynamic fingerprint mining

The dynamic fingerprint mining approach assumes that a high-level description of security-sensitive operations is available. The code patterns that are associated with each security-sensitive operation are not known *a priori*, and the goal of the dynamic fingerprint mining algorithm is to recover this association.

Two novel observations help us achieve this goal. The first observation is that security-sensitive operations are typically associated with an observable change in the state of the system. For example, the security-sensitive operations Window_Create, Window_Map and Window_Enumerate of the X server are associated with opening, mapping, and enumerating child windows of an X client window, respectively (the changes visible on the screen when these operations happen are the observable changes associated with these operations). Thus, if we induce the server to perform a security-sensitive operation, and trace the server as we do so, the code patterns that form the fingerprint of the security-sensitive operation *must* be in the trace. For example, the function `CreateWindow`, which is implemented in the X server, is responsible for allocating memory and initializing a new window. We observed that creating a new window results in a call to this function. As a result, the *Call* `CreateWindow` was identified as a fingerprint for Window_Create. Note that the high-level descriptions of security-sensitive operations that are input to the dynamic mining algorithm are used to determine how security-sensitive operations can be induced in the system.

However, program traces are typically long, and it is still challenging to identify the code patterns that form the fingerprint of a security-sensitive operation from several thousand entries in a program trace. Our second observation addresses this challenge—to identify the fingerprint of a security-sensitive operation, it suffices to compare program traces that perform a security-sensitive operation against those that do not. For example, displaying a visible X client window (*e.g.,* `xterm`), which involves mapping the window on the screen, is associated with Window_Map; closing and typing to an `xterm` window are not. Thus, to identify the code patterns that characterize to Window_Map, it suffices to compare the trace generated by opening an `xterm` window against the trace generated by closing, or typing to the window. Similarly, closing a browser window is associated with closing all child windows, which involves Window_Enumerate, while typing to a window is not.

With these two observations, identifying fingerprints reduces to studying about 15 entries, on average, in a program trace. Using this technique, we identified, for example, the fingerprints of Window_Create as *Call* `CreateWindow`; of Window_Map as writes of `True` to the field `mapped` of a variable of type `Window` and `MapNotify` to the field `type` of a variable derived from type `xEvent`; and of Window_Enumerate as *Read* `WindowPtr->firstChild` and *Read* `WindowPtr->nextSib` and `WindowPtr ≠ 0`, which are intuitively performed during linked-list traversal. Note that code patterns are expressed at the granularity of reads and writes to individual fields of data structures. We discuss the tracing infrastructure, and algorithms to compare traces to identify fingerprints in more detail in Chapter 4.

2.5.3.2 Static fingerprint mining

The static fingerprint mining approach overcomes three important limitations of the dynamic approach. First, the dynamic approach requires an *a priori* description of security-sensitive operations. As described earlier, such descriptions may not always be available, as indeed was the case with PennMUSH. Second, the dynamic approach requires that an expert induce these security-sensitive operations and collect program traces; doing so may be tedious and error-prone. Third, because dynamic analysis only explores the code paths exercised by the manually-chosen inputs to the server, it will not examine the resource accesses in other portions of the server. As a result, the set of fingerprints identified will not be complete.

The static approach directly addresses these shortcomings of the dynamic approach. In particular, it makes novel use of a hierarchical clustering technique called concept analysis [Wil82]. The static approach is based upon the observation that a client can access server resources only via the server's API. For example, X clients can only access X server resources via the X protocol, which in turn invokes X server functions from a well-defined API. This approach identifies how data structures representing resources (*e.g.,* `Window`, `Font`, `xEvent`) are accessed via the API. It does so by distilling each statement of source code into a set of code patterns, and using concept analysis to cluster these code patterns based upon the API functions that they are accessed from. Each of these clusters is then output as a candidate fingerprint.

In our experiments on three real-world systems, namely, the ext2 file system, a subset of the X server, and PennMUSH, the static approach reduced the analysis of several thousand lines of code to the analysis of under 115 candidate fingerprints with fewer than 4 code patterns each (on average). For example, this approach reduced the analysis of PennMUSH, a server with $94,014$ lines of C code, to the analysis of 38 candidate fingerprints, with an average of 1.42 code patterns each. In the case of the ext2 file system and the X server, we were also able to correlate these candidate fingerprints with manually-identified security-sensitive operations (in the LSM project for ext2, and in the X11/SELinux project for X server). For example, in the analysis of the X server, one of the candidate fingerprints mined by the static approach was a write of the value `MapNotify` to the field `type` of variable derived from type `xEvent` and the value `True` to the field `mapped` of a variable of type `Window`. This fingerprint denotes key resource accesses performed when mapping a window to the screen, and is thus the fingerprint for the security-sensitive operation Window_Map. Recall that the same fingerprint was also identified for Window_Map by the dynamic approach.

The static approach addresses the shortcomings of the dynamic approach. Concept analysis mines candidate fingerprints without the need for an *a priori* description of security-sensitive operations or the need to manually induce the server to perform security-sensitive operations. Further, because static program analysis ensures better coverage than dynamic analysis, the static approach can mine more fingerprints than the dynamic approach.

2.5.4 Step 3: Find all locations that are security-sensitive

The third step uses the fingerprints identified in Step 2 to statically identify all locations in the server where code patterns that form the fingerprint of a security-sensitive operation occur. Each of these locations is said to perform the operation. Consider Figure 2.2, which shows a snippet of code from `MapSubWindows`, a function in the X server. It contains writes of `True` to `pWin->mapped`, and `MapNotify` to `event.u.u.type`, as well as a traversal of the children of the window pointer `pParent`. Thus, a call to the function `MapSubWindows` performs both the operations Window_Map

```
/* Implementation of the function MapSubWindows in the  X server.
   Several lines of code irrelevant to this example have been omitted */

MapSubWindows(Window *pParent, Client *pClient) {
    Window *pWin;
    xEvent event;

    ...
    pWin = pParent->firstChild;
    for (; pWin; pWin = pWin->nextSib) {
        pWin->mapped = TRUE;
        ...
        event.u.u.type = MapNotify;
        ...
    }
    ...
}
```

Figure 2.2 X server function MapSubWindows

and Window_Enumerate. We use a static fingerprint matching algorithm, described in Chapter 6, to determine the set of security-sensitive operations performed by each function.

In addition to identifying the locations where security-sensitive operations occur, in this step we also try to identify the subject and object associated with the operation. To do so, we identify the variables corresponding to subject and object data types (such as `Client` and `Window`) in scope. In most cases, this heuristic is good enough to identify the subject and the object. In Figure 2.2, the subject is the client requesting the operation (`pClient`), and the object for Window_Enumerate is the window whose children are enumerated (`pParent`), and the object for Window_Map is the variable denoting the child windows (`pWin`) that are mapped to the screen.

Step 2 and 3 together identify all locations where the server performs security-sensitive operations.

2.5.5 Step 4: Instrument the server

Having identified all locations where security-sensitive operations are performed, the server can be retrofitted by inserting calls to a reference monitor at these locations, to achieve complete mediation. In particular, if we determine that a statement **`Stmt`** is security-sensitive, and that it generates the security event ⟨*sub*, *obj*, *op*⟩, it is instrumented as shown below. Note that if **`Stmt`** is a call to a function `foo`, the query can instead be placed in the function-body of `foo`.

```
if (QueryRefmon(sub, obj, op) != TRUE) {
    HandleFailure;
}
else {
    Stmt;
}
```

For example, because the function `MapSubWindows` performs the security-sensitive operation Window_Enumerate (where children of `pParent` are enumerated) calls to `MapSubWindows` are protected as shown below.

```
if (QueryRefmon (pClient, pParent, Window_Enumerate) != TRUE) {
    HandleFailure;
}
else {
    MapSubWindows(pParent,pClient);
}
```

The statement `HandleFailure` can be used by the server to take suitable action against the offending client, either by terminating the client, or by auditing the failed request. Our approach currently does not automate the generation of failure-handling code—this must be manually written on a case-by-case basis. Developing an approach to gracefully handle failure in a principled way is an important topic for future research.

As mentioned earlier, authorization policies are expressed in terms of security-labels of subjects and objects. Security-labels can be stored in a table within the reference monitor, or instead, with data structures used by the server to represent subjects and objects. For example, in the X server, extra fields can be added to the `Client` and `Window` data structures to store security-labels. In either case, because we pass pointers to both the subject and the object to the reference monitor using `QueryRefmon`, the reference monitor can look up the corresponding security-labels, and consult the policy.

2.5.6 Step 5: Generate the reference monitor

This step generates code for the `QueryRefmon` function. We generate a template for this function, omitting two details that must be completed manually by a developer. First, the developer must specify how the policy is to be consulted, *i.e.,* he must implement $\mathcal{R}$ using an appropriate policy management API (*e.g.,* [Trea, Treb]). Second, he must implement the state update function, $\mathcal{U}$, by specifying how the state of the reference monitor is to be updated.

For example, when a security-event ⟨`pClient`, `pWin`, Window_Create⟩ succeeds, corresponding to creation of a new window, the security-label of `pWin`, the newly-created window, must be initialized appropriately. Similarly, a security-event that copies data from $\texttt{pWin}_1$ to $\texttt{pWin}_2$ may entail updating the security-label of $\texttt{pWin}_2$ (*e.g.,* under the Chinese-Wall policy [BN89]). Because security-labels are stored either as a table within the reference monitor or as fields of subject or object data structures as described earlier, the developer must modify these data structures appropriately to update security-labels. This step is described in further detail in Chapter 6.

Note that while Steps 2-4 are policy independent, Step 5 requires implementation of $\mathcal{R}$ and $\mathcal{U}$, which depend on the specific policy to be enforced.

2.5.7 Step 6: Link the modified server and reference monitor

The last step involves linking the retrofitted server and the reference monitor code to create an executable that can enforce authorization policies.

2.6 Discussion I: Security analysis

We now examine the security of our approach.

- **The enforcer** is implemented using instrumentation inserted in Step 4. Because the subject, object, and operation are passed to the reference monitor, security-labels can be retrieved, and the authorization policy consulted. If the requested operation is not permitted by the policy, the instrumentation ensures that it will not be executed. Further, because the server controls client connections, it can use `HandleFailure` to terminate the execution of malicious clients.

- **The reference monitor** is part of the server's address space, and is thus tamper-proof based upon our assumptions in Section 2.2. Alternately, the reference monitor can run as a separate process, and communicate with the server using IPC. The policy itself must be protected by storing it on the file-system with permissions such that it can be modified only by a privileged system user.

The security provided by our approach is thus contingent on whether calls to the reference monitor are placed so as to satisfy the Principle of Complete Mediation. Because reference monitor calls are placed by matching fingerprints, the security of our approach depends on the soundness and completeness of the fingerprint mining algorithms (Chapter 4 and Chapter 5) and the fingerprint matching algorithm (Chapter 6).

A noteworthy feature of our approach is its modularity. In particular, alternative implementations of fingerprint mining algorithms (*e.g.,* using program slicing techniques [AH90, KR97, ZG03]) and instrumentation (*e.g.,* using aspect weavers [AOS]) can be used in place of the algorithms developed in this dissertation. Thus, our technique benefits directly from improved algorithms for these tasks.

2.7 Discussion II: Why retrofit the server?

A question that may arise based upon the discussion of the technique so far is: "Why does the server itself have to be retrofitted to enforce authorization policies on its clients? In particular, why can't existing policy enforcement mechanisms in a security-enhanced operating system (*e.g.,* SELinux), upon which the server runs, be used to enforce these policies?"

The answer is that the server may provide channels of communication between clients that are not readily visible to the operating system. For example, consider enforcing a policy in the X server that disallows a cut operation from a Top-secret window followed by a paste operation into an Unclassified window. Cut and paste are X server-specific channels for X client communication. While these operations do have a kernel footprint, they are not as readily visible in the operating system as they are within the X server, where they are primitive operations. It is not advisable in such cases to use the operating system to enforce authorization policies, because it must be modified to be made aware of kernel footprints of X server-specific operations, which introduces application-specific code into the operating system. In addition, the X server must also be modified to expose more information to the operating system, such as internal data structures that will be affected by the requested operation. It has been argued that this is impractical [KSV03].

Chapter 3

Fingerprints

This chapter introduces fingerprints—the representation used by our approach for security-sensitive operations. As outlined in Chapter 2, fingerprints are matched against source code to locate security-sensitive operations and are thus central to our approach.

A server receives and processes client requests to access and modify the resources that it manages. Each such client request may perform one or more security-sensitive operations on the resource, each of which must be mediated by an authorization policy lookup. For example, an X client's request to map the child windows of a window `pParent` using a call to `MapSubWindows` (see Figure 2.2) results in the security-sensitive operations of enumerating the children of `pParent` (denoted by Window_Enumerate) and that of mapping each of the child windows onto the screen (denoted by Window_Map). Each such security-sensitive operation can have one or more *fingerprints*, where each fingerprint denotes the resource accesses needed to perform that security-sensitive operation.

3.1 Syntax

The syntax of a fingerprint of a security-sensitive operation OP is as defined in Figure 3.1. A fingerprint is a rule, where the left-hand-side of the rule is the name of a security-sensitive operation, OP, and the right hand side of the rule is a conjunction of one or more *code patterns* (also called *code templates*) or their negations.

A code pattern is a *Read*, *Write* or *Call* operation on a resource; the resource is represented in the code pattern as an abstract syntax tree (AST). In this dissertation, we restrict ourselves to the

F	::==	OP :- $(\bigwedge_{i=1}^{n}{}_{Intra}\ \mathrm{CP}_i)$ *Subject to* C
		$\mid \mathsf{OP}$:- $(\bigwedge_{i=1}^{n}{}_{Inter}\ \mathrm{CP}_i)$ *Subject to* C
CP	::==	C -P | ¬C -P
C	::==	*Same*(C -P $_i$, C -P $_j$)
		| *Different*(C -P $_i$, C -P $_j$)
C	::==	C ∧ C | True
C -P	::==	*Write* V *To* AST
		| *Read* AST
		| *Call* AST
		| *CallWith* AST (V , V , . . .)
		| B R (AST, AST)
		| B R (AST, V)
		| U R (AST)
AST	::==	(*type-name*->)$^+$*fieldname**
V	::==	⊥(unknown) | *constant*
		| *Binary Arithmetic Operator* (V , V)
B R	::==	≠ | ==
U R	::==	*Decrement* | *Increment*

Figure 3.1 BNF grammar defining fingerprints. The symbol OP in the definition of F denotes the name of a security-sensitive operation. Note that an abstract syntax tree (AST) representing a resource access is represented using data types, denoting the data type of the resource being accessed.

analysis of C source code, and assume that resources are represented as C structures (`structs`). The ASTs in code patterns thus represent accesses to fields of data types representing resources. A few simple extensions of the above operations are also included in the definition of a code pattern, *e.g., CallWith*, which denotes a *Call* with constraints on actual parameters, and simple binary and unary relations, such as tests for equality and disequality, and the increment and decrement operator.

We make the following observations on the syntax of fingerprints.

1. Resources are expressed in fingerprints using data types, rather than individual variable names. This decision was motivated by two considerations.

   First, specifying resources using data types makes a fingerprint easy to write. A security-sensitive operation can simply be expressed in terms of the data types of the resources that it manipulates. Each piece of code that matches this fingerprint (as described in Chapter 6) performs this fingerprint. Second, using data types for resources relieves the matching algorithm from having to use precise alias information. The matching algorithm instead abstracts each variable in the program to its data type before checking for a match. The matching algorithm is thus conservative—it may return spurious matches (*i.e.,* false positives), but will never miss a piece of code that matches a fingerprint (*i.e.,* false negatives). A false negative means that an authorization check is not placed where it should be, thus resulting in a potentially exploitable security hole.

2. Even though resources are expressed in terms of data types, we have found that in some cases further constraints on code patterns in a fingerprint improves the precision of matching. The syntax of fingerprints allows *Same* and *Different* constraints that can be used to restrict the code fragments that match a fingerprint. For example, in the fingerprint for Window_Enumerate shown below, the *Different* constraint mentions that the variable of type WindowPtr that matches the first code pattern must be different from the variable that matches the second code pattern.

Window_Enumerate :-	*Read* $\texttt{WindowPtr}_1$`->firstChild`
	$\wedge$ *Read* $\texttt{WindowPtr}_2$`->nextSib`
	$\wedge$ `WindowPtr` $\neq$ `0` *Subject to*
	Different($\texttt{WindowPtr}_1$, $\texttt{WindowPtr}_2$)

Note, however, that to avoid false negatives, precise alias information must be used if such constraints are used in fingerprints. For example, *Same*($\texttt{WindowPtr}_1$, $\texttt{WindowPtr}_2$) is a constraint that restricts $\texttt{WindowPtr}_1$ and $\texttt{WindowPtr}_2$ to be the same, *i.e.,* point to the same resource. Alias information is needed to resolve this constraint precisely.

3. Fingerprints can either be specified as *intraprocedural* or *interprocedural* (represented in Figure 3.1 using $\wedge_{Intra}$ and $\wedge_{Inter}$, respectively). The matching algorithm described in Chapter 6 matches intraprocedural fingerprints against code contained in a procedure, while it matches interprocedural fingerprints against code contained in all procedures in the program being analyzed. In most cases in our experiments, however, we found that intraprocedural fingerprints were sufficient to represent security-sensitive operations.

4. Finally, we note that temporal ordering information cannot be expressed using the syntax of fingerprints shown in Figure 3.1. Thus, we cannot express fingerprints to represent the rule "Read `WindowPtr->firstChild` before reading `WindowPtr->nextSib`". While the simpler fingerprint language results in simpler, more intuitive fingerprint mining algorithms, the inability to express temporal ordering information is a limitation, which can potentially result in false positives in the output of the matching algorithm.

   However, in our experiments (described in Chapter 6), we found that the number of false positives was manageable. For example, in our analysis of the X server, we found that one source of false positives was the Window_Enumerate fingerprint. The fingerprint for this security-sensitive operation only approximates linked-list traversal, and thus triggers spurious matches. In particular, out of 20 locations that were output by the matching algorithm as performing Window_Enumerate, only 10 did.

   Extending the fingerprint language to include temporal information is a topic for future work. Existing tools, such as MOPS [CW02] and `xgcc` [HCXE02], already employ algorithms to match such temporal patterns and our work can benefit directly from these tools. Note, however, that mining temporal fingerprints requires designing more sophisticated algorithms than those developed in this dissertation.

3.2 Interpretation

The fingerprint of a security-sensitive operation OP represents the resource accesses needed to perform that operation. Thus, the effect that a security-sensitive operation has on a resource is defined by the resource accesses in its fingerprint.

Using the above interpretation, however, the fingerprint of a security-sensitive operation OP must contain all the resource accesses needed to perform that operation on the resource. For example, one fingerprint for the operation Window_Map of the X server, which represents mapping a window onto the screen, is *Call* `MapWindow`, or alternatively, the set of statements that implements `MapWindow`. Expressing fingerprints with *all* the resource accesses needed to perform an operation is certainly acceptable, but makes the fingerprint ineffective in the presence of even minor changes to the source code of the server.

