Published in ICSE'07: Proceedings of the
29th International Conference on Software Engineering, Minneapolis, Minnesota, May 2007

Mining Security-Sensitive Operations in Legacy Code using Concept Analysis

Vinod Ganapathy
University of Wisconsin
vg@cs.wisc.edu

David King Trent Jaeger
Pennsylvania State University
{dhking,tjaeger}@cse.psu.edu

Somesh Jha
University of Wisconsin
jha@cs.wisc.edu

Abstract

This paper presents an approach to statically retrofit legacy servers with mechanisms for authorization policy enforcement. The approach is based upon the observation that security-sensitive operations performed by a server are characterized by idiomatic resource manipulations, called fingerprints. Candidate fingerprints are automatically mined by clustering resource manipulations using concept analysis. These fingerprints are then used to identify security-sensitive operations performed by the server. Case studies with three real-world servers show that the approach can be used to identify security-sensitive operations with a few hours of manual effort and modest domain knowledge.

1 Introduction

Software systems must protect shared resources that they manage from unauthorized access. This is achieved by formulating and enforcing an appropriate authorization policy (also called access control policy). The policy specifies the set of *security-sensitive operations* that a user can perform on a resource. For example, a popular policy on UNIX-like systems allows only the `root` to perform the security-sensitive operations Read and Write on the `/etc/passwd` file (the resource). Operating systems have historically had mechanisms such as reference monitors [4] to enforce authorization policies. It is also important for user-space servers, such as middleware, web-, proxy and window-management servers, to implement such mechanisms because they manage shared resources on behalf of their clients. Unfortunately, economic and practical considerations force developers to choose functionality and performance over security. As a result, several legacy servers often completely lack policy enforcement mechanisms. For example, the X11 server [31] can simultaneously manage multiple X client windows, but was not built with mechanisms to isolate one X client from another, leading to several published attacks [20].

This paper investigates techniques for retrofitting legacy servers with authorization policy enforcement mechanisms. The main questions to be addressed when retrofitting a server are *what are the security-sensitive operations to be mediated?* and *where in the server's source code are these operations performed?* In current practice, these questions are answered manually. A team of software engineers inspects the code of the server to determine locations where security-sensitive operations are performed, and places appropriate authorization checks guarding these locations. Not surprisingly, this process is time consuming and error prone [18, 32]. For example, it took almost two years each for the Linux Security Modules (LSM) project [30], where additional authorization checks were added to the Linux kernel to enable enforcement of mandatory access control policies, and the X11/SELinux [20] project, where authorization checks were added to the X11 server. Similar recent efforts have also been time consuming [13, 17]. In short, there are no automated techniques to aid the process of securing legacy servers for authorization.

We build on prior work [15] and develop an approach using concept analysis [29] to drastically reduce the manual effort involved in retrofitting legacy servers. Key to our approach is the observation that security-sensitive

operations performed by a server are associated with idiomatic ways in which resources are manipulated by the server. Such idioms, which we call *fingerprints*, are code-level descriptions of the security-sensitive operations that they represent. Each fingerprint is expressed as a combination of several abstract syntax trees (ASTs), called *code patterns*. We use static program analysis in combination with concept analysis to automatically mine candidate fingerprints. These are then examined and refined manually by a domain expert. After refinement, we statically match each fingerprint against the code of the server to determine locations where the corresponding security-sensitive operation is performed. We then weave hooks to a reference monitor at all these locations to authorize that security-sensitive operation. This ensures that security-sensitive operations performed by the server are mediated by authorization policy lookups.

Our results demonstrate the effectiveness of our approach. We conducted case studies on three real-world systems of significant complexity: the ext2 file system, a subset of the X11 window-management server, and PennMUSH, an online game server [2]. In each case, our 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, our 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. We then refined these candidate fingerprints manually and determined whether each refined fingerprint indeed denoted a security-sensitive operation or not. It took just a few hours of manual effort and modest domain knowledge to find security-sensitive operations in each of our case studies. Without our approach, the entire code base must be examined to find such security-sensitive operations.