As a result, we typically express fingerprints using only the set of resource accesses that suffice to differentiate one security-sensitive operation from another. For example, we found that the following fingerprint suffices to precisely identify all locations in the X server that perform Window_Map:

Window_Map :-	*Write* `True` *To* `WindowPtr->mapped` ∧
	Write `MapNotify` *To* `xEvent->union->type`

The problem of expressing fingerprints at an appropriate granularity that precisely represents security-sensitive operations is referred to by Erlingsson as *security event synthesis* [Erl04, Pages 73–82]. This is the subject of the next two chapters, which develop *mining* algorithms to extract fingerprints by analyzing source code.

Chapter 4

Mining Fingerprints using Dynamic Program Analysis

This chapter presents the use of dynamic program analysis to mine fingerprints of security-sensitive operations. In particular, it presents a technique that makes novel use of tangible side-effects to locate fingerprints in runtime execution traces of a program. It also presents an application of this technique to mine fingerprints of security-sensitive operations for the X server.

4.1 Problem statement

Given a server program, and a description of the security-sensitive operations that clients can request the server to perform, the technique presented in this chapter outputs fingerprints of these security-sensitive operations.

This chapter thus assumes that the set of security-sensitive operations is known *a priori*. The description can be informal, and must describe the high-level intent of the security-sensitive operation. For example, the document by Kilpatrick *et al.* [KSV03] contains such descriptions for 59 security-sensitive operations on different resources managed by the X server, such as `Windows`, `Fonts` and `Colors`. This document describes, for instance, the Window_Map security-sensitive operation as the action of mapping a window to the screen and the Window_Enumerate operation as the action of listing child windows.

Because a fingerprint characterizes the resource accesses performed by a security-sensitive operation, the technique presented here offers a way to formalize an informal description of security-sensitive operations. This technique is motivated by current practice in retrofitting code, where the aforementioned informal descriptions are also used to locate where code performs these security-sensitive operations (see, for example, [KSV03, Sections 5.2 and 5.3]).

4.2 Identifying fingerprints using analysis of program traces

How can informal descriptions of security-sensitive operations be converted into precise code-level descriptions (*i.e.,* fingerprints) of these operations? We present two novel observations that enable us to do so.

Observation 1 (Tangible side-effects) Security-sensitive operations are typically associated with observable change in the system state—these changes will be refered to as *tangible side-effects*.

Tangible side-effects help us determine whether a server has performed a security-sensitive operation. Thus, if we induce the server to perform a security-sensitive operation, *i.e.,* the occurrence of a tangible side-effect denotes that the operation is performed, then the resource accesses associated with that security-sensitive operation *must* be in the trace generated by the server. Thus, identifying fingerprints reduces to tracing the server as it performs a tangible side-effect, and recording accesses to resources as it does so. Each trace records function calls, and reads and writes to resources as well as the functions in which they were performed. In particular, each trace records resource accesses using the *Write*, *Read* and *Call* code patterns, shown in Figure 3.1.

However, the program trace generated by the server, even in a controlled experiment to perform a tangible side-effect, may be huge. For example, using our tracing infrastructure, the X server generates a trace of length 10459 when the following experiment is performed: start the X server, open an `xterm`, close the `xterm`, and close the X server (each of these is a tangible side-effect). It is infeasible to identify succinct fingerprints of security-sensitive operations (*e.g.,* those of Window_Create and Window_Destroy) by studying this trace. Our second observation addresses this problem.

Observation 2 (Comparing traces) The fingerprint of a security-sensitive operation can be localized by comparing traces generated by server executions that perform a security-sensitive operation against traces generated by executions that do not.

The key idea underlying this observation is that if an execution of the server does not perform a security-sensitive operation, then the trace produced by the server will not contain a fingerprint of that operation. For example, the trace T_{open} that opens an X client window on the X server will contain the fingerprint of Window_Create, but the trace T_{close} that closes a window will not. Thus, T_{open} - T_{close}, a shorter trace, still contains the fingerprint of Window_Create. Continuing this process with other traces that do not perform Window_Create reduces the size of the trace to be examined even further. In fact, for the X server we were able to reduce the size of the trace several-fold using this technique (see Table 4.1 and Table 4.2), whittling down the search for fingerprints to about 15 functions, on average.

A technical difficulty must be addressed before we compare traces. A tangible side-effect may be associated with multiple security-sensitive operations, and all the security-sensitive operations associated with it must be identified. For instance, when an `xterm` window is opened on the X server, the security-sensitive operations include (amongst others) creating a window (Window_Create), mapping it to the screen (Window_Map), and initializing several window attributes (Window_Setattr).

We manually identify the security-sensitive operations associated with each tangible side-effect. Because the side-effects we consider are *tangible*, programmers typically have an intuitive understanding of the operations involved in performing the side-effect. The trace generated by the tangible side-effect is then assigned a *label* with the set of security-sensitive operations that it performs. It is important to note that tangible side-effects are not specific to the X server alone, and are applicable to other servers as well. For example, in a database server, dropping or adding a record, changing fields of records, and performing table joins are tangible side-effects. Because labeling traces is a manual process, it is conceivable that they are not labeled correctly. However we show empirically that fingerprints can be identified succinctly and precisely, *in spite of errors in labeling*. Because each trace can be associated with multiple security-sensitive operations, we formulate *set equations* for each operation in terms of the labels of our traces.

Definition 4.1 (Set equation) Given set S, a set $B \subseteq S$, and a collection C={C_1, C_2, …, C_n} of subsets of S, a set equation for B is $B{=}C_{j_1} {*} C_{j_2} {*} \ldots {*} C_{j_k}$, where each C_{j_i} is an element, or the complement of an element of C, and '*' is ∪ or ∩.

To find a fingerprint for an operation OP, we do the following: Let S be the set of all security-sensitive operations, and B = {OP}. Let C_i denote the label (*i.e.,* the set of security sensitive operations performed) of trace T_i, which is obtained when the server performs the tangible side-effect seff_i. Formulate a set equation for B in terms of C_i's, and apply the *same set-operations* on the set of code patterns in the corresponding T_i's. The resulting set of code patterns is the fingerprint for OP.

For example, if T_1 is a trace that performs OP and OP′, and T_2 is a trace that performs OP′, then C_1 = {OP, OP′}, and C_2 = {OP′}. Say T_1 contains the set of code patterns {p_1, p_2}, and T_2 contains the set of code patterns {p_2}. Then to find the fingerprint of OP, we let B = {OP}, and observe that $B = C_1 - C_2$. We perform the *same* set-operations on the set of code patterns in T_1 and T_2 to obtain {p_1}, which is then reported as the fingerprint of OP. This process is formalized in Algorithm 1.

Finding a set equation for a set B is equivalent to computing an *exact cover* for this set. An exact cover may not always exist; if one exists, it can be computed efficiently. However, because each trace is manually labeled with the set of security-sensitive operations that it performs (using tangible side-effects to aid reasoning), these labels may potentially be erroneous. We would thus like to compute the smallest set equation for the set B.

Finding the smallest set equations is, in general, a hard problem. More precisely, define a CNF set equation as a set equation expressed in conjunctive normal form, with '∩' and '∪' as the conjunction and disjunction operators, respectively. Each disjunct in the equation is a *clause*. The k-CNF SET EQUATION problem, which is equivalent to the problem of finding the smallest set equations, can be shown to be NP-complete.

Definition 4.2 (k-CNF SET EQUATION) Given a set S, a set $B \subseteq S$, a collection C of subsets of S (as in Definition 4.1), and an integer k, does B have a CNF-set equation with at most k clauses?

Algorithm: F _F ($\mathcal{X}$, S, Seff)

Input : (i) $\mathcal{X}$: Server to be retrofitted,

(ii) S: A set of security-sensitive operations $\{\mathsf{OP}_1, \ldots, \mathsf{OP}_n\}$, and

(iii) *Seff*: A set of tangible side-effects $\{\mathit{seff}_1, \ldots, \mathit{seff}_m\}$.

Output : $\mathrm{FP}_1, \ldots, \mathrm{FP}_n$: Each FP_i is the fingerprint of the security-sensitive operation OP_i.

$\mathcal{X}' := \mathcal{X}$ instrumented to perform tracing;

foreach (tangible side-effect $\mathit{seff}_i \in \mathit{Seff}$) **do**

- T_i := Trace generated by $\mathcal{X}'$ when induced to perform seff_i;
- $\mathit{label}(\mathrm{T}_i)$:= Set of operations (from S) involved in seff_i;

foreach ($\mathsf{OP}_i \in S$) **do**

- SE_i := Set-equation for OP_i in terms of $\mathit{label}(\mathrm{T}_1), \ldots, \mathit{label}(\mathrm{T}_m)$;
- CPset_i := Set of code patterns in T_i;
- FP_i := Result when the set operations in SE_i are performed on $\mathrm{CPset}_1, \ldots, \mathrm{CPset}_m$;

return $\mathrm{FP}_1, \ldots, \mathrm{FP}_n$

Algorithm 1: Dynamic program analysis-based algorithm to mine fingerprints of security-sensitive operations.

We currently use a simple brute-force algorithm to find set equations. This works for us, because the number of sets we have to examine (which is the number of traces we gather) is fortunately quite small (15 for the X server).

4.3 Implementation

We have implemented Algorithm 1 in a prototype tool called A . We use a modified version of `gcc` to compile the server. During compilation, instrumentation is inserted statically at statements that read and write to fields of data structures denoting resources that we want to protect access to. We log the field and the data structure that was read from, or written to, and the function name, file

Trace name	A	B	C	D	E	F	G	H	I
Side-effect $\longrightarrow$ **Security-sensitive Operation** $\downarrow$	open xterm	close xterm	open browser	close browser	type to window	move window	open & close twm menu	switch windows	open menu (browser)
Window_Create	✓		✓				✓		✓
Window_Destroy		✓	$\times_{fn}$	✓			✓	$\times_{fn}$	
Window_Map	✓		✓				✓		✓
Window_Unmap		✓	$\times_{fn}$	✓			✓	$\times_{fn}$	
Window_Chstack	✓		✓				✓	✓	✓
Window_Getattr	✓		✓			$\times_{fp}$	$\times_{fp}$		✓
Window_Setattr	✓		✓			✓	$\times_{fp}$	$\times_{fn}$	✓
Window_Move			$\times_{fn}$			✓		$\times_{fn}$	$\times_{fn}$
Window_Enumerate	$\times_{fn}$	$\times_{fn}$	✓	✓		✓	$\times_{fn}$	✓	✓
Window_InputEvent					✓	✓	✓	✓	✓
Window_DrawEvent	✓	✓	✓	✓	$\times_{fn}$	✓	✓	✓	✓
Distinct Functions	115	148	251	161	68	148	96	93	166

Table 4.1 Examples of labeled traces obtained from the X server. A "✓" entry in (*row*, *column*) denotes that the trace represented by *column* performs the security-sensitive operation represented by *row*. A "$\times_{fn}$" or a "$\times_{fp}$" entry denotes a mistake in manual labeling.

name, and the line number at which this occurs. We then induce the modified server to perform a set of tangible side-effects, and proceed as in Algorithm 1 to mine fingerprints.

We applied this to mine fingerprints of security-sensitive operations in the X server. In particular, we recorded reads and writes to fields of data structures such as `Client`, `Window`, `Font`, `Drawable`, `Input`, and `xEvent`. Table 4.1 shows the result of performing lines (1)-(4) of Algorithm 1. Columns represent traces of 9 tangible side-effects, and rows represent 11 security-sensitive operations on the `Window` data structure. We manually labeled each trace with the security-sensitive operations that it performs. These entries are marked in Table 4.1 using ✓ and

X_{fp}. For example, opening an `xterm` on the X server includes creating a window (Window_Create), mapping it onto the screen (Window_Map), placing it appropriately in the stack of windows that X server maintains (Window_Chstack), getting and setting its attributes (Window_Getattr, Window_Setattr), and drawing the contents of the window (Window_DrawEvent). This trace of operations contains 115 calls to distinct functions in the X server, as shown in the last row of Table 4.1.

Table 4.2 and Figure 4.1 show the result of performing lines (5)-(8) of Algorithm 1 with the labeled traces obtained above. For each operation, the set equation used to obtain fingerprints and the size of the resulting set are shown in Table 4.2, while the set of fingerprints is shown in Figure 4.1. Note that each security-sensitive operation can have more than one fingerprint, as for example, is the case with Window_Enumerate and Window_InputEvent.

Operation	**Set Equation**	**|FP|**
Window_Create	∩(A, C, G) - D - H	9
Window_Destroy	∩(B, D) - A	7
Window_Map	∩(A, C, G) - D - H	9
Window_Unmap	∩(B, D) - A	7
Window_Chstack	∩(A, C, G, H, I) - D - E	6
Window_Getattr	∩(A, C, I) - B - D - E - F	25
Window_Setattr	∩(A, C, F, I) - B - D - E	15
Window_Move	F - A - B - D - E - G	38
Window_Enumerate	∩(C, D, F, H, I)	21
Window_InputEvent	E - C	19
Window_DrawEvent	∩(A, B, C, D, E, F, G, H, I)	12
	Average value of |FP|:	**15.3**

Table 4.2 Set equations for security-sensitive operations computed using the annotations in Table 4.1, and the sizes of the resulting sets.

To find errors in manual labeling of traces, we did the following. After finding fingerprints of security-sensitive operations, we checked each trace for the presence of these fingerprints. Presence of a fingerprint of a security-sensitive operation in a trace that is not labeled with that security-sensitive operation shows an error in manual labeling; such entries are marked X_{fn} in Table 4.1. For example, we did not label the trace generated by opening a browser (`htmlview`) with Window_Unmap. On the other hand, absence of fingerprints of a security-sensitive operation in a trace that is labeled with the security-sensitive operation also shows an error in manual labeling; such entries are marked X_{fp} in Table 4.1. Thus for example, we did label the trace generated by moving a window with Window_Getattr, whereas in fact, this operation is not performed when a window is moved.

4.4 Evaluation of the dynamic fingerprint mining algorithm

We now evaluate the dynamic fingerprint mining algorithm, as implemented in A , by answering four questions.

4.4.1 How effective is Aid at locating fingerprints?

Raw traces generated by tangible-side effects have, on average, 53829 code patterns. However, A abstracts each trace to the granularity of functions: it first identifies fingerprints at the function level; if necessary, it delves into the code patterns exercised by the function. The number of distinct functions called in each trace is shown in the last row of Table 4.1. The third column of Table 4.2 shows, in terms of the number of functions, the size of FP, which is the result obtained by computing the set equation for each security-sensitive operation, to determine fingerprints. A was able to achieve about one order of magnitude reduction in terms of the number of distinct functions to be examined for fingerprints.

We examined each of the functions in FP to determine if it is indeed a fingerprint. In most cases, we found that for a security-sensitive operation, a single function in FP performs the operation. However, in some cases, multiple functions in FP seemed to perform the security-sensitive operation. For example, both *Call* `MapWindow` and *Call* `MapSubWindow`, which were present in

Window_Create :-	*Call* `CreateWindow`
Window_Destroy :-	*Call* `DeleteWindow`
Window_Map :-	*Write* True *To* `Window->mapped` $\wedge$ *Write* `MapNotify` *To* `xEvent->union->type`
Window_Unmap :-	*Write* `UnmapNotify` *To* `xEvent->union->type`
Window_Chstack :-	*Call* `MoveWindowInStack`
Window_Getattr :-	*Call* `GetWindowAttributes`
Window_Setattr :-	*Call* `ChangeWindowAttributes`
Window_Move	*Call* `ProcTranslateCoords`
Window_Enumerate :-	*Read* `WindowPtr->firstChild` $\wedge$ *Read* `WindowPtr->nextSib` $\wedge$ `WindowPtr`$\neq$`0`
Window_Enumerate :-	*Read* `WindowPtr->lastChild` $\wedge$ *Read* `WindowPtr->prevSib`
Window_InputEvent :-	*Call* `CoreProcessPointerEvent`
Window_InputEvent :-	*Call* `CoreProcessKeyboardEvent`
Window_InputEvent :-	*Call* `xf86eqProcessInputEvents`
Window_DrawEvent :-	*Call* `DeliverEventsToWindow`

Figure 4.1 Fingerprints obtained by analyzing set equations in Figure 4.1 by applying Algorithm 1 to the labeled traces from Table 4.1.

FP, performed Window_Map. In such cases, we examined the execution traces of the server to determine common code patterns exercised by the call to these functions. Doing so for Window_Map reveals that the common code patterns in `MapWindow` and `MapSubWindow` are (*Write* True *to* `Window->mapped` $\wedge$ *Write* `MapNotify` *to* `xEvent->union->type`). For security-sensitive operations such as Window_InputEvent, where we did not find common code patterns exercised by candidate functions from FP, we deemed each of these function calls to be fingerprints of the operation.

4.4.2 How precise are the fingerprints found?

For each of the fingerprints recovered by A for the X server, we manually verified that it is indeed a fingerprint of the security-sensitive operation in question.

However, in general, A need not recover all fingerprints of a security-sensitive operation. Because A employs dynamic program analysis, it can only capture the fingerprints of a security-sensitive operation exercised by the runtime traces, and may miss *other* ways to perform the operation. By collecting traces for a larger number of tangible side-effects, and verifying the fingerprints collected by A against these traces, confidence can be increased in the precision of fingerprints obtained by A .

4.4.3 How much effort is involved in manual labeling of traces?

In all, we collected 15 traces for different tangible side-effects exercising different `Window`-related security-sensitive operations. It took us a few hours to manually label traces with security-sensitive operations.

4.4.4 How effective is manual labeling of traces?

In most cases, it is easy to reason about the security-sensitive operations that are performed if a tangible side-effect is induced. However, because this process is manual, we may miss security-sensitive operations that may be performed ($\times_{fn}$ entries in Table 4.1), or erroneously label a trace with security-sensitive operations that are not actually performed ($\times_{fp}$ entries). Our experience of manually labeling traces for the X server shows that this process has an error rate of approximately 15%.

However, it must be noted that we were able to recover fingerprints *in spite of labeling errors*. If a security-sensitive operation is wrongly omitted from the labels of a trace that performs a tangible side-effect associated with that operation (the $\times_{fn}$ case), then because the same security-sensitive operation often appears in the labels of other traces, a set equation can still be formulated for the operation, and the fingerprint can be recovered. On the other hand, if a security-sensitive operation is wrongly added to the labels of a trace (the $\times_{fp}$ case), none of the functions in FP will perform the tangible side-effect. In this case, trace labels are refined, and the process is iterated until a fingerprint is identified.

4.5 Limitations

While the technique presented in this chapter was motivated by the need to formalize descriptions of security-sensitive operations, it has three limitations.

1. The technique relies on the availability of high-level descriptions of security-sensitive operations. While such descriptions are available for several Linux subsystems and the X server, identifying security-sensitive operations is, in general, an ad hoc and manual exercise, and it is not realistic to assume that such descriptions will be available for all servers.

2. The technique relies on tangible side-effects to determine whether a security-sensitive operation was performed during an execution of a server. This is problematic for two reasons.

   First, it may not always be possible to identify tangible side-effects for a security-sensitive operation. Indeed, in our experiments with the X server reported in this chapter, we faced difficulties in identifying tangible side-effects for several security-sensitive operations.

   Second, the technique relies on a human to manually perform controlled experiments on the server, collect the runtime execution traces so generated, and reason about the security-sensitive operations associated with each trace. The accuracy and effectiveness of the technique is thus dependent on how carefully the experiments were performed. For example, the X server permits transparent windows, which can be mapped to the screen (thus inducing Window_Map). If such a transparent window is mapped to the screen during one of the experiments, and the human fails to include Window_Map in the label of the associated execution trace, the technique will fail to compute a fingerprint for Window_Map.

3. Finally, because the technique uses dynamic program analysis, it cannot guarantee that all fingerprints of a security-sensitive operation have been found, *i.e.,* the technique is not *complete*, and can have false negatives. In particular, the number and quality of fingerprints

mined by the technique directly relies on the code coverage of the manually-induced experiments.

These limitations are fundamental, and prevented this technique from being applied to a wide variety of servers. The static fingerprint mining technique presented in the next chapter directly addresses these limitations of the dynamic mining technique.

4.6 Using the dyamic fingerprint mining tool

This section summarizes the steps that a security analyst must follow to find fingerprints using the dynamic fingerprint mining tool.

- Design a set of experiments to induce security-sensitive operations in the server. Label each experiment with the set of security-sensitive operations performed in that experiment.
- Instrument the server to log accesses to sensitive data structures, conduct the experiments from the first step, and collect the traces emitted by the server.
- Compute set equations for each security-sensitive operation.
- Apply set equations to traces and obtain pruned sets of code-patterns.
- Manually examine and refine the pruned sets to identify fingerprints of security-sensitive operations.

4.7 Summary of key ideas

To summarize, the key contributions of this chapter are:

- The use of dynamic program analysis to convert high-level, informal descriptions of security-sensitive operations into fingerprints.
- The use of tangible side-effects as a means to determine whether an execution of a server performs a security-sensitive operation.

- An algorithm to localize fingerprints by comparing execution traces using set equations.
- An implementation of the above techniques in a prototype tool called A , and its evaluation on the X server.

Chapter 5

Mining Fingerprints using Static Program Analysis

This chapter develops a technique that uses static program analysis and a clustering technique, called concept analysis [Wil82], to mine fingerprints of security-sensitive operations. This technique directly addresses the main shortcomings of the dynamic program analysis technique presented in the previous chapter. In particular, the technique mines fingerprints without the need for an *a priori* description of security-sensitive operations. Further, because static program analysis ensures better coverage than dynamic analysis, the technique presented in this chapter can mine more fingerprints than the technique in the previous chapter. This chapter also presents three case studies, showing the application of this technique to mine fingerprints of security-sensitive operations for the Linux ext2 file system, the X server, and PennMUSH, a multi-user dungeon.

5.1 Problem statement

Given a server program, and the data types of resources, accesses to which must be protected, the technique presented in this chapter outputs *building blocks* that satisfy Property 1 defined below. These building blocks can be used to construct fingerprints [1].

Property 1 (Happens together) A building block BB output by the technique presented in this chapter is a set of code patterns $BB = \{pat_1, \ldots, pat_m\}$ that satisfies the following property: if one

[1]For the rest of this chapter, we will relax the strict syntactic definition of a fingerprint (as in Figure 3.1), and refer to a fingerprint as a set of code patterns instead. The interpretation of a fingerprint remains unchanged with this syntax: the set of code patterns in a fingerprint represents all the resource accesses needed to perform the security-sensitive operation represented by that fingerprint.

of the code patterns pat_i in BB appears in any valid execution trace of the server, then *all* the code patterns in BB appear in that trace.

Thus, the building blocks satisfying Property 1 represent resource accesses that are always performed together. Our hypothesis is that building blocks satisfying Property 1 can be used to construct fingerprints of security-sensitive operations. Intuitively, this is because such a set denotes the set of resource accesses that must be performed to achieve a high-level operation on the resource (*e.g.,* an operation such as Window_Map or Window_Enumerate). Indeed, in our experiments with the Linux ext2 file system and the X server, we found that the building blocks output by the technique were excellent indicators of security-sensitive operations identified manually (and independently, in the LSM project [WCS+02] and the X11/SELinux project [KSV03], respectively).

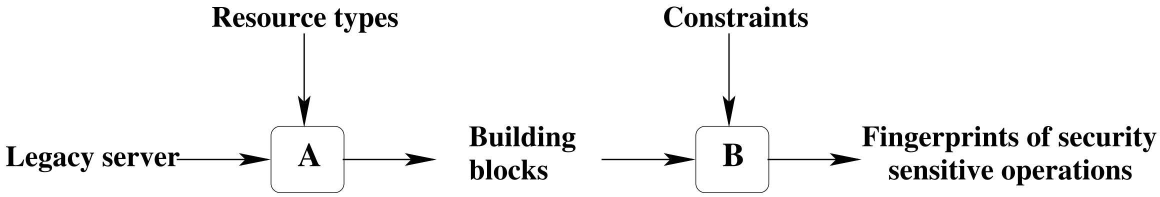
5.2 Identifying fingerprints using static program analysis and concept analysis

This section presents a high-level overview of our technique. Using a running example, it demonstrates how a software engineer would use this technique to mine fingerprints of security-sensitive operations. The entire process is depicted in Table 5.1.

5.2.1 Running example

We use a subset of ext2, a Linux file system, and one of the case studies in Section 5.5 as our running example. In particular, ext2 is responsible for laying out and interpreting disk blocks as belonging to specific files or directories. It represents metadata information using several internal data structures. This metadata is used to retrieve files and directories from raw disk blocks.

File systems on Linux are pluggable, and must thus export a standard API to the kernel. A system call that manipulates files or directories ultimately resolves to one or more calls to this API. The relevant file system functions then serve this request. Thus a file system is a server that manages files and directories. For ext2, we considered 10 API functions related to manipulation of directories (*e.g.,* `ext2_rmdir`, `ext2_mkdir` and `ext2_readdir`). We show how our technique can identify security-sensitive operations that ext2 performs on directories.



Step	Description	Techniques used
A	Extraction of building blocks from source code.	Static analysis and concept analysis.
B	Refinement of building blocks.	Application of constraints and interpretation of fingerprints.

Table 5.1 Steps to statically mine fingerprints of security-sensitive operations, and the techniques used in each step. Static analysis is first used in conjunction with concept analysis to extract building blocks. These building blocks are refined/composed to yield fingerprints.

5.2.2 Step A: From source code to building blocks

In the first step, we employ static source code analysis and identify different ways in which ext2 accesses shared resources in response to client requests.

To do so, we must first identify resources, accesses to which must be authorized. As before, we express the resources to be protected using their data types. For ext2, these resources include internal data structures used to represent files and directories. These data structures are specified by a domain expert, and for ext2 they are variables of type `inode`, `ext2_dirent`, `ext2_dir_entry_2` and `address_space`, each of which is a C `struct`.

We also assume that a client accesses server resources only via the server's API. With ext2, this is indeed the case, and as mentioned earlier ext2 exports a well-defined API to the kernel. The inputs to our static analyzer are thus the source code of ext2, and two files, specifying, respectively, the types of resource data structures, accesses to which must be authorized, and a set of API functions.

The static analyzer identifies how these resource data structures are manipulated by the ext2 API. It does so by distilling each statement of ext2 source code into a (possibly empty) set of code patterns. Code patterns are as defined in Figure 3.1, and include *Read*s, *Write*s and *Call*s. For example, the C statement `de->file_type = 0`, where `de` is a variable of type `ext2_dirent` is distilled to *Write* `0` *To* `ext2_dirent->file_type`. Note in particular that this transformation ignores specific variable names and focuses instead on types of variables. As a result, we identify generic resource manipulations but not the specific instance of the resource (*e.g.,* the instance `de`) that they happen on.

Statements that do not manipulate resource data structures are ignored. *Call* code patterns correspond to calls via unresolved function pointers. For each function `ext2_api` in the ext2 API, the static analyzer then aggregates code patterns of all statements potentially reachable via a call to `ext2_api`. Thus, at the end of this step each ext2 API function `ext2_api` is associated with a set of code patterns *CodePats*(`ext2_api`). Intuitively, *CodePats*(`ext2_api`) denotes all possible ways in which `ext2_api` can potentially manipulate tracked resources.

The next step is to find sets of code patterns that always appear together during server execution. That is, if one code pattern from a set of code patterns appears in an execution of ext2, then all the other code patterns from that set appear in that execution as well (*i.e.,* a set of code patterns satisfying Property 1). Note that we can have sets {*pat*} with singleton code patterns as well, denoting that no other code pattern always appears together with {*pat*}. Each set of such code patterns denotes an idiomatic way in which a resource is manipulated by ext2, and potentially indicates a security-sensitive operation. Each such set is called a *building block*.

We identify building blocks using concept analysis [Wil82], a well-known hierarchical clustering technique. At a high-level (details are presented in Section 5.3), concept analysis identifies building blocks, as well as the API functions whose code pattern sets contain these building blocks.

For example, concept analysis inferred that the set of six code patterns shown in Figure 5.1 is a building block, and that it appears in *CodePats*(`ext2_rename`), *CodePats*(`ext2_rmdir`) and *CodePats*(`ext2_unlink`).

(1)	*Read* `address_space->host`
(2)	*Read* `ext2_dir_entry_2->rec_len`
(3)	*Write* `0` *To* `ext2_dir_entry_2->inode`
(4)	*Read* `inode->i_mtime`
(5)	*Read* `inode->u->ext2_inode_info->i_dir_start_lookup`
(6)	*Write* ⊥ *To* `inode->u->ext2_inode_info->i_dir_start_lookup`

Figure 5.1 One of the building blocks that concept analysis identifies for ext2.

For ext2, we identified 18 such building blocks, each denoting a unique way in which ext2 manipulates files and directories. While concept analysis is asymptotically inefficient—its complexity is exponential in $\max_i(|CodePats(\texttt{ext2\_api}_i)|)$—our experiments showed that it is efficient in practice. In particular, our analysis completed in about 2 seconds for ext2, and in just over 310 seconds for the largest of our case studies.

5.2.3 Step B: Refining building blocks

In the second step, a domain expert (i) refines building blocks obtained from Step A and (ii) post refinement, determines, for each fingerprint, whether it embodies a security-sensitive operation that must be mediated by an authorization policy lookup.

Refinement of building blocks is necessary for two reasons.

- The first reason is because the code analysis employed in Step A is imprecise. As a result, a set of code patterns that appears in the results of concept analysis may not satisfy Property 1. There are two ways in which precision is lost:

1. The static analysis algorithm employed in Step A is *flow-insensitive*. A building block may contain a pair of code patterns pat_1, pat_2 that do not always appear together in all executions of the server (thus violating Property 1).

2. We ignore specific instances of resources that are manipulated and focus instead on their types. Thus, a building block may contain manipulations of multiple, possibly unrelated, resources.

We employ *precision constraints* to identify such cases and enable refinement of each building block, separating the code patterns that it contains into several fingerprints. Intuitively, a precision constraint is a rule that determines the set of code patterns that can be grouped together in a fingerprint.

- The second reason why refinement is necessary is because a domain expert may deem that a set of code patterns is irrelevant for the authorization policies to be enforced for the server, or may wish to separate or group together a pair of code patterns in a fingerprint of a security-sensitive operation. Such *domain-specific constraints* further refine building blocks.

For example, consider the building block shown in Figure 5.1. Using the output of our static analysis tool, we were able to determine that the code patterns (1)-(4) appear together in each successful invocation of the ext2 function `ext2_delete_entry` and that the code patterns (5) and (6) appear together in each successful invocation of the function `ext2_find_entry`. Each of the three API functions, `ext2_rename`, `ext2_rmdir` and `ext2_unlink`, that contain this building block call both these functions. Both `ext2_rmdir` and `ext2_unlink` call these functions on the *same* resource instance, namely the directory being removed (or unlinked). However, as Figure 5.2 shows, while `ext2_rename` calls both these functions on the instances `old_dir` and `old_dentry`,[2] it calls `ext2_find_entry` only on the instances `new_dir` and `new_dentry` when a certain predicate `new_inode` is satisfied.

[2]The variable `old_de`, which `ext2_delete_entry` is invoked with on line 18 is derived from `old_dir` and `old_dentry`.

```
int ext2_rename (inode *old_dir, dentry *old_dentry,
                 inode *new_dir, dentry *new_dentry) {
    /* Declarations of old_page, new_page, old_de and new_de */
    new_inode = new_dentry->d_inode;
    ...
    old_de = ext2_find_entry (old_dir, old_dentry, &old_page);
    if (new_inode) {
        ...
        new_de = ext2_find_entry(new_dir, new_dentry, &new_page);
        ...
    }
    else {
        ...
        /* No call to ext2_find_entry in this branch */
        ...
    };
    ...
    ext2_delete_entry(old_de, old_page);
    ...
}
```

Figure 5.2 Example showing the need for precision constraints.

Because `ext2_rename` performs the resource manipulations corresponding to code patterns (5) and (6) on additional resource instances as compared to the code patterns (1)-(4), code patterns (1)-(4) and (5)-(6) likely represent different security-sensitive operations. Imposing the constraint that code patterns on different resource instances must be part of separate fingerprints, the building block shown in Figure 5.1 is split into two parts, as shown in Figure 5.3. Additional examples of the use of precision constraints appear in Section 5.4. Note that such constraints can potentially be avoided with sophisticated program analyses, which we plan to explore in future work. However, in our case studies we found that more than 50% of the building blocks did not require refinement.

Thus our current approach provides a good tradeoff between precision of results and simplicity of the code analysis algorithm.

Domain-specific constraints encode rules that are formulated by a domain-expert. In particular, whether the resource manipulation embodied by a fingerprint is security-sensitive depends on the set of policies that must be enforced on clients. For example, it may only be necessary to protect the integrity of directories, and not their confidentiality. In this case, fingerprints that embody a write operation on directories are security-sensitive, while fingerprints that embody a read operation are not. Fingerprints expose possible operations on resources, and let an administrator decide whether an operation is security-sensitive or not. For example, an analyst may decide that Fingerprint (2) in Figure 5.3, which corresponds to a directory lookup, is not interesting for a specific set of policies to be enforced.

Fingerprint (1)	
(1)	*Read* `address_space->host`
(2)	*Read* `ext2_dir_entry_2->rec_len`
(3)	*Write* `0` *To* `ext2_dir_entry_2->inode`
(4)	*Read* `inode->i_mtime`

Fingerprint (2)	
(5)	*Read* `inode->u->ext2_inode_info->i_dir_start_lookup`
(6)	*Write* $\perp$ *To* `inode->u->ext2_inode_info->i_dir_start_lookup`

Figure 5.3 Fingerprints obtained after refinement with precision constraints.

After refinement, the domain expert assigns semantics to each fingerprint, associating it with a security-sensitive operation. For example, Fingerprint (1) in Figure 5.3 embodies the directory removal operation, while Fingerprint (2) embodies the lookup operation. The LSM project [WCS+02] has identified a comprehensive set of security-sensitive operations for Linux by considering a wide range of policies to be enforced, including security-sensitive operations on the file system. It turns out that Fingerprint (1) embodies the LSM operation Dir_Remove_Name, while Fingerprint (2) embodies the LSM operation Dir_Search. Thus, at the end of the second step, we have a set of fingerprints, each of which is associated with a security-sensitive operation.

5.3 Extracting building blocks from code

This section discusses Step A in detail. We discuss the use of static analysis to identify resource manipulations potentially performed by each API function, and concept analysis to find building blocks.

5.3.1 Static analysis

Algorithm 2 describes the static code analysis that we have implemented (in CIL [NMRW02]). Lines 1-5 employ a simple flow-insensitive analysis to extract for each function a set of code patterns describing how the function manipulates resource data structures. While this step sacrifices precision, it simplifies the rest of the analysis by making the output amenable to concept analysis. As described earlier, we recover some of the precision lost in this step by applying precision constraints. While we intend to explore in future work how a flow-sensitive program analysis can interact with concept analysis, we have found that our current implementation offers a reasonable tradeoff between simplicity of analysis and precision of the results obtained. Lines 6-9 compute *CodePats*($\texttt{api}_i$), the set of resource accesses performed by $\texttt{api}_i$, for each API function $\texttt{api}_i$ of the server by finding functions in the call-graph reachable from $\texttt{api}_i$. We resolve calls through function pointers using a simple pointer analysis: each function pointer can resolve to any function whose address is taken and whose type signature matches that of the function pointer. This analysis is conservative in the absence of type-casts, but may miss potential targets in the presence of type-casts.

Recall that *CodePats*($\texttt{api}_i$) is the set of resource accesses that a client can perform by invoking API function $\texttt{api}_i$. However, we would like to identify resource accesses satisfying Property 1, *i.e.,* we would like to identify sets $BB = \{pat_1, \ldots, pat_m\}$ such that if one of the code patterns $pat_i \in BB$ appears in any valid execution trace of the server, then *all* the patterns in BB appear in that trace.

Note that Property 1 implies that each building block BB is such that either $FP \subseteq$ *CodePats*(`api`$_i$) or $BB \cap$ *CodePats*(`api`$_i$) $= \emptyset$, for each API function `api`$_i$. As described below, we use concept analysis to identify a set of building blocks. Each building block may then be used to construct fingerprints.

Algorithm: E _C -P (Server, API, RSC)
Input : (i) Server: source code of server,
(ii) API={`api`$_1$, . . ., `api`$_n$}: set of API functions of Server, and
(iii) RSC: data types of sensitive resources.
Output : *CodePats*(`api`$_1$), . . ., *CodePats*(`api`$_n$), for `api`$_1$, . . ., `api`$_n$ $\in$ API.

foreach (function `f` in Server) **do**
  Summary(`f`) := $\emptyset$;
  foreach (statement $s \in$ `f` that affects a data structure of type $\in$ RSC) **do**
    CP := Decomposition of s into code patterns (see C -P in Figure 3.1);
    Summary(`f`) := Summary(`f`) $\cup$ CP;
foreach (`api`$_i$ $\in$ API) **do**
  CodePats(`api`$_i$) := $\emptyset$;
  foreach (function `f` reachable from `api`$_i$) **do**
    CodePats(`api`$_i$) := *CodePats*(`api`$_i$) $\cup$ Summary(`f`);
return *CodePats*(`api`$_1$), . . ., *CodePats*(`api`$_n$)

Algorithm 2: Static analysis algorithm to extract resource manipulations.

5.3.2 Background on concept analysis

Concept analysis is a well-known hierarchical clustering technique that has found use in software engineering (*e.g.,* for aspect mining [CMM+05, EKS03, TC04, TM04], to identify modular structure in legacy code [LS97, Sif98, ST98, vDK99], to automatically convert non-object-oriented programs into object oriented ones [Sif98], and to debug automatically mined temporal specifications [AMBL03]). We give a brief overview of concept analysis and describe how we adapt it to find building blocks.

The inputs to concept analysis are (i) a set of *instances* I, (ii) a set of *features* F, and (iii) a binary relation $R : I \rightarrow F$ that associates instances with features. It produces a *concept lattice* as output. Intuitively, each node in the concept lattice pairs a set of instances X with a set of features Y, such that Y is the largest set of features in common to *all of the instances* in X. Formally, each node is a pair $\langle X, Y \rangle$, where $X \in I$ and $Y \in F$, such that $\alpha(X)$=Y and $\gamma(Y)$=X, where $\alpha(X) = \{f \in F | \forall x \in X\ (x, f) \in R\}$, and $\gamma(Y) = \{i \in I | \forall y \in Y\ (i, y) \in R\}$. A node $\langle X, Y \rangle$ appears as an ancestor of a node $\langle P, Q \rangle$ in the concept lattice if $P \subset X$. In fact, this ordering also implies $Y \subset Q$. This is because a smaller set of instances will share a larger set of features in common. Thus, the root node shows the set of features common to all instances in I, while the leaf node shows the set of instances that share all features in F.