The approach presented in this paper overcomes two important limitations of our prior work [15]. While we introduced fingerprints in that work, our approach for finding fingerprints (i) required a high-level description of security-sensitive operations, and (ii) used dynamic program analysis to find fingerprints. Both (i) and (ii) prevented our approach from easily being applied to a wide variety of servers. In particular, while a high-level description of security-sensitive operations was available for the case study that we considered (the X server), this may not be the case with other servers, as indeed was the case with PennMUSH. A dynamic approach to fingerprint-finding meant that the fingerprints found were restricted to code paths exercised by the manually chosen inputs to the server. This paper directly addresses both these shortcomings. Concept analysis automatically mines candidate 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 approach presented here can mine more fingerprints than our prior work.

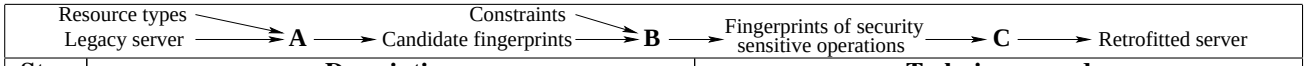
In summary, our main technical contributions are:

1. A fully static approach to retrofit policy enforcement into legacy servers. The key observation used by the approach is that security-sensitive operations performed by a server are associated with idiomatic resource manipulations, called fingerprints.

2. 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.

3. Case studies on three real-world servers of significant complexity. Our case studies demonstrate that our approach is efficient and effective. Our analysis completed in just over 310 seconds even for the largest of our benchmarks and produced manageable concept lattices. In each case, we were able to inspect the lattice and identify security-sensitive operations with a few hours of manual effort and modest domain knowledge.

Note that our approach to retrofit legacy servers follows the aspect-oriented paradigm. In particular, each fingerprint denotes a region of code before which a reference monitor hook must be placed, and thus helps identify join points [5, 19]. The reference monitor query that executes as a result of the hook call is the body of the advice at that join point. Fingerprint-mining is thus aspect mining to find join points relevant to security.



Step	Description	Techniques used
A	Extraction of candidate fingerprints from source code.	Static analysis and concept analysis.
B	Refinement of candidate fingerprints using constraints.	Application of constraints and interpretation of fingerprints.
C	Authorization hook placement using fingerprints.	Pattern (fingerprint) matching and weaving hook calls.

Figure 1. Steps to retrofit policy enforcement, and the techniques used in each step.

2 Approach overview

We give a high-level overview of our approach, depicted in Figure 1. Using a running example, we show how a software engineer would use our approach to mine fingerprints of security-sensitive operations and place hooks. We have currently implemented our analysis to work with C programs, but the underlying principles apply to servers written in other languages as well.

2.1 Running example

We use a subset of ext2, a Linux file system, and one of the case studies in Section 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 approach can identify security-sensitive operations that ext2 performs on directories.

2.2 Step A: From source code to candidate fingerprints

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. This analysis is based upon two assumptions.

First, we assume that it suffices to examine accesses to internal data structures that ext2 uses 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`. Second, we 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 critical data structures to be tracked, and a set of API functions.

The static analyzer identifies how these tracked 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*. A code pattern is either a *Read*, *Write* or a *Call* and is expressed in terms of abstract syntax trees (see Figure 2). 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 tracked 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.

Code-pattern	:=	*Call* AST | *Read* AST
		| *Write* Value *to* AST
Value	:=	constant | AST | $\perp$ (unknown)
AST	:=	(`type-name->`)*field

Figure 2. Grammar for code patterns.

The next step is to identify idiomatic resource manipulations by the ext2 API. The goal here 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. 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. We call each such set a *fingerprint*.

We identify candidate fingerprints using concept analysis [29], a well-known hierarchical clustering technique. At a high-level (details are presented in Section 3), concept analysis identifies candidate fingerprints, as well as the API functions whose code pattern sets contain these candidate fingerprints. We use the term *candidate fingerprints* because as described in Step B, imprecisions introduced in the program analysis step means that each candidate fingerprint may contain multiple fingerprints.

For example, concept analysis inferred that the set of six code patterns shown in Figure 3 is a candidate fingerprint, 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* $\perp$ *To* `inode->u->ext2_inode_info->i_dir_start_lookup`

Figure 3. One of the candidate fingerprints that concept analysis identifies for ext2.