Figure 5.4 shows an example of a concept lattice, as applied to our problem. Each API function $\texttt{api}_1$, $\texttt{api}_2$, $\texttt{api}_3$ and $\texttt{api}_4$ is considered an instance, and each code pattern pat_1, pat_2, pat_3, pat_4 is considered a feature. They are related by *CodePats*, which is obtained from static analysis, depicted in Figure 5.4(a) as a table. Each node $\langle X, Y \rangle$ is such that *all* the code patterns in Y appears in each *CodePats*($\texttt{api}_i$) for $\texttt{api}_i \in X$. This lattice shows, for example, that (i) there are no code patterns in common to all API functions (node A in the lattice), (ii) Both pat_1 and pat_3 appear in both *CodePats*($\texttt{api}_2$) and *CodePats*($\texttt{api}_3$), and these are the only such API functions (node D), and that (iii) No API functions have all code patterns (node G).

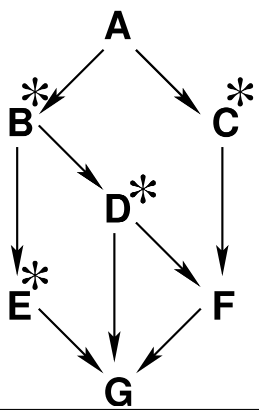
5.3.3 Using concept analysis

We compute building blocks using Algorithm 3. It first invokes concept analysis (line 1) on the set of API functions and the set of code patterns to obtain a concept lattice as shown in Figure 5.4. It then finds building blocks, in lines 2-9, by finding nodes in the lattice where new code patterns are introduced. Each such node is marked, and the set of new code patterns introduced in that node is considered as a building block.

(a) The relation *CodePats*

CodePats	pat_1	pat_2	pat_3	pat_4
$\texttt{api}_1$	✓	✓		
$\texttt{api}_2$	✓	✓	✓	
$\texttt{api}_3$	✓		✓	✓
$\texttt{api}_4$				✓

(b) Concept lattice



(c) Nodes in the concept lattice

A : $\langle\{\texttt{api}_1, \texttt{api}_2, \texttt{api}_3, \texttt{api}_4\}, \emptyset\rangle$

B : $\langle\{\texttt{api}_1, \texttt{api}_2, \texttt{api}_3\}, \{pat_1\}\rangle$

C : $\langle\{\texttt{api}_3, \texttt{api}_4\}, \{pat_4\}\rangle$

D : $\langle\{\texttt{api}_2, \texttt{api}_3\}, \{pat_1, pat_3\}\rangle$

E : $\langle\{\texttt{api}_1, \texttt{api}_2\}, \{pat_1, pat_2\}\rangle$

F : $\langle\{\texttt{api}_3\}, \{pat_1, pat_3, pat_4\}\rangle$

G : $\langle\emptyset, \{pat_1, pat_2, pat_3, pat_4\}\rangle$

Figure 5.4 Concept analysis example.

For the example in Figure 5.4, the nodes B, C, D, and E are marked because these nodes introduce the code patterns pat_1, pat_4, pat_3 and pat_2—*i.e.,* any node containing one of these patterns *must* have the corresponding node as an ancestor. Each of these code patterns is classified as a building block.

Intuitively, Algorithm 3 works because each building block BB satisfies $BB \subseteq CodePats(\texttt{api}_i)$ or $BB \cap CodePats(\texttt{api}_i) = \emptyset$, for each API function $\texttt{api}_i$. Concept analysis ensures that the node of the concept lattice in which a new code pattern $pat_i \in BB$ is introduced will introduce *all* of the code patterns in BB. Line 9 identifies and marks nodes where a new code pattern *pat* is introduced into the lattice. Because of the property above, all the code patterns that appear in the same building block as *pat* appear in that node. Note however, that code patterns in each building block may not satisfy Property 1 (because static analysis was flow-insensitive). Thus the building blocks computed by Algorithm 3 must be refined (in Step B).

Algorithm: F _B _B (*CodePats*, API)

Input : (i) *CodePats*: The relation obtained from Algorithm 2, and

(ii) API= {$\texttt{api}_1, \ldots, \texttt{api}_n$}, set of API functions of the server.

Output : $CFP_1, \ldots, CFP_k$, a set of building blocks.

1 Run concept analysis with the set of instances I=API, the set of features $F = \bigcup_{i \in [1..n]} CodePats(\texttt{api}_i)$, and the relation R=*CodePats*;

2 $count := 1$;

3 **foreach** (node $\langle X, Y \rangle$ in the concept lattice) **do**

4   Let $\{\langle X_j, Y_j \rangle\}$ be the set of parents of $\langle X, Y \rangle$ in the concept lattice;

5   Diff := $Y - \bigcup_j Y_j$;

6   **if** (Diff $\neq \emptyset$) **then**

7   CFP_{count} := Diff;

8         $count := count + 1$;

9   Mark the node $\langle X, Y \rangle$;

10 **return** $CFP_1, \ldots, CFP_{count}$ /* Note: k is the value of *count* in this line. */

Algorithm 3: Algorithm for finding building blocks.

The number of building blocks identified by Algorithm 3 has an upper bound of $|\cup_{i \in [1..n]} CodePats(\texttt{api}_i)|$. Note that while the concept lattice can be exponentially large in the number of API functions (because asymptotically, it is a lattice on the power set of API functions), this upper bound places a restriction on the number of nodes that will be marked in line 9 of Algorithm 3. This is key, because these nodes introduce building blocks, and as discussed in Section 5.2, they must be manually examined for refinement in Step B.

Several algorithms have been proposed in the literature to compute concept lattices. We chose to implement the incremental algorithm by Godin *et al.* [GMA95, Algorithm 1] because it has been shown to work well in practice [AMBL03]. While this algorithm is asymptotically exponential—its complexity is $O(2^{2^p}|I|)$, where p is an upper bound on the number of features of any instance in I—the algorithm scaled well in our case studies.

5.4 Refinement with constraints

As described in Section 5.2.2, building blocks obtained from concept analysis are imprecise for two reasons. First, because of flow-insensitivity, a pair of code patterns pat_1 and pat_2 that do not satisfy Property 1 may appear in the same building block. Second, the resource manipulations in a building block may be associated with multiple, possibly unrelated resource instances. Thus, building blocks must be refined using precision constraints. Domain-specific constraints can additionally be applied to refine constraints with domain-specific requirements.

This section presents a unified framework to express constraints and refine building blocks (Step B). Both precision constraints and domain-specific constraints can be expressed in this framework.

As Figure 5.5 shows, each constraint is either a *Separate*(X, Y), an *Ignore*(X) or a *Combine*(X, Y), where X and Y are sets of code patterns. *Separate*(X, Y) refines building blocks by separating code pattern sets X and Y into separate fingerprints. *Ignore*(X) refines building blocks by discarding the code pattern set X from building blocks. *Combine*(X, Y), for which we have only felt occasional need, combines code pattern sets X and Y in two building blocks into a single fingerprint, thus coarsening the results of concept analysis. For example, the constraint *Separate*({1, 2, 3, 4}, {5, 6}) refines the building block in Figure 5.1 to yield the fingerprints in Figure 5.3. We now discuss precision and domain-specific constraints in this framework.

Precision constraints are *Separate*(X, Y) constraints and as discussed in Section 5.2, they serve two goals. The first goal is to refine building blocks based upon resource instances manipulated. *Separate*({1, 2, 3, 4}, {5, 6}), the use of which was illustrated earlier, serves this goal. Formally, each set of code patterns can be associated with one or more resource instances that it manipulates. We use a constraint *Separate(X, Y)* to separate code pattern sets X and Y that manipulate different sets of resource instances. For example, consider the code patterns (1)-(4) in Figure 5.1,

C	::==	*Separate* (P S , P S)
		| *Combine* (P S , P S)
		| *Ignore* (P S)
P S	::==	Set of code patterns (see C -P in Figure 3.1)

Figure 5.5 BNF grammar for constraints.

that appear in the function `ext2_delete_entry`, and the code patterns (5) and (6), that appear in the function `ext2_find_entry`. Because of the way these functions are invoked in `ext2_rename` (see Figure 5.2), code patterns (5) and (6) are associated with the resource instances `old_dir`, `old_dentry`, `new_dir` and `new_dentry`, while code patterns (1)-(4) are associated with resource instances `old_dir` and `old_dentry`. Because the code patterns (5) and (6) are applied to additional resource instances, they are separated out using the constraint above. We currently manually identify resource instances associated with a set of code patterns. However, this can potentially be automated using a program analysis that is sensitive to resource instances manipulated.

The second goal of precision constraints is to identify and remove imprecision introduced because of flow-insensitive program analysis. In particular, a pair of code patterns pat_1 and pat_2 may appear together in a building block, but may not appear together in all executions of the server. In such cases, a *Separate*(pat_1, pat_2) constraint separates these code patterns into different fingerprints. For example, one of the building blocks that we obtained in the analysis of ext2 is shown below; it appeared in *CodePats*(`ext2_ioctl`).

(1)	*Write* ⊥ *To* `inode->i_flags`
(2)	*Write* ⊥ *To* `inode->i_generation`

However, `ext2_ioctl` either performs the resource manipulation corresponding to code pattern (1) or (2), but not both, in each execution, based upon the value of a flag that it is invoked with. Thus, a constraint *Separate*({*1*}, {*2*}) is used to refine the building block above.

Note that precision constraints are not necessary if more precise program analysis is employed. Algorithm 2 currently lacks flow-sensitivity and data-flow information that can potentially avoid the imprecision reported above. However, in each of our case studies we needed precision constraints for no more than 50% of the building blocks mined—9/18 for ext2, 24/115 for X server, and 4/38 for PennMUSH. Thus, we believe that our current technique strikes a good balance between simplicity and precision of building blocks.

Domain-specific constraints encode domain knowledge to further refine building blocks. A domain specific constraint that we have found useful is *Ignore*(*Pat*), using which we can eliminate certain code patterns that we deem irrelevant for security. For example, in the X server, which is an event-based server, each request from an X client is converted into a one or more events that are processed by the server. It may only be necessary to enforce an authorization policy governing the set of events that an X client can request on a resource. In such cases, all code patterns except those related to event-processing can be filtered out from fingerprints using *Ignore* constraints.

The use of *Combine* constraints is relatively infrequent, and may be used if the building blocks mined by concept analysis are at too fine a granularity. For example, in PennMUSH, we found that 30 of the 38 building blocks contained only one code pattern. An administrator may wish to write authorization policies at a higher level of granularity—where the fingerprint of each security-sensitive operation contains multiple code patterns. *Combine* constraints can be used to group together code patterns to get such fingerprints.

		Analysis	Concept lattice				Refinement
Benchmark	**LOC**	**time (secs)**	**# Nodes**	**# Edges**	**Number**	**Size**	**needed for**
ext2	4,476	2.1	21	32	18	3.67	9 (50%)
X server/`dix`	30,096	58.1	329	978	115	3.76	24 (20.87%)
PennMUSH	94,014	318.9	127	301	38	1.42	4 (10.53%)

Table 5.2 Results for each of our case studies. The sixth column denotes the number of building blocks mined, while the seventh column shows their average size, in terms of the number of code patterns per building block. The table also shows the number of building blocks that had to be refined with precision constraints. Figure 5.6, Figure 5.7 and Figure 5.8 depict the concept lattices produced for each of these case studies.

5.5 Evaluation of the static fingerprint mining algorithm

We conducted case studies on three complex systems, each of which has been in development for several years. We used (i) the ext2 file system from Linux kernel distribution 2.4.21, (ii) a subset of the X server (X11R6.8), and (iii) PennMUSH, an online game server ($v1.8.1p9$).

We evaluated our technique using four criteria.

- First, we measured the number and average size of building blocks extracted from source code. Because an analyst must examine these building blocks to identify security-sensitive operations, these metrics indicate the amount of manual effort needed to supplement our technique. Note that without our technique, the analyst must examine the *entire* code base to find security-sensitive operations.

- Second, we measured the number of building blocks that had to be refined with constraints. This metric shows the effect of imprecise static analysis and the effort needed to refine building blocks.

- Third, we evaluated the quality of fingerprints by manually interpreting the operation embodied by each fingerprint.

- Last, for ext2 and the X server, we correlated the fingerprints extracted by our technique with security-sensitive operations that were identified independently for these servers [KSV03, WCS+02].

Table 5.2 presents statistics on the time taken by the analysis and the size of concept lattices produced. It also shows the number and size of building blocks and the number of building blocks that needed refinement. As these results show, our analysis is effective at distilling several thousand lines of code into concept lattices of manageable size. None of our benchmarks had more than 115 building blocks. These building blocks were, on average, smaller than 4 code patterns, and fewer than 50% of these had to be refined manually. Identifying security-sensitive operations reduces to refining and interpreting these building blocks, instead of having to analyze several thousand lines of code, thus drastically cutting the manual effort required. In our case studies, this required a few hours, with modest domain knowledge. As Table 5.2 also shows, our analysis is efficient in practice, completing in just over 310 seconds even for PennMUSH, our largest benchmark (on a 1GHz AMD Athlon processor with 1GB RAM). Sections 5.5.1-5.5.3 present each case study in detail, including our experience interpreting fingerprints and correlating these fingerprints against independently identified security-sensitive operations.

5.5.1 The ext2 file system

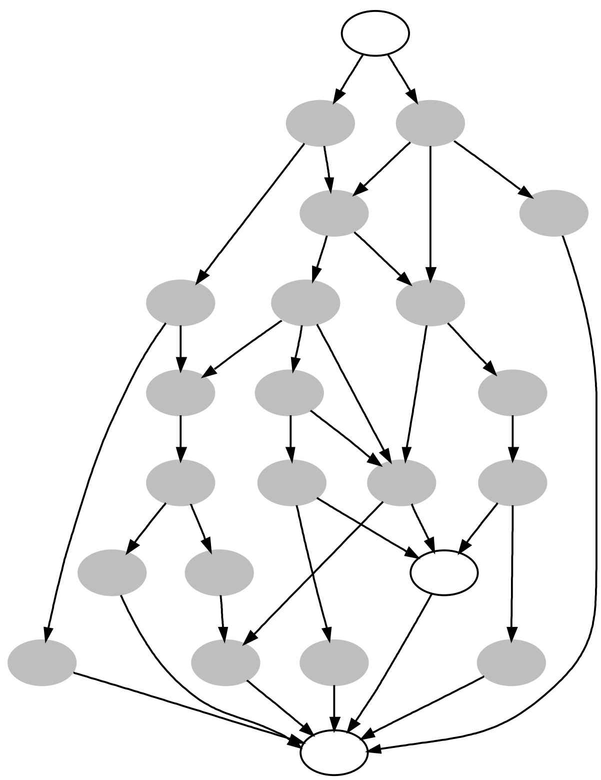


Figure 5.6 Concept lattice for ext2. The shaded nodes represent those marked by Algorithm 3, and the concepts represented by these node contain building blocks. This concept lattice has 21 nodes and 32 edges. Algorithm 3 identified 18 building blocks.

As discussed in Section 5.2, we focused on how directories are manipulated by the ext2 file system. Concept analysis produced the concept lattice shown in Figure 5.6. The shaded nodes in this lattice depict 18 building blocks containing an average of 3.67 code patterns, of which we had to refine 9 with precision constraints to obtain a total of 44 fingerprints. We then determined the resource manipulation embodied by each fingerprint and tried to associate it with a security-sensitive operation. Section 5.2 presented two such examples. Two more examples are discussed below.

1. The fingerprint {*Write* `0` *To* `inode->i_blocks`, *Write* `4096` *To* `inode->i_blksize`, *Write* `1` *To* `inode->u->ext2_inode_info->i_new_inode`} appears in *CodePats*(`ext2_create`), *CodePats*(`ext2_mkdir`), *CodePats*(`ext2_mknod`) and *CodePats*(`ext2_symlink`). The code patterns in this fingerprint were all extracted from the function called `ext2_new_inode` and embody creation and initialization of a new `inode`.

2. The fingerprint {*Write* `0` *To* `inode->i_size`} appears in *CodePats*(`ext2_rmdir`). This code pattern embodies a key step in directory removal.

The LSM project has identified a set of 11 operations on directories. These operations are used to write SELinux policies governing how processes can manipulate directories. We were able to identify at least one fingerprint for each of these LSM operations from the fingerprints that we mined. For example, the fingerprints presented in Section 5.2 were for the LSM operations Dir_Remove_Name and Dir_Search, while the examples above correspond to the File_Create[3] and Dir_Rmdir operations, respectively.

5.5.2 The X11 server

The X server is a popular window-management server. X clients can connect to the X server, which manages resources such as windows and fonts on behalf of these X clients. The X server has historically lacked mechanisms to isolate X clients from each other, and has been the subject of several attacks. Such attacks can be prevented with an authorization policy enforcement, which determines the set of security-sensitive operations that an X client can perform on a resource. Indeed, there have been several efforts to secure the X server [BPWC90, EMO+93, KSV03].

[3] Note that some LSM directory operations have the File_ prefix.

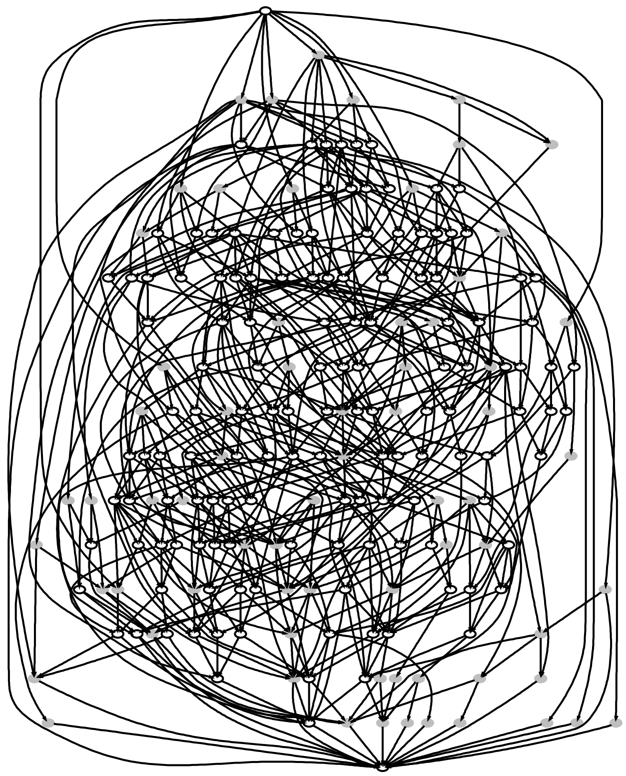


Figure 5.7 Concept lattice for the X server. This concept lattice has 329 nodes and 978 edges. Algorithm 3 identified 115 building blocks in this concept lattice.

We focused on a subset of the X server, its main dispatch loop (called `dix`) that contains code to accept client requests and translate them to lower layers of the server. We focused on this subset because it contains the bulk of code that processes client windows, represented by the `Window` data structure, the resource on which we wanted to identify security-sensitive operations. In addition to `Window`, we also included the `xEvent` data structure, because the X server uses it extensively to process client requests. The API that we used contains 274 functions that the X server exposes to clients.

Concept analysis produced 115 building blocks with 3.76 code patterns, on average, of which 24 had to be refined with precision constraints. The interpretation of two of these fingerprints is discussed below.

1. {*Write* $\perp$ *To* `xEvent->u->mapRequest->window`, *Write* `20` *To* `xEvent->u->type`} is a fingerprint contained in *CodePats* of 5 API functions, embodies an X client request to map a `Window` on the screen, and potentially represents a security-sensitive operation.

2. The fingerprint {*Write* `0` *To* `Window->mapped`, *Write* `18` *To* `xEvent->u->type`}, contained in *CodePats* of 7 API functions embodies unmapping a visible X client window from the screen, also a potential security-sensitive operation.

There have been efforts to secure the X server in the context of the X11/SELinux project, which identified 22 operations on the `Window` resource. As with ext2, we were able to identify at least one fingerprint for each of these security-sensitive operations from those that we mined. For instance, the fingerprints presented above correspond to the Window_Map and Window_Unmap operations on a `Window`, respectively.

The fingerprint mining technique presented in Chapter 4 identified fingerprints for 11 security-sensitive operations on the `Window` resource. However, because that technique is based upon dynamic program analysis, it can only identify fingerprints along paths exercised by manually-chosen test inputs to the X server. Further, that technique, as implemented in A , could automate fingerprint-finding only up to the granularity of function calls; these were then manually refined to the granularity of code patterns. Concept analysis not only identified the fingerprints mined by A at the granularity of code patterns, but did so automatically.

5.5.3 The PennMUSH server

PennMUSH is an open-source online game server. Clients connecting to a PennMUSH server assume the role of a virtual character, as in other popular massively-multiplayer online roleplaying games. For this work, it suffices to think of PennMUSH as a collaborative database of objects that clients can modify. Objects are shared resources, and an authorization policy must govern the set of security-sensitive operations that a client can perform on each object.

Clients interact with PennMUSH by entering commands to a text server, which activates one or more of 603 internal functions, which we used as the API of PennMUSH. Most of these API

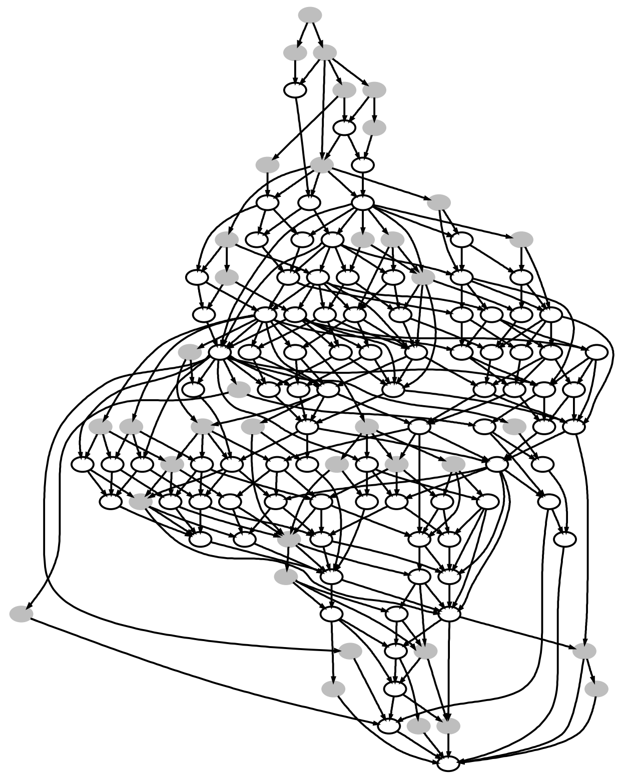


Figure 5.8 Concept lattice for PennMUSH. This concept lattice has 127 nodes and 310 edges. Algorithm 3 identified 38 building blocks in this concept lattice.

functions modify a database of objects. Thus, we tracked how the PennMUSH API manipulates resources of type `object`. Concept analysis produced 38 building blocks. Most of them had only one or two code patterns, so we only had to refine 4 of these building blocks using precision constraints. Two of these fingerprints are discussed below.

1. The fingerprint *Write* $\perp$ *To* `object->name` potentially modifies an object name, and was contained in *CodePats* of 16 API functions, representing creation, destruction and modification of objects. Unauthorized clients must be disallowed from changing the name of an `object`, indicating that this is a fingerprint of a security-sensitive operation.

2. The fingerprint {*Write* `8` *To* `object->type`, *Write* `0` *To* `object->modification_time`, *Write* `1118743` *To* `object->warnings`} appears in *CodePats*(`cmd_pcreate`) and *CodePats*(`fun_pcreate`), both of which are API functions associated with creation of a "character" object.

   Here, the number `1118743` represents a flag that signifies that a character should be warned about problems with the `objects` that they own, and the number `8` written to the field `type` indicates that the newly created object is a character. These code patterns represent necessary steps in character creation in PennMUSH, and thus indicate that this is fingerprint of a security-sensitive operation.

In PennMUSH, the `object` data structure has just 18 fields, while the API contains 603 functions. Each security-sensitive operation is performed at the granularity of accesses to just one or two of the fields of `object`. This explains the smaller number and size of building blocks extracted by concept analysis (as compared to X server).

While the security-sensitive operations that we extracted for PennMUSH can definitely form the basis for writing policies, site-specific policies may be created by combining several security-sensitive operations. For example, an administrator might decide that reading an object's name is as security-sensitive as determining the kind of object. He can then use the domain-specific constraint *Combine*(*Read* `object->name`, *Read* `object->type`) to combine these code patterns together into a single fingerprint that embodies this security-sensitive operation.

5.6 Limitations

An important limitation of the technique presented in this chapter is that it cannot guarantee that all fingerprints have been mined. In particular, it is *incomplete* for unsafe languages such as C, and can thus have *false negatives*, *i.e.,* it can fail to identify a security-sensitive operation, as a result of which insufficient authorization checks will be placed in the retrofitted server.

Two reasons contribute to this limitation, both of which are artifacts of an unsafe language such as C:

1. **Pointer arithmetic** can be used to read from/write to a C `struct` representing a resource data structure. Because code patterns in fingerprints are expressed as ASTs denoting *Read*, *Write* and *Call* operations on structure fields, our static analysis tool can potentially miss accesses to fields, thus resulting in spurious fingerprints, or can completely miss fingerprints.

2. **Direct writes** to data structures are possible via functions such as `memcpy`, which write to untyped regions of memory. Thus, a `memcpy` can be used to write to the field of a data structure, and this write will be missed by the static analysis presented in this chapter.

Further research is necessary to develop a provably complete approach to mine fingerprints for servers written in unsafe languages. However, we conjecture that the technique presented in this chapter is complete (*i.e.,* will not miss fingerprints) for servers written in safe languages, such as Java.

5.7 Static fingerprint mining versus dynamic fingerprint mining

As discussed earlier and demonstrated in our case studies, the static fingerprint mining technique has both better coverage than dynamic mining, and mines fingerprints without the need for an *a priori* description of security-sensitive operations.

While this may seem to suggest that the static fingerprint mining technique subsumes the dynamic fingerprint mining technique, the dynamic technique can potentially be used to improve the results of static fingerprint mining. Static analysis mines building blocks, which are manually examined to identify security-sensitive operations. A description of these security-sensitive operations can then be used as input to the dynamic fingerprint mining technique. The fingerprints so obtained can then be compared against the fingerprints obtained from the static technique. This comparison can potentially be used to prune out false positives produced by the static fingerprint mining technique. In future work, we plan to explore this application of dynamic fingerprint mining to benefit static fingerprint mining.

5.8 Using the static fingerprint mining tool

This section summarizes the steps that a security analyst must follow to find fingerprints and security-sensitive operations using the static mining tool.

- Specify the API to the server and the data types used by the server to represent resources that must be protected.

- Run the tool to obtain building blocks.

- Refine building blocks using constraints. Precision constraints refine building blocks by accounting for imprecision introduced by flow-insensitive program analysis, while domain-specific constraints further refine building blocks using domain knowledge.

- Manually examine building blocks, and interpret the security-sensitive operation performed by the resource accesses contained in the building block. Building blocks may potentially have to be combined during this process.

- Output security-sensitive operations. The building block (or combination of building blocks) of each security-sensitive operation is output as the fingerprint of that security-sensitive operation.

5.9 Summary of key ideas

To summarize, the key contributions of this chapter are:

- A fully static technique to mine fingerprints of security-sensitive operations.

  The use of static analysis overcomes an important limitation of the dynamic analysis-based technique presented in Chapter 4, namely the ability to find fingerprints only along paths exercised by manually chosen inputs to the server. Because static program analysis ensures better coverage than dynamic analysis, the static technique can mine more fingerprints than the dynamic technique.

- A novel algorithm using concept analysis to automatically mine fingerprints of security-sensitive operations.

  To our knowledge, this is the first application of concept analysis to mine security properties of software. The use of concept analysis overcomes another limitation of the technique in Chapter 4, namely the need for an *a priori* description of security-sensitive operations. Concept analysis automatically mines building blocks without the need for an *a priori* description of security-sensitive operations. We were thus able to apply this technique to find security-sensitive operations for PennMUSH, for which no *a priori* description of security-sensitive operations was available.

- Case studies on three real-world servers of significant complexity.

  In each case study, we were able to inspect the lattice and identify security-sensitive operations with a few hours of manual effort and modest domain knowledge. Without our approach, the entire code base must be examined to find such security-sensitive operations.

Chapter 6

Using Fingerprints to Retrofit Legacy Code

This chapter presents a technique that uses fingerprints to statically retrofit legacy code with reference monitor calls (also called *authorization hooks*). It also discusses techniques to synthesize reference monitor code (if a reference monitor implementation is not available) and to analyze reference monitor code (if an implementation is available). The techniques presented in this chapter have been applied to retrofit the X server, and enforce authorization policies on X client requests.

6.1 Problem statement

Given the source code of a server program and a set of fingerprints of security-sensitive operations, the technique presented in this chapter statically identifies all locations that match these fingerprints (and hence perform the corresponding security-sensitive operation).

The server must then be modified by inserting appropriate authorization checks at each location where a security-sensitive operation is performed. This chapter presents a technique to retrofit the server with calls to a reference monitor, and also describes the key steps involved in implementing the reference monitor.

In some cases, the security analyst may decide to use an existing reference monitor with the retrofitted server. This reference monitor may contain existing implementations of authorization queries, where each authorization query may consult the policy to check whether one or more security-sensitive operations is permitted. Such an implementation exists, for example, in SELinux, and the problem in this case is to place calls to these authorization queries at appropriate locations in the server (in the case of SELinux, the server is the Linux kernel). This chapter also presents an algorithm to analyze existing implementations of authorization queries to identify the security-sensitive operations authorized by each query.

6.2 Identifying security-sensitive locations

We employ static pattern matching on the source code of the server to locate all occurrences of fingerprints. Each location that matches a fingerprint is deemed to perform the security-sensitive operation that it represents. Such locations are then instrumented with reference monitor calls. In this respect, our approach bears close resemblance to aspect-oriented programming, where static pattern matching is employed to match *pointcuts* against source code to locate *joinpoints* and weave *advice* [AOS].

Our approach currently identifies security-sensitive locations at the granularity of function calls. Each function that contains all the code patterns in a fingerprint (and satisfies the constraints in the fingerprint) is said to match the fingerprint. The idea is that by mediating calls to functions that contain these patterns, the corresponding security-sensitive operations are mediated as well. This is done using a flow-insensitive, analysis, as described in Algorithm 4.

Algorithm 4 is a simple intraprocedural analysis that first identifies the set of code patterns that appear in the body of a function, and then checks to see if the code patterns contained in the fingerprint of a security-sensitive operation appear in this set. If so, the function is marked as performing the security-sensitive operation. Note that a fingerprint can contain a code pattern of the form *Call* `f`: in this case, the function `f` is marked as performing the security-sensitive operation.

Recall from Figure 3.1 that a fingerprint can either be *intraprocedural* or *interprocedural*. Intraprocedural fingerprints are matched as shown in Algorithm 4, by considering the set of code patterns contained in each function. Interprocedural fingerprints contain code patterns that may appear in different functions, and the matching algorithm shown in Algorithm 4 must thus be extended to match interprocedural fingerprints. This is achieved by a straightforward (and standard) extension to Algorithm 4. We first compute the set of code patterns contained in each function intraprocedurally, as shown in Algorithm 4. We then traverse the call-graph in reverse topological order, gathering the set of code patterns contained in each function. The interprocedural extension is based upon the summary-based approach to interprocedural program analysis [SP81].

To match a fingerprint, we compare the code patterns contained in a fingerprint against the set of code patterns gathered at each node in the call-graph. The node in the call-graph closest to the leaf that contains all the code patterns in a fingerprint is marked as performing the security-sensitive operation. While the fingerprints for the X server and PennMUSH were intraprocedural, we encountered a few interprocedural fingerprints in the case of ext2.

Consider the function `MapSubWindows` in the X server (shown in Figure 2.2). This function maps all children of a given window (`pParent` in Figure 2.2) to the screen. Note that it contains code patterns that constitute the fingerprint of both Window_Enumerate and Window_Map. Thus, Opset(`MapSubWindows`) = {Window_Map, Window_Enumerate}.

Constraints in fingerprints can be used to restrict matches. For example, the fingerprint for Window_Enumerate, shown below, constrains the `WindowPtr` variable used in the first code pattern to be different from the variable in the second `WindowPtr` variable. This is especially useful for cases such as the one shown in Figure 2.2, where the *parent* window's `firstChild` field is read, followed by the `nextSib` field of child windows.

Window_Enumerate :-	*Read* $\texttt{WindowPtr}_1$`->firstChild`
	$\wedge$ *Read* $\texttt{WindowPtr}_2$`->nextSib`
	$\wedge$ `WindowPtr` $\neq$ `0` *Subject to*
	Different($\texttt{WindowPtr}_1$, $\texttt{WindowPtr}_2$)

Figure 6.1 and Figure 6.2 illustrate how interprocedural matching of fingerprints proceeds using the example of the ext2 file system. Figure 6.1 shows interprocedural fingerprints for four security-sensitive operations, Dir_Write, Dir_Rmdir, File_Unlink and Dir_Search; these security-sensitive operations were identified in the LSM project [WCS+02].

Figure 6.2 shows a portion of the call-graph of the ext2 file system, rooted at the node corresponding to the function `ext2_rmdir`. Note that the functions shown in the call-graph (`ext2_unlink`, `ext2_dec_count`, etc.) can also be called by other functions in the kernel; these edges are not shown in Figure 6.2. When a request is received to remove directory `bar` from directory `foo`,

Algorithm: F _S - _L $(\mathcal{X}, S, \mathcal{FP})$

Input : (i) $\mathcal{X}$: Server to be retrofitted,

(ii) S: Set of security-sensitive operations $\{\mathsf{op}_1, \ldots, \mathsf{op}_n\}$, and

(ii) $\mathcal{FP}$: Set of fingerprints $\{\mathrm{fp}_1, \ldots, \mathrm{fp}_n\}$ of $\mathsf{op}_1, \ldots, \mathsf{op}_n$, respectively.

Output : Opset: $\mathcal{X} \rightarrow 2^S$, where Opset($f$) denotes the set of security-sensitive operations performed by a call to f, a function of $\mathcal{X}$.

foreach (function f in $\mathcal{X}$) **do**
  Opset(f) := ϕ;
/* Preprocess function call code patterns */;
foreach (fingerprint fp_i in $\mathcal{FP}$) **do**
  fpset_i := Set of code patterns in fp_i;
  if (fpset_i == $\{Call\ \mathtt{f}_1, \ldots, Call\ \mathtt{f}_m\}$) **then**
    foreach ($f \in \{\mathtt{f}_1, \ldots, \mathtt{f}_m\}$) **do**
      Opset(f) = Opset(f) $\cup$ $\{\mathsf{op}_i\}$;
/* Process all the fingerprints */;
foreach (function f in $\mathcal{X}$) **do**
  CP(f) := Set of code patterns in f (as determined using the ASTs of statements in f);
  foreach (fingerprint fp_i in $\mathcal{FP}$) **do**
    if ($\mathrm{fpset}_i \subseteq$ CP(f) and all contraints specified in fp_i are satisfied in f) **then**
      Opset(f) := Opset(f) $\cup$ $\{\mathsf{op}_i\}$;
return Opset;

Algorithm 4: Finding functions that contain code patterns that appear in fingerprints.

`ext2_rmdir` checks to see that `bar` is empty via a call to `ext2_rmdir_empty` (not shown in Figure 6.2). It then calls `ext2_unlink`, which modifies ext2-specific data structures and removes the entry of `bar` from the inode of `foo`. Finally, it calls `ext2_dec_count` to decrement the field `i_nlink` on the inodes of both `foo` and `bar`.

Dir_Write :-	*Write* $\perp$ *To* `inode->i_ctime`
	$\wedge$ *Call* `address_space_ops->prepare_write()`
Dir_Search :-	*Read* `inode->i_mapping`
File_Unlink :-	*Decrement* (`inode->i_nlink`)
	$\wedge$ $\neg$*Write* 0 *To* `inode->i_size`
Dir_Rmdir :-	*Write* 0 *To* `inode->i_size`
	$\wedge$ *Decrement* `inode->i_nlink`

Figure 6.1 Interprocedural fingerprints for four security-sensitive operations for the ext2 file system

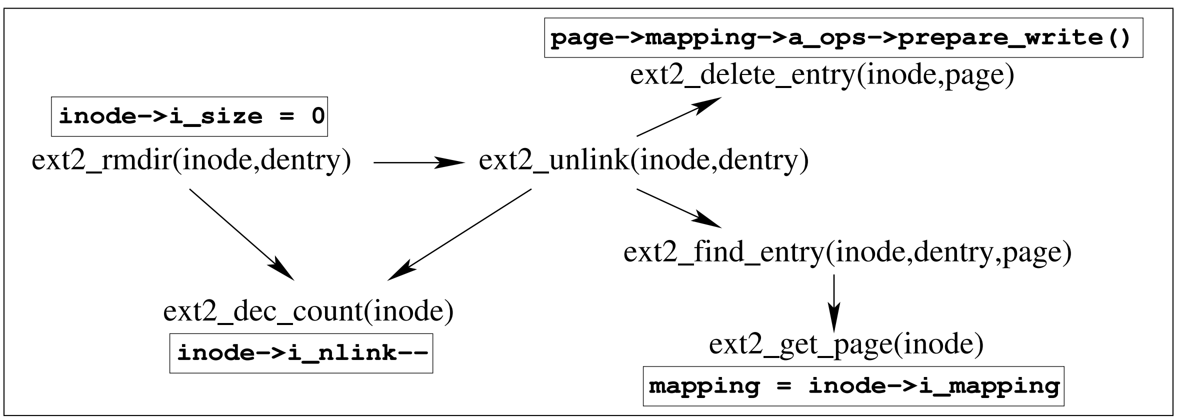


Figure 6.2 A portion of the call-graph of the ext2 file system, rooted at the function `ext2_rmdir`. Code snippets relevant to the example are shown in boxes near the functions that they appear in.