For ext2, we identified 18 such candidate fingerprints, each denoting a unique way in which ext2 manipulates files and directories. While concept analysis is asymptotically inefficient—its complexity is exponential in $\max_i(|\textit{CodePats}(\texttt{ext2\_api}_i)|)$—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 even for the largest of our case studies.

2.3 Step B: Refining candidate fingerprints

In the second step, a domain expert (i) refines candidate fingerprints 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 candidate fingerprints is necessary for two reasons. The first reason is because code analysis employed in Step A is imprecise. As a result, multiple fingerprints may be combined into a single candidate fingerprint. There are two ways in which precision is lost:

1. Code analysis is *flow-insensitive*. A candidate fingerprint may contain a pair of code patterns pat_1, pat_2 that do not always appear together in all executions of the server.

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

We employ *precision constraints* to identify such cases and enable refinement of each candidate fingerprint, 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 candidate fingerprints.

For example, consider the candidate fingerprint shown in Figure 3. 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 candidate fingerprint 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 4 shows, while `ext2_rename` calls both these functions on the instances `old_dir` and `old_dentry`,[1] it calls `ext2_find_entry` only on the instances `new_dir` and `new_dentry` when a certain predicate `new_inode` is satisfied.

```
 1 int ext2_rename (inode *old_dir, dentry *old_dentry,
 2                  inode *new_dir, dentry *new_dentry) {
 3   /* declarations of old_page,new_page,old_de and new_de */
 4   new_inode = new_dentry->d_inode;...
 5   old_de = ext2_find_entry(old_dir,old_dentry,&old_page);
 6   if (new_inode) { ...
 7     new_de = ext2_find_entry(new_dir,new_dentry,&new_page);
 8   } else { ...
 9     /* no call to ext_find_entry */
10   };...
11   ext2_delete_entry(old_de,old_page); ...
12 }
```

Figure 4. 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 candidate fingerprint shown in Figure 3 is split into two fingerprints, as shown in Figure 5. Additional examples of the use of precision constraints appear in Section 4. Note that such constraints can potentially be avoided with sophisticated program analyses, that we plan to explore in future work. However, in our case studies we found that more than 50% of the candidate fingerprints 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, which corresponds to a directory lookup, is not interesting for a specific set of policies to be enforced.

After refinement, the domain expert assigns semantics to each fingerprint, associating it with a security-sensitive operation. For example, Fingerprint (1) in Figure 5 embodies the directory removal operation, while Fingerprint (2) embodies the lookup operation. The LSM project [30] 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

[1] The variable `old_de`, which `ext2_delete_entry` is invoked with on line 11 is derived from `old_dir` and `old_dentry`.

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. Fingerprints obtained after refinement with precision constraints.

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.

2.4 Step C: From fingerprints to hooks

The final step is to place reference monitor hooks and implement the appropriate policy lookups for each hook (*i.e.,* the advice at each join point). We discuss this step in brief here and refer the reader to prior work [15] for details.

Each fingerprint is a set of code patterns that can be matched against the server's source code. Each code fragment that matches a fingerprint is deemed as performing the security-sensitive operation associated with that fingerprint. In prior work [15], we had presented an approach to place hooks at the granularity of function calls, *i.e.,* for each fingerprint that matched the set of code patterns in a function, we would place a reference monitor hook to guard calls to this function with the appropriate security-sensitive operation. For example, using the fingerprints from our running example, we would place a hook guarding the call to `ext2_find_entry` on line (5) of Figure 4 to check that the LSM operation Dir_Search is authorized as follows:

```
 if (check_policy(current process, old_dir, Dir_Search))
      { ext2_find_entry(old_dir, old_dentry,
&old_page); }
 else { Notify current process of failed authorization check }
```

A similar hook will also be placed for the call on line (7). The call to `ext2_delete_entry` on line (11) will be protected with a hook that checks that the client is authorized to perform the LSM operation Dir_Remove_Name on the directory being removed. Several optimizations are possible to this hook placement technique, *e.g.,* placing hooks so as to minimize the number of reference monitor queries executed at runtime. We leave such optimizations for future work.