Figure 6.3 shows the results of matching the fingerprints shown in Figure 6.1 interprocedurally using the ext2 call-graph snippet shown in Figure 6.2. The results are self-explanatory. For example, the matching algorithm infers that `ext2_rmdir` performs the security-sensitive operations Dir_Rmdir, Dir_Search and Dir_Write. Note that in this case, matching is performed interprocedurally, and the function in the call-graph, closest to the leaf, that contains all the code patterns in a fingerprint is marked as performing the security-sensitive operation represented by that fingerprint.

In addition to fingerprint matching, we also employ a simple heuristic to help identify the subject requesting the security-sensitive operation, and the object upon which the security-sensitive

ext2_delete_entry	Dir_Write
ext2_get_page	Dir_Search
ext2_find_entry	Dir_Search
ext2_dec_count	File_Unlink
ext2_unlink	File_Unlink, Dir_Write, Dir_Search
ext2_rmdir	Dir_Rmdir, Dir_Write, Dir_Search

Figure 6.3 Results of matching the interprocedural fingerprints shown in Figure 6.1.

operation is performed. To do so, we identify variables of the relevant types that are in scope (some domain knowledge may be required here). For example, in the X server, the subject is always the client requesting the operation, which is a variable of the `Client` data type, and the object can be identified based upon the kind of operation requested. For window operations, the object is a variable of the `Window` data type. This set is then manually inspected to recover the relevant subject and object at each location.

6.3 Evaluation of the matching algorithm

We implemented Algorithm 4 and its interprocedural variant as a plugin to CIL [NMRW02]. We evaluate its effectiveness on the X server in the following sections. In our experiments, we used the fingerprints that were mined using the dynamic fingerprint mining algorithm, described in Chapter 4.

6.3.1 How precise are the security-sensitive locations found?

Algorithm 4 precisely identifies the set of security-sensitive operations performed by each function, with one exception. It reports false positives for the Window_Enumerate operation, *i.e.,* it reports that certain functions perform this operation, whereas in fact, they do not. Out of 20 functions reported as performing Window_Enumerate, only 10 actually do.

We found that this was because of the inadequate expressive power of the code pattern language. In particular, Algorithm 4 matches functions that contain the code patterns `WindowPtr` $\neq$ `0`, *Read* `WindowPtr->firstChild`, and *Read* `WindowPtr->nextSib`, but do not perform linked-list traversal. These false positives can be eliminated by enhancing the fingerprint language with more constructs (in particular, loop constructs).

6.3.2 How easy is it to identify subjects and objects?

As mentioned earlier, we identify subjects and objects using variables of relevant data types in scope. This simple heuristic is quite effective: out of 25 functions in the X server that were identified as performing `Window` operations, the subject, of type `Client`, and object, of type `Window`, were available as formal parameters or derivable from formal parameters in 22 of them. In the remaining functions, specifically, those performing Window_InputEvent, the subject and object were derived from global variables. Even in this case, however, manual inspection quickly reveals the relevant global variables.

6.4 Synthesizing a reference monitor implementation

Locations identified as performing security-sensitive operations by Algorithm 4 are then protected using instrumentation. Because we recover the complete description of security events (*i.e.,* the subject, the object and the security-sensitive operation), adding instrumentation is straightforward, and calls to the `QueryRefmon` function (the reference monitor's API function to place an authorization query) are inserted as described in Section 2.5.5. If the function to be protected is implemented in the server itself and not within a library (*e.g.,* as is the case with all the security-sensitive function calls in the ext2 file system, the X server, and PennMUSH), calls to `QueryRefmon` can be placed within the function body itself. Because the same variables that constitute the security-event are also passed to `QueryRef,on` (*i.e.,* if $\langle sub, obj, op\rangle$ is the security event, then the corresponding call is `QueryRefmon`($\langle sub, obj, op\rangle$)), and the data structures used to represent subjects and objects are internal to the server, this approach avoids TOCTTOU bugs [BD96] by construction.

```
bool QueryRefmon (Client *sub, Window *obj, Operation OP) {
    switch (OP) {
        case Window_Create:
            rc = PolicyLookup (sub->label, NULL, Window_Create);
            if (rc == ALLOW) {
                obj->label = sub->label;
                return TRUE;
            }
            else {
                return FALSE;
            }
        case Window_Map:
            rc = PolicyLookup (sub->label, NULL, Window_Map);
            /* Rest of the code to handle Window_Map */
            ...
        /* More cases to handle security-sensitive operations */
    }
}
```

Figure 6.4 Code fragment showing the implementation of `QueryRefmon` for Window_Create.

We also generate a template implementation of the authorization query function, `QueryRefmon`, as shown in Figure 6.4 (this example is for the X server). The developer is then faced with two tasks:

1. **Implementing the policy consulter:** The developer must insert appropriate calls from a policy management API of his choice into the template implementation of `QueryRefmon`, generated as shown in Figure 6.4. We impose no restrictions on the policy language, or the policy management framework. Figure 6.4 shows an example: it shows a snippet of code generated. Subject and object labels are stored as fields (`label`) in the data structures representing them. The statement in italics, a call to the function `PolicyLookup`, must be

changed by the developer, and substituted with a call from the API of a policy-management framework of the developer's choice.

Several off-the-shelf policy-management tools are now available, including the SELinux policy management toolkit [Trea], which manages policies written in the SELinux policy language. If this tool is used, the relevant API call to replace `PolicyLookup` is `avc_has_perm`.

2. **Implementing reference monitor state updates:** The developer must update the state of the reference monitor based upon the state update function $\mathcal{U}$. Note that $\mathcal{U}$ depends on the policy to be enforced; different policies may choose to update security-labels differently. Functionality to determine how security-labels must change based upon whether an authorization request succeeds or fails must ideally be provided by the policy-management tool that is used (because how security-labels change is policy-dependent).

   However, if this functionality is not available in the policy-management tool used, the developer must update the state of the reference monitor manually. The fragment of code in the case for Window_Create in Figure 6.4 shows a simple example of $\mathcal{U}$: When a new window is created, its security-label is initialized with the security-label of the client that created it.

   It is worth noting for this example that a pointer to the window is created only after memory has been allocated for it (in the `CreateWindow` function of the X server). Thus we place the call to `QueryRefmon` in `CreateWindow` just after the statement that allocates memory for a window; if this call succeeds, the security-label of the window is initialized. Otherwise, we free the memory that was allocated, and return a `NULL` window (*i.e.,* `HandleFailure`) is implemented as `return NULL;`).

Finally, it remains to explain how we bootstrap security-labels in the server. As mentioned earlier, we assume that the server runs on a machine with a security-enhanced operating system. We use operating system support to bootstrap security-labels based upon how clients connect to the server (as has been done by others [Sma05b]). For example, in an SELinux system, all socket connections have associated security-labels, and servers can bootstrap security using these labels.

For example, X clients connect to the X server using a socket. In this case, we can use the security-label of the socket (obtained from the operating system) as the security-label of the X client. We then propagate X client security-labels as they manipulate resources on the X server, as shown in Figure 6.4, where the client's security-label is used as the security-label for the newly-created window.

6.5 Example: Retrofitting the X server to enforce authorization policies

We demonstrate how an X server retrofitted using the techniques presented thus far enforces authorization policies on X clients. In our experiments, we ran the retrofitted X server on a machine running SELinux/Fedora Core 4. Thus, we bootstrapped security-labels in the X server using SELinux security-labels (*i.e.,* a client gets the label of the socket it uses to connect to the server). We describe two attacks that are possible using the unsecured X server, and describe corresponding policies, which when enforced by the retrofitted X server prevent these attacks. In each case we implemented the policy to be enforced within the `QueryRefmon` function itself.

6.5.1 Example I: Setting window properties

Attack. Several well-known attacks against the X server rely on the ability of an X client to set properties of windows belonging to other X clients, *e.g.,* by changing their background or content [KSV03].

Policy. *Disallow an X client from changing properties of windows that it does not own.*

Note that this policy is enforced more easily by the X server than by the operating system. The operating system would have to understand several X server-specific details to enforce this policy. X clients communicate with each other (via the X server) using the X protocol. To enforce this policy, the operating system would have to interpret X protocol messages to determine which messages change properties of windows, and which do not. On the other hand, this policy is easily enforced by the X server because setting window properties involves exercising the Window_Chprop security-sensitive operation.

Enforcement. The call to `QueryRefmon` placed in the `ChangeProperty` function of the X server mediates Window_Chprop. To enforce this policy, we check that the security-label of the subject requesting the operation, and the security-label of the window whose properties are to be changed are equal.

6.5.2 Example II: Secure cut-and-paste

Attack. Operating systems can ensure that a file belonging to a Top-secret user cannot be read by an Unclassified user (the Bell-LaPadula policy [BL76]). However, if both the Top-secret and Unclassified users have `xterm`s open on an X server, then a cut operation from the `xterm` belonging to the Top-secret user and a paste operation into the `xterm` of the Unclassified user violates the Bell-LaPadula policy.

Policy. *Ensure that a cut from a high-security X client window can only be pasted into X client windows with equal or higher security*. This is akin to the Bell-LaPadula policy [BL76].

Existing security mechanisms for the X server (namely, the X security extension [Wig96a]) cannot enforce this policy if there are more than two security-levels.

Enforcement. The cut and paste operations correspond to the security-sensitive operation Window_Chselection of the X server. A identifies the fingerprints of Window_Chselection as calls to two functions, `ProcSetSelectionOwner` and `ProcConvertSelection` in the X server. It turns out that the former is responsible for the cut operation, and the latter for the paste operation. Calls to `QueryRefmon` placed in these functions are used to mediate the cut and paste operations, respectively.

We created three users on our machine with security-labels Top-secret, Confidential and Unclassified, in decreasing order of security. The X clients created by these users inherit their security-labels. We were able to successfully ensure that a cut operation from a high-security X client window (*e.g.,* Confidential) can only result in a paste into X client windows of equal or higher security (*e.g.,* Top-secret or Confidential).

6.6 Performance of the retrofitted X server

We measured the runtime overhead imposed by instrumentation by running a retrofitted X server and an unmodified X server on 25 `x11perf` [x11b] benchmarks. We ran the retrofitted X server with a null policy, *i.e.,* all authorization requests succeed, to measure performance overhead. We measured performance overhead by comparing the number of operations per second (as computed by the `x11perf` benchmark suite) in the retrofitted X server against the number of operations per second in an unmodified X server. Overhead ranged from 0% to 18% across the benchmarks, with an average overhead of 2%.

6.7 Analyzing a reference monitor implementation

In some cases, an implementation of the reference monitor may be available, and the security analyst may only wish to determine locations where authorization queries to the reference monitor must be placed. Indeed, we encountered this scenario in our analysis of the ext2 file system, where we had a reference monitor implementation available (namely, that implemented in SELinux), and wanted to determine where to place authorization queries. In such cases, the reference monitor need not be synthesized, as described in Section 6.4. Instead, the reference monitor implementation must be analyzed to determine the set of security-sensitive operations that are authorized by each authorization query function that is exported by the reference monitor.

This section describes an algorithm that analyzes reference monitor implementations, and extracts, for each query function in the implementation, the set of security-sensitive operations authorized by that query function, and the parameters with which the query must be invoked. We use the ext2 file system and the SELinux reference monitor implementation as the running example in our explanation of the algorithm.

Consider Figure 6.5, which shows a snippet of the implementation of the authorization query `selinux_inode_permission` in the SELinux reference monitor implementation. This snippet authorizes searching, writing to, or reading from an inode representing a directory, based upon the value of `mask`. The authorization is performed by the call to `inode_has_perm`, which authorizes

```
int selinux_inode_permission(struct inode *inode, int mask) {
    ...
    if (!mask) {
        return 0;
    }
    return inode_has_perm (current, inode,
              file_mask_to_av(inode->i_mode,mask), NULL);
}

static inline access_vector_t file_mask_to_av (int mode, int mask) {
    access_vector_t av = 0;
    if ((mode & S_IFMT) != S_IFDIR) {
        /* File-related security-sensitive operations */
        ...
    }
    else {
        if (mask & MAY_EXEC)
            av |= Dir_Search;
        if (mask & MAY_WRITE)
            av |= Dir_Write;
        if (mask & MAY_READ)
            av |= Dir_Read;
    }
    return av;
}
```

Figure 6.5 Code for the SELinux authorization query `selinux_inode_permission` (borrowed from the Linux-2.4.21 kernel).

a security-sensitive operation on an inode based upon the *access vector* it is invoked with [1]. In Figure 6.5, the access vector is obtained by a call to `file_mask_to_av`.

[1]Security-sensitive operations are represented in SELinux using bit-vectors, called access vectors. The macros Dir_Search, Dir_Write and Dir_Read in Figure 6.5 represent these bit-vectors

Our analysis algorithm, described in Algorithm 5 and Algorithm 6, produces the output shown below upon analyzing the code snippet shown in Figure 6.5.

$\langle$(mask $\neq$ 0) $\wedge$ inode_isdir $\wedge$ (mask & MAY_EXEC) $\|$ Dir_Search$\rangle$
$\langle$(mask $\neq$ 0) $\wedge$ inode_isdir $\wedge$ (mask & MAY_WRITE) $\|$ Dir_Write$\rangle$
$\langle$(mask $\neq$ 0) $\wedge$ inode_isdir $\wedge$ (mask & MAY_READ) $\|$ Dir_Read$\rangle$
where "inode_isdir" denotes o(inode->i_mode & S_IFMT == S_IFDIR).

Each line of the output is a tuple of the form $\langle$predicate $\|$ operation$\rangle$, where the predicate only contains formal parameters of the authorization query. This tuple is interpreted as follows: if the authorization query is invoked in a context such that predicate holds, then it checks that the security-sensitive operation operation is authorized. In this case, our analysis algorithm infers that for inodes that represent directories (*i.e.,* the inodes with (inode->i_mode & S_IFMT == S_IFDIR)) the hook `selinux_inode_permission` checks that the security-sensitive operations Dir_Search, Dir_Write or Dir_Read are authorized, based upon the value of `mask`.

We now proceed to explain Algorithm 5. For ease of explanation, assume that there is no recursion; we explain how we deal with recursion later in the section. The analysis proceeds by first constructing the call-graph of the reference monitor implementation. The call-graph is processed in reverse topologically sorted order, *i.e.,* starting at the leaves, and proceeding upwards. For each node in the call-graph, it produces a summary, and outputs summaries of authorization queries, exported by the reference monitor.

Summary construction is described in Algorithm 6. The summary of a function f is a set of pairs $\langle$pred $\|$ op$\rangle$, denoting the condition (pred) under which a security-sensitive operation (op) is authorized by f. The analysis in Algorithm 6 is flow- and context-sensitive. That is, it respects the control-flow of each function, and precisely models call-return semantics.

Intuitively, summary construction for a function f proceeds by propagating a predicate p though the statements of f. At any statement, the predicate denotes the condition under which control-flow reaches the statement. The analysis begins at the first statement of the function f (denoted by Entrypoint(f)), with the predicate set to true.

Algorithm: A _R -M (M, H)

Input : (i) M: Reference monitor containing source code of authorization query functions,

(ii) H: A set containing the names of authorization queries exported by the reference monitor.

Output : For each $h \in H$, a set $\{\langle$predicate $\|$ operation$\rangle\}$, denoting the security-sensitive operations authorized by each authorization query, and the conditions under which they are authorized.

```
Construct the call-graph G of the reference monitor M
L := List of vertices of G, reverse topologically sorted
foreach (f ∈ L) do
    Summary(f) := A _F (f, Entrypoint(f), true)
foreach (h ∈ H) do
    Output Summary(h)
```

Algorithm 5: Analyzing a reference monitor implementation to determine security-sensitive operations authorized by each authorization query.

At an `if-(q)-then-else` statement, the true branch is analyzed with the predicate $\mathsf{p} \wedge \mathsf{q}$, and the false branch is analyzed with the predicate $\mathsf{p} \wedge \neg\mathsf{q}$. For instance, the value of p at line 203 in Figure 6.5 is true. Thus, lines 207-214 are analyzed with true $\wedge$ (mode & S_IFMT) == S_IFDIR. At `Call` $g(a_1, a_2, \ldots, a_n)$, a call to the function g, the summary of g is specialized to the calling-context. Note that because of the order in which functions are processed in Algorithm 5, the summary of g is computed before f is processed. The summary of g is a set of tuples $\langle \mathsf{q}_i \| \mathsf{op}_i \rangle$. Because of the way summaries are computed, formal parameters of g appear in the predicate q_i. To specialize the summary of g, actual parameters $a_1, a_2, \ldots, a_n$ are substituted in place of formal parameters in q_i. The resulting predicate r_i is then combined with p, and the entry $\langle \mathsf{p} \wedge \mathsf{r}_i \| \mathsf{op}_i \rangle$ is included in the summary of f. Intuitively, g authorizes operation op_i if the predicate q_i is satisfied. By substituting actual parameters in place of formal parameters, we determine whether the current call to g authorizes operation op_i; *i.e.,* whether the predicate q_i, specialized to the calling context, is satisfiable. Because the call to g is reached in f under the condition p, an operation is authorized by g only if $\mathsf{p} \wedge \mathsf{r}_i$ is satisfiable.

For other statements, the analysis determines whether the statement potentially authorizes an operation op. Determining whether a statement authorizes an operation op is specific to the way security-sensitive operations are represented in the kernel module. For instance, in SELinux, security-sensitive operations are denoted by bit-vectors, called access vectors and there is a one-to-one mapping between access vectors and security-sensitive operations. Thus, for the SELinux reference monitor we use the occurrence of an access vector (*e.g.,* reading its value) in a statement to determine if the statement authorizes a security-sensitive operation.

Where possible, the predicate p is also updated appropriately based upon the action of statement s. For instance, if the statement in question is `j := i`, and predicate p propagated to this statement is $(\mathsf{i} == 3)$, then the predicate p is updated to $(\mathsf{j} == \mathsf{i}) \wedge (\mathsf{i} == 3)$. In cases where the effect of s on p cannot be determined, the new value of p is set to $\mathsf{Unknown}$, a special value denoting that the value of p cannot be determined precisely.

For functions that have a formal parameter of type `access_vector_t`, but do not refer to any particular access vector (such as Dir_Read, Dir_Write, or Dir_Search), the analysis returns $\{\langle \mathsf{true} \parallel \lambda x.x \rangle\}$ (not shown in Algorithm 6), which says that the function potentially authorizes any security-sensitive operation, based upon the access vector it is invoked with (the variable x in $\lambda x.x$ denotes the access vector).

After processing a statement s in f, the analysis continues by processing the control-flow-successors of s. The analysis terminates when all the statements reachable from $\mathsf{Entrypoint}(f)$ have been analyzed. To keep the analysis tractable, Algorithm 6 analyzes loop bodies exactly once. That is, it ignores back-edges of loops. As a result, loops are treated as conceptually equivalent to `if-then-else` statements.

Finally, any local variables of f appearing in predicates p (for each $\langle \mathsf{p} \parallel \mathsf{op} \rangle$ in the summary of f) are quantified-out. As a result, predicates appearing in the summary of f only contain formal parameters of f.