Note that fingerprints are useful even when hook placements have been decided in advance. For example, if a team of software engineers decides to place just one hook guarding calls to `ext2_rename` (as was done in LSM), then fingerprints determine the security-sensitive operations that must be authorized by that hook. In this case, the hook must authorize Dir_Remove_Name on the old directory (instance `old_dir`) and Dir_Search on both the old (`old_dir`) and new directories (`new_dir`). Indeed, these security-sensitive operations are authorized in the implementation of the hook in the LSM implementation of security-enhanced Linux (SELinux) [22].[2]

[2]SELinux authorizes more security-sensitive operations, corresponding to fingerprints that match code fragments that were omitted from Figure 4.

3 Extracting candidate fingerprints 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 candidate fingerprints.

3.1 Static analysis

Algorithm 1 describes the static code analysis that we have implemented (in CIL [23]). Lines 1-5 employ a simple flow-insensitive analysis to extract for each function a set of code patterns describing how the function manipulates tracked 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 manipulations 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 manipulations that a client can perform by invoking API function $\texttt{api}_i$. However, we would like to identify idiomatic resource manipulations. Each such idiom is a set of code patterns $FP=\{pat_1,\ldots,pat_m\}$ satisfying the following property: if one of the code patterns $pat_i \in FP$ appears in any valid execution trace of the server, then *all* the patterns in FP appear in that trace. Each such idiom is called a *fingerprint* and denotes a potential security-sensitive operation performed on the resource. Note that the above property implies that each fingerprint FP is such that $FP \subseteq$ *CodePats*($\texttt{api}_i$) or $FP \cap$ *CodePats*($\texttt{api}_i$)$=\emptyset$, for each API function $\texttt{api}_i$. As described below, we use concept analysis to identify a set of *candidate fingerprints*. Each candidate fingerprint may possibly contain multiple fingerprints, and must be refined to yield the actual fingerprints.

```
   Algorithm  : EXTRACT_CODE-PATTERNS(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 ∈ API.
1  foreach (function f in Server) do
2      Summary(f) := ∅;
3      foreach (statement s ∈ f that affects a data structure of type ∈ RSC) do
4          CP := Breakdown of s into code patterns (see Figure 2);
5          Summary(f) := Summary(f) ∪ CP;
6  foreach (api_i ∈ API) do
7      CodePats(api_i) := ∅;
8      foreach (function f reachable from api_i) do
9          CodePats(api_i) := CodePats(api_i) ∪ Summary(f);
10 return CodePats(api_1),...,CodePats(api_n)
```

Algorithm 1: Static analysis algorithm to extract resource manipulations.

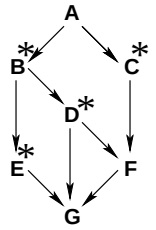
3.2 Background on concept analysis

Concept analysis is a well-known hierarchical clustering technique that has found use in software engineering [3, 8, 9, 21, 24, 25, 26, 27, 28]. We give a brief overview of concept analysis and describe how we adapt it to find candidate fingerprints.

(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 6. Concept analysis example.

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 6 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 6(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).

3.3 Using concept analysis

We compute candidate fingerprints using Algorithm 2. 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 6. It then finds candidate fingerprints, in lines 2-7, 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 candidate fingerprint.

For the example in Figure 6, 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 candidate fingerprint.

Intuitively, Algorithm 2 works because each fingerprint FP satisfies $FP \subseteq$ *CodePats*($\texttt{api}_i$) or $FP \cap$ *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 $pat_i \in FP$ is introduced will introduce *all* of the code patterns in FP. Line 7 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 fingerprint as *pat* appear in that node. Note however, that code patterns from other fingerprints may also be introduced in the same node. Thus, Algorithm 2 only computes candidate fingerprints: each candidate fingerprint may contain multiple fingerprints that must be obtained via refinement (in Step B).

```
   Algorithm  : FIND_CANDIDATE_FINGERPRINTS(CodePats,API)
   Input      : (i) CodePats: The relation obtained from Algorithm 1, and (ii) API= {api_1,...,api_n}, set of API functions of the server.
   Output     : CFP_1,...,CFP_k, a set of candidate fingerprints.
1  Run concept analysis with the set of instances I=API, the set of features F=∪_{i∈[1..n]} CodePats(api_i), and the relation R=CodePats;
2  count := 1;
3  foreach (node ⟨X, Y⟩ in the concept lattice) do
4      Let {⟨X_j, Y_j⟩} be the set of parents of ⟨X, Y⟩ in the concept lattice;
5      Diff := Y - ∪_j Y_j;
6      if (Diff ≠ ∅) then
7          CFP_count := Diff; count := count + 1; Mark the node ⟨X, Y⟩;
8  return CFP_1,...,CFP_count /* Note: k is the value of count in this line. */
```

Algorithm 2: Finding candidate fingerprints.

It can be shown that the number of candidate fingerprints identified by Algorithm 2 has an upper bound of $|\cup_{i\in[1..n]}$ *CodePats*(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 7 of Algorithm 2. This is key, because these nodes introduce candidate fingerprints, and as discussed in Section 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.* [16] because it has been shown to work well in practice [3]. While this algorithm is asymptotically exponential—its complexity is $O(2^{2p}|I|)$, where p is an upper bound on the number of features of any instance in I—the algorithm scaled well in our case studies.

4 Refining fingerprints with constraints

As described in Section 2.2, candidate fingerprints 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 are not part of the same fingerprint may appear in the same candidate fingerprint. Second, the resource manipulations in a candidate fingerprint may be associated with multiple, possibly unrelated resource instances. Thus, candidate fingerprints 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 candidate fingerprints (Step B of our approach). Both precision constraints and domain-specific constraints can be expressed in this framework.

As Figure 7 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 candidate fingerprints by separating code pattern sets X and Y into separate fingerprints. *Ignore*(X) refines candidate fingerprints by discarding the code pattern set X from candidate fingerprints. *Combine*(X, Y), for which we have only felt occasional need, combines code pattern sets X and Y in two candidate fingerprints into a single fingerprint, thus coarsening the results of concept analysis. For example, the constraint *Separate*({1,2,3,4}, {5,6}) refines the candidate fingerprint in Figure 3 to yield the fingerprints in Figure 5. We now discuss precision and domain-specific constraints in this framework.

Constraint	:=	*Separate*(PatSet, PatSet) | *Ignore*(PatSet)
		| *Combine*(PatSet, PatSet)
PatSet	:=	Set of code patterns (as defined in Figure 2)

Figure 7. Grammar for constraints.

Benchmark	LOC	Analysis time (secs)	Concept lattice # Nodes	Concept lattice # Edges	Num. of cand. fings.	Avg. size of cand. fings.	Refinement 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%)

Figure 8. Results for each of our case studies. Concept lattices are also available online [1].

Precision constraints are *Separate*(*X*, *Y*) constraints and as discussed in Section 2, they serve two goals. The first goal is to refine candidate fingerprints 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 3, 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 4), 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 candidate fingerprint, 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 candidate fingerprints 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 candidate fingerprint above.

Note that precision constraints are not necessary if more precise program analysis is employed. Algorithm 1 currently lacks flow-sensitivity and data-flow information that can potentially avoid the imprecisions reported above. However, in each of our case studies we needed precision constraints for fewer than 50% of the candidate fingerprints mined—9/18 for ext2, 24/115 for X server, and 4/38 for PennMUSH. Thus, we believe that our current approach strikes a good balance between simplicity and precision of candidate fingerprints.

Domain-specific constraints encode domain knowledge to further refine fingerprints. 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 from the set of fingerprints. 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 fingerprints mined by concept analysis are at too fine a granularity. For example, in PennMUSH, we found that 30 of the 38 candidate fingerprints 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.

5 Case studies

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 approach using four criteria. First, we measured the number and size of candidate fingerprints extracted from source code. Because an analyst must examine these candidate fingerprints to identify security-sensitive operations, these metrics indicate the amount of manual effort needed to supplement our approach. Note that without our approach, the analyst must examine the *entire* code base to find security-sensitive operations. Second, we measured the number of candidate fingerprints that had to be refined with constraints. This metric shows the effect of imprecise static analysis and the effort needed to refine candidate fingerprints. 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 approach with security-sensitive operations that were identified independently for these servers [20, 30].