Algorithm: A _F (f, s, p)

Input : (i) f: Function name,

(ii) s: Statement in f from which to start the analysis,

(iii) p: A Boolean predicate.

Output : A set $\{\langle \mathsf{predicate} \parallel \mathsf{operation} \rangle\}$.

$R := \phi$

switch T - (s) **do**

  case `if` (q) `then` B_{true} `else` B_{false}

    $R :=$ A _F $(f, \mathsf{Entrypoint}(B_{true}), \mathsf{p} \wedge \mathsf{q})$

      $\cup$ A _F $(f, \mathsf{Entrypoint}(B_{false}), \mathsf{p} \wedge \neg \mathsf{q})$

  case `Call` $g(a_1, a_2, \ldots, a_n)$

    $G := \mathsf{Summary}(g)$

    foreach $(\langle \mathsf{q}_i \parallel \mathsf{op}_i \rangle \in G)$ **do**

      $\mathsf{r}_i := \mathsf{q}_i$ specialized with $a_1, a_2, \ldots, a_n$

      $R := R \cup \{\langle (\mathsf{p} \wedge \mathsf{r}_i) \parallel \mathsf{op}_i \rangle\}$

    $R := R \cup$ A _F $(f, \mathsf{ControlFlowSucc}(f, s), \mathsf{p})$

  otherwise

    if (s authorizes security-sensitive operation op) **then** $R := \{\langle \mathsf{p} \parallel \mathsf{op} \rangle\}$

    Update p appropriately

    $R := R \cup$ A _F $(f, \mathsf{ControlFlowSucc}(f, s), \mathsf{p})$

foreach $(\langle \mathsf{p} \parallel \mathsf{op} \rangle \in R)$ **do**

  Existentially quantify-out any local variables of f appearing in p

return R

Algorithm 6: Determining the security-sensitive operations authorized by each function, and the conditions under which they are authorized.

Algorithm 5 and Algorithm 6 are, in effect, a simple implementation of a symbolic execution engine [Kin76]. Of course, implementing a full-fledged symbolic execution engine for C is a significant engineering exercise [GKS05, SMA05a]; our implementation is simplistic, and ignores effects of aliasing (and is thus incomplete).

We illustrate Algorithm 5 using Figure 6.5(A). For the function `file_mask_to_av`, Algorithm 6 returns the output shown below, where mode_isdir denotes mode & S_IFMT == S_IFDIR.

⟨mode_isdir ∧ (mask & MAY_EXEC) ‖ Dir_Search⟩
⟨mode_isdir ∧ (mask & MAY_WRITE) ‖ Dir_Write⟩
⟨mode_isdir ∧ (mask & MAY_READ) ‖ Dir_Read⟩

Observe that the summary only contains formal parameters of `file_mask_to_av`. When this summary is specialized to the call on line 107, formal parameters are replaced with the actual parameters (e.g., `mode` by `inode->i_mode`), thus specializing the summary to the call-site, producing the output shown below, where inode_isdir denotes inode->i_mode & S_IFMT == S_IFDIR.

⟨inode_isdir ∧ (mask & MAY_EXEC) ‖ Dir_Search⟩
⟨inode_isdir ∧ (mask & MAY_WRITE) ‖ Dir_Write⟩
⟨inode_isdir ∧ (mask & MAY_READ) ‖ Dir_Read⟩

For `inode_has_perm`, Algorithm 6 returns $\{\langle \text{true} \parallel \lambda x.x \rangle\}$, which intuitively means that the function authorizes a security-sensitive operation based upon the access vector (x) passed to it. Thus, when this call to `inode_has_perm` is specialized to the call on line 107, the summary obtained is the same shown above. Because line 107 in `selinux_inode_permission` is reached when (mask $\neq$ 0), this predicate is combined with predicates in the summary of the function `inode_has_perm` to produce the result shown in Figure 6.5(B).

Recursion in the kernel module introduces strongly-connected components in its call-graph. Note that Algorithm 5 requires the call-graph to be a directed acyclic graph (DAG). To handle recursion, we consider the functions in a strongly-connected component together. That is, we produce a consolidated summary for each strongly-connected component. Intuitively, this summary is the set of security-sensitive operations (and the associated conditions) that could potentially be authorized if *any* function in the strongly-connected component is called. Observe that handling recursion also requires a small change to lines (7)-(11) of Algorithm 6. Because of recursion, the summary of a function g that is called by a function f may no longer be available in line (7), in which case we skip forward to line (11).

Precision of the analysis. Observe that Algorithm 6 analyzes all reachable statements of each function. Thus, if a function f authorizes operation op, then $\langle \mathsf{q} \parallel \mathsf{op} \rangle \in \mathsf{Summary}(f)$, for some predicate q.

However, because of the approximations employed by Algorithm 5 and Algorithm 6 to keep the analysis tractable, the predicate q may not accurately describe the condition under which op is authorized. In particular, because Algorithm 6 ignores back-edges on loops, loop bodies are analyzed exactly once, and the predicates retrieved will be imprecise. Similarly, because Algorithm 6 employs a heuristic to handle recursion, the predicates retrieved will be imprecise. These predicates are used during hook placement to determine the arguments that the hook must be invoked with. Thus, imprecision in the results of the analysis will mean manual intervention to determine how hooks must be invoked.

In our experiments on the SELinux reference monitor, we found that the code of the reference monitor was relative simple, and we were able to retrieve the conditions precisely in most cases. For instance, there were no loops in any of the functions from the SELinux reference monitor that we analyzed.

6.8 Using the matching tool

This section summarizes the steps that a security analyst must follow to use fingerprints to locate where security-sensitive operations are performed by a server.

- Run Algorithm 4, which identifies the security-sensitive operations performed by each function by matching fingerprints.

- For each function and for each security-sensitive operation performed by that function, determine the subject requesting the operation and the object affected by the operation.
- Run the transformation tool to insert authorization checks. If a reference monitor implementation is already available, use Algorithm 5 and Algorithm 6 to analyze the implementation and determine the authorization check that must be inserted.
- For each authorization check, determine how to handle failed authorizations, and insert appropriate failure-handling code.

6.9 Summary of key ideas

To summarize, the key contributions of this chapter are:

- A static pattern-matching algorithm to match fingerprints against server source code, and locate security-sensitive operations. The pattern-matching algorithm works both intraprocedurally and interprocedurally, to match intraprocedural and interprocedural fingerprints, respectively.
- Techniques to synthesize a reference monitor implementation, in cases where an implementation is not available.
- Techniques to statically analyze a reference monitor implementation, and determine the security-sensitive operations authorized by each authorization query function exported by the reference monitor, when an implementation is available.

Chapter 7

Related Work

Authorization policy enforcement is a topic of central interest to computer security, and has received much attention over the last thirty five years. This chapter surveys related work in the area.

7.1 Foundations of authorization

Authorization was first formalized by Lampson using the notion of an access control matrix [Lam74]. Each column of the access control matrix corresponds to a system resource, and each row corresponds to a system user. The matrix entry (*sub*, *obj*) denotes the *rights* [1]. (*e.g.,* read, write, create, own) that system user *sub* has on system resource *obj*. Given a set of rules to create, modify and delete entries in an access control matrix, it is natural to ask the following *safety* question: can a subject *sub* ever have a right *r* on a resource *obj*? This problem was shown to be undecidable [HRU76].

While an access control matrix is an instantaneous description of the set of system resources that a subject can access, there are historically two ways to administer such an access matrix: the Discretionary Access Control (DAC) and the Mandatory Access Control (MAC) model [TCS85]. In the DAC model (*e.g.,* the Graham-Denning model [GD72]), the access rights that a user has on system resources that he owns can be *delegated* to others, *i.e.,* the access rights on resources that he owns are at his discretion. In contrast, in the MAC model (*e.g.,* the Bell LaPadula model [BL76] and the Biba model [Bib77]), access rights that a user has on system resources are decided by a

[1] The term *rights* is synonymous with the term *security-sensitive operation* used in this document as well as with the term *permission* that is also used in the literature [GHRS05, JSZ03, Sma03]

central authority, such as the system administrator, and cannot be changed at the discretion of the user. MAC policies have historically been used only in military applications, while DAC has been available on commercial operating systems, such as UNIX. However, recent developments, such as SELinux [LS01a, LS01b], have enabled the deployment of MAC in commodity operating systems.

There are two popular ways to represent an access control matrix, namely, *access control lists* (ACLs) and *capabilities*. Access control lists typically associate each resource on the system with the set of access rights that each subject has on the resource. In contrast, capabilities typically associate each subject with the set of access rights that the subject has on system resources. Thus, if we assume that each column of the access control matrix represents a system resource, and each row corresponds to a subject, access control lists are obtained by reading off columns of the matrix, while capabilities are obtained by reading off rows of the matrix. Most modern commercial operating systems implement access control matrices as access control lists, while several historic systems and research operating systems have used capabilities (the book by Levy [Lev84] gives a good overview of historic systems that implemented capabilities; EROS is a modern research operating system that implements capabilities [SSF99]).

7.2 Authorization policy enforcement systems

Reference monitors, introduced by Anderson in 1972 [And72], have historically been the standard mechanism for authorization policy enforcement. As explained in Chapter 1, a reference monitor must satisfy three properties, namely, Complete Mediation, Tamper Resistance, and Verifiability. A reference monitor takes as input a description of the subject (*e.g.,* user ID), a description of the object (*e.g.,* file name), and the security-sensitive operation requested. It consults an authorization policy, and returns a Boolean, which determines whether the subject is allowed to perform the requested security-sensitive operation on the object. An enforcement mechanism (*e.g.,* appropriate runtime checks inserted in code) uses this Boolean value to ensure that the policy is enforced.

Historically, reference monitors have been implemented in the operating system. The main reason is because the operating system manages and mediates access to system resources. For example, most Linux distributions implement mechanisms to enforce DAC authorization policies using ACLs (the `rwx` bits associated with files are an example of ACLs). More recently, operating systems are being augmented and restructured to enforce more powerful access control policies (*e.g.,* MAC) and information flow policies. Security-enhanced Linux (SELinux) [LS01a, LS01b] and the Asbestos operating system [EKV+05, KEF+05] are two examples of such efforts. Both SELinux and Asbestos associate *security labels* with subjects and objects managed by the operating system. They enforce mandatory access control policies and track information flow using these security labels. One of the main differences between SELinux and Asbestos is that SELinux was constructed by augmented the Linux kernel, while Asbestos was designed afresh. Consequently, Asbestos exports new interfaces (*e.g.,* a new system call interface), and applications must be modified or redesigned to run on Asbestos. In contrast, legacy applications can be supported on SELinux.

While an ideal location to implement a reference monitor that mediates access to system resources, as argued in Section 2.7, the operating system may not be suitable to implement a reference monitor that mediates access to resources managed by applications (unless the operating system is equipped with new primitives, as in Asbestos). There thus is an extensive body of research on implementing reference monitors that enforce application-specific authorization policies. For example, Java's security mechanism [GE03] implements the reference monitor as an object of type `AccessController`. Calls to the function `AccessController.checkPermission()` are placed at appropriate locations in code. These calls consult an authorization policy, and determine whether an access should be allowed.

Inlined reference monitors (IRM) are another approach to implement reference monitors [Erl04]. In the IRM approach, security policies are specified as security automata (and are thus safety properties). For example, a policy to protect confidential data managed by a server can be "disallow `send` operations over the network after a `read` operation of sensitive data". These policies are enforced by *inlining* the security automaton into the application to be secured. The application is rewritten by introducing new variables that track the state of the security automaton. These state variables are then used to determine whether a security-sensitive operation (*e.g.,* a `send` or a `read`) should be allowed. IRMs have been used to implement a variety of security policies, including Java stack inspection [ES00]. IRMs were implemented using the PoET/PSLang framework; the framework allows specification of security policies written as security automata, and rewrites Java bytecode to inline these security automata. Naccio [ET99], Polymer [BLW05], Ariel [PH99] and the work by Grimm and Bershad [GB01] are projects similar to the PoET/PSLang framework, and enforce safety policies by rewriting Java bytecode. However, each of these frameworks requires the security analyst to provide a description of the code patterns that represent a security-sensitive operation. These code patterns are used by the rewriting framework to identify locations that perform these security-sensitive operations. Erlingsson [Erl04, Pages 73–82] refers to the problem of identifying these code patterns as the *security event synthesis problem*. These code patterns are akin to fingerprints, developed in this dissertation, and the fingerprint mining techniques presented in Chapter 4 and Chapter 5 address the security event synthesis problem.

7.3 Code retrofitting and refactoring systems

There are numerous tools, both prototypes and commercial, that augment and/or modify existing code. These tools can be broadly classified as *static tools* or *runtime tools*, based upon whether they modify code statically or at runtime. While these tools have been used for a variety of applications ranging from performance debugging to adding extra functionality to legacy applications, this section discusses the application of these tools to application security.

Tools that statically modify code can be further sub-categorized based upon whether they modify binary executables or source code.

One of the first systems that augmented binary executables was the Informer execution profiler [DG71], implemented in Berkeley SDS 940 time-sharing system. The primary purpose of binary modification in this case was to gather and filter profiling events. However, instrumentation could also be added to restrict memory accesses to profiler memory. This idea later appeared in Software Fault Isolation (SFI) [WLAG93], where binary executables were statically modified to restrict accesses to memory. While these systems implemented a fixed policy on memory accesses, more general binary rewriting tools, such as ATOM [SE94] and its successors, Vulcan [ESV01] and Phoenix [Pho], allow arbitrary modification, and thus enforcement of arbitrary security policies (*e.g.,* specified as security automata). For example, Control-Flow Integrity [ABEL05] is a sandboxing technique built on Vulcan that uses binary analysis and modification to restrict acceptable control flows in an application, and thus restrict the effect of control-hijacking attacks. While the tools described above work on machine code, several tools that work on Java bytecode have also been proposed. These include the PoET/PSLang framework [Erl04] discussed earlier, and the SOOT framework [soo], which provides intermediate representations and tools to analyze and modify Java bytecode.

Among the tools that modify source code, CIL is a well-used framework [NMRW02] that allows analysis and modification of C source code. CIL simplifies and distills C code into a few constructs, which enables easy design and implementation of program analysis and transformation tools. The algorithms that were discussed in Chapter 5 and Chapter 6 were implemented in CIL. CIL has also been used for a variety of other code retrofitting and refactoring projects, including CCured [NCH+05, NMW02], which analyzes and instruments C programs to enforce type safety, as well as PrivTrans [BS04], which statically partitions C programs to enforce privilege separation.

While all the above tools statically modify code, tools such as Valgrind [NS07], Dyninst [dyn, HMC94] and Dynamo [BDB00] allow arbitrary modification of code at runtime. These tools have also been used for security, *e.g.,* to perform dynamic taint analysis [NS05] and for program shepherding (a sandboxing technique) [KBA02].

7.4 Aspect-oriented programming

The approach to retrofitting legacy code presented in this dissertation follows the aspect-oriented programming paradigm (AOP) [AOS, KLM+97]. An aspect is defined to be a concern, such as security or error-handling, that crosscuts a program. In aspect-oriented programming languages, (*e.g.,* AspectJ [Aspb], AspectC++ [Aspa]) these concerns are developed independently, as advice. An aspect-weaver merges advice with the program at certain joinpoints. Pointcuts are often used to express a family of joinpoints (*e.g.,* using regular expressions). Thus, pointcuts are patterns that succinctly represent joinpoints. The aspect weaver matches these patterns with the program to identify joinpoints.

Drawing parallels to the approach presented in this dissertation, each location in source code where a reference monitor call must be inserted is a joinpoint. Because a fingerprint is a set of code patterns that identifies multiple such locations, each fingerprint is a pointcut. The matching algorithm and the associated program transformation implement compile-time aspect weaving, while the body of the reference monitor that executes at runtime to consult an authorization policy serves as the advice. Note that other projects, such as IRM [Erl04], Naccio [ET99] and Polymer [BLW05], as well as our own prior work on Tahoe [GJJ05] follow the aspect-oriented programming paradigm.

A key problem in aspect-oriented programming is that of identifying joinpoints—this is known as the problem of *aspect mining*, and is an area of active current research. Concept analysis is one approach that has been used both in conjunction with static analysis as well as with dynamic analysis to mine aspects (Ceccato *et al.* present a survey of such techniques [CMM+05]). For example, concept analysis has been used on identifier names to statically find methods and classes that implement similar functionality [TM04]. Dynamic analysis in conjunction with concept analysis has been used to find methods that implement a particular feature [EKS03, TC04]. The idea here is to run an instrumented version of the program under different use-cases and label the traces with these use cases. Each trace contains information about the methods executed. Traces are then clustered using concept analysis to find crosscutting concerns, and thus identify aspects.

7.5 Authorization policy formulation and analysis

While this dissertation has focused on the problem of authorization policy enforcement, the problem of formulating appropriate authorization policies to meet site-specific security goals, and the problem of analyzing an existing policy to ensure that it meets site-specific security goals are also important to ensure security.

Most prior work on authorization policy formulation has focused on formulating policies that ensure that an application satisfies the Principle of Least Privilege, *i.e.,* that an application has access to all, and only, those resources that it needs to accomplish its task [SS75]. Systrace [Pro03] and Polgen [HGH+05] are two such tools. Both these tools run the program for which a policy is to be written, and observe the set of resource accesses that it makes during a training phase. Audit logs generated during the training phase are examined, and are appropriately converted into policy statements. Note that this is also the intended usage of the `audit2allow` tool from the SELinux policy development toolkit [Treb].

Work on analyzing authorization policies focuses on ensuring that these policies conform to site-specific security goals. For example, the Gokyo tool [JSZ03] analyzes SELinux authorization policies to detect integrity violations. Guttman *et al.* [GHRS05] present the use of LTL model checking to analyze SELinux policies. Desirable safety properties are expressed as LTL formulae. Appropriately expressed SELinux policies and LTL formulae are then fed to the SPIN model checker [Hol03], which reports violations of these safety properties.

7.6 Other related work

This section presents related work in two areas, namely, root-cause analysis, where the techniques developed are related to dynamic fingerprint mining, and X window system security, where there is a rich body of work from the early nineties on securing the X server.

7.6.1 Root-cause analysis

Because fingerprints denote code patterns that embody security-sensitive operations, mining fingerprints is akin to mining root-causes of security-sensitive operations. There is a rich body of research on root-cause analysis techniques, developed primarily for debugging. Most existing root-cause analysis techniques use “good” and “bad” traces to localize the root-cause of a bug [CZ05, Lib04, Zel02]. The dynamic fingerprint mining technique presented in Chapter 4 is similar to these techniques because it classifies program traces and uses this classification to find fingerprints of security-sensitive operations. The primary difference between these techniques and the dynamic fingerprint mining technique in Chapter 4 is that our technique uses a much richer set of labels for runtime traces, namely an arbitrary set of security-sensitive operations, rather than just “good” or “bad”. As a result, our technique uses the more general concept of set equations (rather than the traditionally-used trace differencing technique) to mine fingerprints. Another approach for trace analysis (primarily for debugging) is dynamic slicing [AH90, KR97, ZG03]. Dynamic slicers use data-flow analysis to work backwards from the effect of a vulnerability, such as a program crash, to the cause of the vulnerability. An interesting avenue for future research will be to adapt the dynamic fingerprint mining technique presented in Chapter 4 to use dynamic slicing to work backwards from the effect of a security-sensitive operation (a tangible side-effect) to the fingerprint of the operation.

7.6.2 X Window system security

There is a rich body of work on techniques to secure the X server. Because the X server was historically developed to promote cooperation between X clients, security (*e.g.,* isolation) of X clients was not built into the design of the server. The X protocol, which X clients use to communicate with the X server, has well-documented security flaws, too [Wig96b]. Prior work to rectify this situation has focused on identifying security requirements for the X server, and creating secure versions of the X server. Most of this work was carried out in the context of the Compartmented Mode Workstation [BPWC90, EP91, Pic91], and the Trusted X projects [EMO+93, Eps90], which built prototype windowing systems to meet the Trusted Computer System Evaluation Criteria. While these efforts focus on retrofitting the X server, there is also work on building X server-like window systems, with security proactively designed into the system, *e.g.,* the Nitpicker secure GUI system [FH05] and the EROS trusted window system [SVNC04]. McCune *et al.* [MPR06] present a system to specifically address the threat of malware that steal sensitive user input by exploiting weaknesses in the X server by establishing a trusted channel between the input device and the target application.

The X security extension [Wig96a] extends the X server by enabling it to enforce authorization policies. It does so by placing reference monitor calls at appropriate locations in the X server, as discussed in this dissertation. To the best of our knowledge, these calls were placed manually, and thus the techniques presented in this dissertation could have assisted in that effort. However, the X security extension is quite limited in the policies that it can enforce. It statically partitions clients into Trusted, and Untrusted, and only enforces policies on interactions between these two classes of clients. Thus for example, if three clients, with security-labels Top-secret, Confidential, and Unclassified connect to the X server simultaneously, the X security extension will group two of them into the same category, and will not enforce policies on clients in the same category. The techniques presented in this dissertation can retrofit the X server with mechanisms to enforce arbitrary authorization policies.

Chapter 8

Conclusions and Future Work

While it is ideal to proactively design software systems for security, economic and practical considerations often preclude this in practice. This motivates the need for retroactive techniques to analyze and transform legacy code for security.

This dissertation has presented techniques to analyze and retofit legacy code with mechanisms for authorization policy enforcement. It introduced fingerprints, a low-level language to represent security-sensitive operations, and showed that fingerprints can be used to identify locations in source code that must be guarded by reference monitor calls. A central contribution of this dissertation is a set of techniques based upon static and dynamic program analysis to mine fingerprints by analyzing legacy code.

However, the work presented in this dissertation is not without its limitations, and there are several directions in which to extend the work presented in this dissertation to overcome these limitations.

1. **Reducing the size of the TCB.** As discussed in Section 2.3, a key shortcoming of the approach presented in this dissertation is that it increases the size of the TCB. In particular, the legacy software system that is being retrofit is assumed to be benign. As a result, the TCB must be extended to include the legacy software system. An interesting future direction will be to examine techniques to enforce authorization policies without increasing the size of the TCB. One way to achieve this will be to redesign the operating system (which is traditionally included in the TCB) to enforce application-level authorization policies. A key challenge here will be to do so while remaining compatible with legacy applications.

2. **Achieving soundness and completeness.** The fingerprint-based approach presented in this dissertation is neither *sound* nor *complete*. Consequently, it is not completely automatic and requires manual intervention to prune false positives (which result because the approach is not sound) and identify false negatives (which result because the approach is not complete).

   The approach is not sound primarily because of the limited expressive power of the language that is currently used to represent fingerprints. Two extensions to the language will greatly improve its expressive power. First, the language must be extended to express temporal relationships between code patterns, *e.g.,* using finite state automata to express fingerprints. Second, the language must be extended to include data-flow information. Augmenting the language with data-flow information will enable the expression of fine-grained information that determines how resources are affected by a security-sensitive operation.

   The approach lacks completeness for unsafe languages (such as C). As discussed in Section 5.6, this is a consequence of type-safety violations, such as those caused by pointer aritmetic and direct writes to memory. One approach to achieving completeness is to add extra runtime checks (*á la* CCured [NCH+05, NMW02]) to the retrofitted program to ensure type safety. The resulting retrofitted system will contain checks that enforce type safety in addition to those that enforce site-specific authorization policies.

3. **Automating failure handling.** An important issue that has been side-stepped by the approach presented in this dissertation is, "how to handle failed authorization policy checks?" While the primary goal is to ensure that a security-sensitive operation is never performed when an authorization policy check fails, an important secondary goal is to ensure that the server notifies clients in such a way that the failed check is handled gracefully by the clients. For example, failure to create a new window or copy from a window must not crash an X client that requested this operation. In current work, we defer the task of implementing failure handling code to a human. An interesting future direction will be to investigate automated techniques to gracefully handle failure in a principled and automated way.

4. **Enforcing policies on unmodified binaries.** The approach presented in this dissertation modifies legacy (source) code by retrofitting it with authorization checks. An interesting future direction will be to investigate techniques to enforce authorization policies on unmodified binaries. Doing so will require constructing a runtime envionment that will enforce authorization policies as code executes. Such a runtime environment will enable the enforcement of authorization policies on commercial off-the-shelf servers that may not be amenable to analysis and transformation.

The above directions provide fodder for much future research and experimentation.

LIST OF REFERENCES

[ABEL05] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353. ACM Press, November 2005.

[AH90] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256. ACM Press, June 1990.

[AMBL03] G. Ammons, D. Mandelin, R. Bodik, and J. R. Larus. Debugging temporal specifications with concept analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 182–195. ACM Press, June 2003.

[And72] J. P. Anderson. Computer security technology planning study, volume II. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, October 1972.

[AOS] Glossary of terms from the aspect-oriented software development Wiki. http://aosd.net/wiki/index.php?title=Glossary.

[Apa] The Apache web-server. http://www.apache.org.

[App] AppArmor Application Security. http://www.novell.com/linux/security/apparmor.

[Arg] Argus PitBull. http://www.argus-systems.com.

[Aspa] The home of AspectC++. http://www.aspectc.org.

[Aspb] AspectJ project. http://www.eclipse.org/aspectj.

[BD96] M. Bishop and M. Digler. Checking for race conditions in file accesses. *Computer Systems*, 9(2):131–152, Spring 1996.

[BDB00] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, June 2000.

[Bib77] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corporation, Bedford, MA, April 1977.

[BL76] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, March 1976.

[BLW05] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 305–314. ACM Press, June 2005.

[BN89] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214. IEEE Computer Society Press, May 1989.

[BPWC90] J. Berger, J. Picciotto, J. Woodward, and P. Cummings. Compartmented mode workstation: Prototype highlights. *IEEE Transactions on Software Engineering*, 16(6):608–618, June 1990.

[BS04] D. Brumley and D. Song. PrivTrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, pages 57–72. USENIX Association, August 2004.

[CMM+05] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. A quantitative comparison of three aspect mining techniques. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 13–22. IEEE Computer Society Press, May 2005.

[CV65] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. *Proceedings of the AFIPS Fall Joint Computer Conference*, 27:185–196, 1965.

[CW02] H. Chen and D. Wagner. Mops: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244. ACM Press, November 2002.

[CZ05] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, pages 342–251. ACM Press, May 2005.

[DG71] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *Proceedings of the IFIP Congress 1971 (Information Processing) - Foundations and Systems*, pages 320–326. North Holland, August 1971.

[DRU05] V. Diwakara, V. Rajegowda, and P. Uppuluri. Preventing information leaks in X windows. In *Proceedings of the International Conference on Communication, Network, and Information Security*, November 2005.

[dyn] Dyninst: An application program interface for runtime code generation. http://www.dyninst.org.

[EKS03] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, March 2003.

[EKV+05] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Maziéres, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, pages 17–30. ACM Press, October 2005.

[EMO+93] J. Epstein, J. McHugh, H. Orman, R. Pascale, A-M. Squires, B. Danner, C. Martin, M. Branstad, G. Benson, and D. Rothnie. A high assurance window system prototype. *Journal of Computer Security*, 2(2-3):159–190, 1993.

[EP91] J. Epstein and J. Picciotto. 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*, October 1991.

[Eps90] J. J. Epstein. A prototype for trusted X labeling policies. In *Proceedings of the 6th Annual Computer Security Applications Conference*, pages 221–230. IEEE Computer Society Press, December 1990.

[Erl04] Ú. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, January 2004.

[ES00] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255. IEEE Computer Society Press, May 2000.

[ESV01] A. Edwards, A. Srivastava, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report 2001-50, Microsoft Research, April 2001.

[ET99] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 32–45. IEEE Computer Society Press, May 1999.

[FGH+04] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 194–208. IEEE Computer Society Press, May 2004.

[FH05] N. Feske and C. Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 85–94. IEEE Computer Society Press, December 2005.

[FHSL96] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE Computer Society Press, May 1996.

[Fle06] B. Fletcher. Case study: Open source and commercial applications in a Java-based SELinux cross-domain solution. In *Proceedings of the 2nd Annual Security-enhanced Linux Symposium*, March 2006.

[GB01] R. Grimm and B. Bershad. Separating access control policy, enforcement and functionality in extensible systems. *ACM Transactions on Computer Systems*, 19(1):36–70, February 2001.

[GD72] G. S. Graham and P. J. Denning. Protection — principles and practice. In *Proceedings of the 1972 AFIPS Spring Joint Computer Conference*, pages 41–429. AFIPS Press, May 1972.

[GE03] L. Gong and G. Ellison. *Inside Java^{TM} 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.

[GHRS05] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in security-enhanced Linux. *Journal of Computer Security (special issue for the 2003 Workshop on Issues in the Theory of Security)*, 13(1), 2005.

[Gia] D. Giampaolo. xkey source code (last accessed in May 2007). http://www.phreak.org/archives/exploits/unix/xwin-exploits/xkey.c.

[GJJ05] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux security modules framework. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 330–339. ACM Press, November 2005.

[GJJ06] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 214–229. IEEE Computer Society Press, May 2006.

[GKJJ07] V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of the 29th International Conference on Software Engineering*, pages 458–467. IEEE Computer Society Press, May 2007.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed, automated, random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages i213–223. ACM Press, June 2005.

[GMA95] R. Godin, R. Missaoui, and H. Alaoui. Incremental concept formation algorithms based on Galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.

[GRS] grsecurity patches for the Linux kernel. http://www.grsecurity.net.

[HAM06] B. Hicks, K. Ahmadizadeh, and P. McDaniel. Understanding practical application development in security-typed languages. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 153–164. IEEE Computer Society Press, December 2006.

[HCXE02] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82. ACM Press, June 2002.

[HGH+05] A. L. Herzog, J. D. Guttman, D. R. Harris, J. D. Ramsdell, A. E. Segall, and B. T. Sniffen. Policy analysis and generation work at MITRE. In *Proceedings of the 1st Annual Security-enhanced Linux Symposium*, March 2005.

[HMC94] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 841–850. IEEE Computer Society Press, May 1994.

[HMS06] M. Hocking, K. Macmillan, and D. Shankar. Case study: Enhancing IBM Websphere with SELinux. In *Proceedings of the 2nd Annual Security-enhanced Linux Symposium*, March 2006.

[Hol03] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, Pearson Education, September 2003.

[HRJM07] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. Integrating SELinuxwith security-typed languages. In *Proceedings of the 3rd Annual Security-enhanced Linux Symposium*, March 2007.

[HRU76] M. Harrison, W. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[IIS] The Microsoft IIS web-server. http://www.microsoft.com/windowsserver2003/iis.

[JDB] JDBC: Java Database Connectors. http://java.sun.com/products/jdbc.

[JEZ04] T. Jaeger, A. Edwards, and X. Zhang. Consistency analysis of authorization hook placement in the Linux security modules framework. *ACM Transactions on Information and System Security (TISSEC)*, 7(2):175–205, May 2004.

[JSZ03] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, pages 59–74. USENIX Association, August 2003.

[KBA02] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206. USENIX Association, August 2002.

[KEF+05] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek, E. Kohler, D. Mazieres, R. Morris, M. Osborne, S. Van DeBogart, and D. Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*. USENIX Association, June 2005.

[Kin76] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[Kle04] A. Klein. Divide and conquer: HTTP response splitting, web cache poisoning, and related topics, March 2004. White Paper, Sanctum Inc.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, June 1997.

[KR97] B. Korel and J. Rilling. Application of dynamic slicing in program debugging. In *Proceedings of the 3rd International Workshop on Automated and Algorithmic Debugging*, volume 2 of *Linkoping Electronic Articles in Computer and Information Science*, pages 43–58. ISSN 1401-9841, May 1997.

[KS74] P. A. Karger and R. R. Schell. Multics security evaluation: Vulnerability analysis. Technical Report ESD-TR-74-193, Deputy for Command and Management Systems, Electronics Systems Division (ASFC), L. G. Hanscom Field, Bedford, MA, June 1974.

[KSV03] D. Kilpatrick, W. Salamon, and C. Vance. Securing the X Window system with SELinux. Technical Report 03-006, NAI Labs, March 2003.

[Lam74] B. W. Lampson. Protection. *Proceedings of the 5th Princeton Conference on Information Sciences and Systems (1971). Reprinted in ACM Operating Systems Review*, 8(1):18–24, January 1974.

[Lev84] H. M. Levy. *Capability-based Computer Systems*. Digital Press, 1984.

[Lib04] B. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Fall 2004.

[LID] LIDS: Linux intrusion detection system. http://www.lids.org.

[LRB+05] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. Hartmann. Protecting against unexpected system calls. In *Proceedings of the 14th USENIX Security Symposium*, pages 239–254. USENIX Association, August 2005.

[LS97] C. Lindig and G. Snelting. Assesing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359. ACM Press, May 1997.

[LS01a] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX track: USENIX Annual Technical Conference*, pages 29–42. USENIX Association, June 2001.

[LS01b] P. Loscocco and S. Smalley. Meeting critical security objectives with security-enhanced Linux. In *Proceedings of the Ottawa Linux Symposium*, July 2001.

[McC04] B. McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc., October 2004.

[McL90] J. McLean. The specification and modeling of computer security. *IEEE Computer*, 23(1):9–16, 1990.

[MPP+07] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. Minimal TCB code execution (extended abstract). In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 267–272. IEEE Computer Society Press, May 2007.

[MPR06] J. M. McCune, A. Perrig, and M. Rieter. Bump in the Ether: A framework for securing sensitive user input. In *Proceedings of the USENIX Annual Technical Conference*, pages 185–198. USENIX Association, June 2006.

[NCH+05] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, May 2005.

[NMRW02] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, April 2002.

[NMW02] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Conference Record of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139. ACM Press, January 2002.

[NS05] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*. ISOC Press, February 2005.

[NS07] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100. ACM Press, June 2007.

[ODB] ODBC: Open Database Connectors. http://www.unixodbc.org.

[Ora] Oracle database. http://www.oracle.com/database.

[Pen] The PennMUSH multi-user dungeon online game server. http://pennmush.org.

[PFH03] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242. USENIX Association, August 2003.

[PH99] R. Pandey and B. Hashii. Providing fine-grained access control for Java programs. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 449–473. Springer, June 1999.

[Pho] The Microsoft Phoenix software optimization and analysis framework. http://research.microsoft.com/phoenix.

[Pic91] J. Picciotto. Towards trusted cut and paste in the X window system. In *Proceedings of the 7th Annual Computer Security Applications Conference*, pages 34–43. IEEE Computer Society Press, December 1991.

[Pos] The Postfix mail program. http://www.postfix.org.

[Pro03] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272. USENIX Association, August 2003.

[RSB] RSBAC: Rule Set Based Access Control. http://www.rsbac.org.

[Sam] The Samba server—opening Windows to a wider world. http://www.samba.org.

[SBBD01] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 144–155. IEEE Computer Society Press, May 2001.

[SE94] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 196–205. ACM Press, June 1994.

[SEL] Security-enhanced Linux. http://www.nsa.gov/selinux.

[Shi07] S. Shimko. Enhancing the security of enterprise products with SELinux. In *The 3rd Security-enhanced Linux Symposium*, March 2007.

[Sif98] M. Siff. *Techniques for Software Renovation*. PhD thesis, University of Wisconsin-Madison, 1998.

[Sma03] S. Smalley. Configuring the SELinux policy. Technical Report 02-007, NAI Labs, February 2003. http://www.nsa.gov/selinux/papers/policy2.pdf.

[SMA05a] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 13th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 263–272. ACM Press, September 2005.

[Sma05b] S. Smalley, 2005. Personal Communication.

[soo] SOOT: A Java optimization framework. http://www.sable.mcgill.ca/soot.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural dataflow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice Hall, 1981.

[SQL] Microsoft SQL server. http://www.microsoft.com/sql.

[SQU] Squid web proxy cache. http://www.squid-cache.org.

[SS75] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[SSF99] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 170–185. ACM Press, December 1999.

[ST98] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of the 6th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 99–110. ACM Press, September 1998.

[SVNC04] J. S. Shapiro, J. Vandenburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *Proceedings of the 13th USENIX Security Symposium*, pages 165–178. USENIX Association, August 2004.

[SVS01] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. Technical Report 01-043, NAI Labs, December 2001. http://www.nsa.gov/selinux/papers/module.pdf.

[TC04] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 112–121. IEEE Computer Society Press, November 2004.

[TCS85] *Trusted Computer System Evaluation Criteria*. United States Department of Defense, December 1985.

[TM04] T. Tourwe and K. Mens. Mining aspectual views using formal concept analysis. In *Proceedings of the 4th IEEE International Workshop on Source code Analysis and Manipulation*, pages 97–106. IEEE Computer Society Press, September 2004.

[Trea] Tresys technology, security-enhanced Linux policy management framework. http://sepolicy-server.sourceforge.net.

[Treb] Tresys technology, SETools policy tools for SELinux. http://www.tresys.com/selinux/selinux_policy_tools.shtml.

[vDK99] A. van Duersen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21st International Conference on Software Engineering*, pages 246–255. ACM Press, May 1999.

[Wal07] E. Walsh. Integrating X.Org with security-enhanced Linux. In *Proceedings of the 3rd Annual Security-Enhanced Linux Symposium*, March 2007.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–344, June 1974.

[WCS+02] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31. USENIX Association, August 2002.

[WD01] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–168. IEEE Computer Society Press, May 2001.

[Wig96a] D. Wiggins. Analysis of the X protocol for security concerns, draft II, X Consortium Inc., May 1996. http://www.x.org/X11R6.8.1/docs/Xserver/analysis.pdf.

[Wig96b] D. Wiggins. Security extension specification, version 7.1, X Consortium Inc., 1996.

[Wil82] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered Sets*, number 83 in NATO ASI, pages 445–470. Dordrecht-Boston: Reidel, 1982.

[WLAG93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 203–216. ACM Press, December 1993.

[X11a] The X Foundation: http://www.x.org.

[x11b] `x11perf`: The X11 server performance test program suite.

[ZEJ02] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48. USENIX Association, August 2002.

[Zel02] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering*, pages 1–10. ACM Press, November 2002.

[ZG03] X. Zhang and R. Gupta. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, pages 319–329. IEEE Computer Society Press, May 2003.