Figure 8 presents statistics on the time taken by the analysis and the size of concept lattices produced. It also shows the number and size of candidate fingerprints and the number of candidate fingerprints that needed refinement. As these results show, our analysis is effective at distilling several thousand lines of code into concept lattices of manageable size (see [1]). There were under 115 candidate fingerprints of average size under 4 across all our benchmarks, fewer than 50% of which had to be refined. Identifying security-sensitive operations reduces to refining and interpreting these candidate fingerprints, 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 Figure 8 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 5.1-5.3 present each case study in detail, including our experience interpreting fingerprints and correlating these fingerprints against independently identified security-sensitive operations.

5.1 The ext2 file system

As discussed in Section 2, we focused on how directories are manipulated by the ext2 file system. Concept analysis produced 18 candidate fingerprints containing an average of 3.67 code patterns, of which we had to refine 9 with precision constraints. We then determined the resource manipulation embodied by each fingerprint and tried to associate it with a security-sensitive operation. Section 2 presented two such examples. Two more examples are discussed below.

1. The fingerprint {*Write* `0` *To* `inode->i_blocks`, *Write* `1` *To* `inode->u->ext2_inode_info->i_new_inode`, *Write* `4096` *To* `inode->i_blksize`} 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 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.

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

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, that 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 [7, 10, 20].

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 candidate fingerprints 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. The fingerprint {*Write* `20` *To* `xEvent->u->type`, *Write* $\perp$ *To* `xEvent->u->mapRequest->window`}, 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 [20], 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 Map and Unmap operations on a `Window`, respectively.

We had previously identified fingerprints for 11 security-sensitive operations on the `Window` resource [15]. However, as discussed in Section 1, that work used dynamic analysis, and could only identify fingerprints along paths exercised by manually-chosen test inputs to the server. Further, that work 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 from prior work at the granularity of code patterns, but did so automatically.

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, that we used as the API of PennMUSH. Most of these API functions modify a database of objects. Thus, we tracked how the PennMUSH API manipulates resources of type `object`. Concept analysis produced 38 candidate fingerprints. Most of them had only one or two code patterns, so we only had to refine 4 of these candidate fingerprints 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 candidate fingerprints extracted by concept analysis (as compared to X server).

6 Limitations

An important limitation of our approach is that it cannot guarantee that all fingerprints have been mined. Our approach 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. This is because the static analyzer can potentially miss resource manipulations. For example, if a C `struct` representing a shared resource is read from/written to using pointer arithmetic, the analysis described in Section 3.1 will miss this resource access, thus leading to a missed (or erroneous) fingerprint. Further research is necessary to develop a provably complete approach to fingerprint-finding.

A second limitation of our approach is that it currently constrains fingerprints to be conjunctions of code patterns; temporal relationships between code patterns cannot be mined by our approach. As a result, interpretation of some of the fingerprints identified by the approach was tedious and time-consuming. To overcome this limitation, we plan to explore a more expressive fingerprint language (*e.g.,* automata over an alphabet of code patterns) and algorithms to extract such fingerprints.

7 Related work

This paper overcomes two important shortcomings that we had identified in prior work [15]. The need for an *a priori* description of security-sensitive operations hindered the application of the techniques developed there to a wide variety of servers. Further, a dynamic trace-based approach to fingerprint-finding meant that only code paths exercised by test inputs to the legacy server would be analyzed, thus leaving large portions of the legacy server unanalyzed.

As discussed in Section 1, our approach follows the aspect-oriented paradigm. Several other tools, such as PoET/PSLang [11], Naccio [12], Polymer [6] and our own prior work on Tahoe [14] also follow an aspect-oriented approach to enforce authorization policies on legacy code. In all these tools, a security analyst provides a description of locations to be protected (join points) as well as the policy check at each location (advice). These tools then weave calls to a reference monitor at each of these locations. However, when legacy servers manage their own resources, identifying locations where policy checks must be weaved becomes a challenge. The techniques developed in this paper can benefit the above tools by reducing the manual effort involved in identifying locations for reference monitoring, as well as the advice to be integrated at these locations.

Concept analysis has previously been used in software engineering, including aspect mining (Ceccato *et al.* present a survey of such techniques [8]) and software modularization. For example, concept analysis has been

used on identifier names to find methods and classes that implement similar functionality [27]. Dynamic analysis in conjunction with concept analysis has been used to find methods that implement a particular feature [9, 26]. 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. Concept analysis has also found use to identify modular structure in legacy programs [21, 24, 25, 28]. The modular structure so identified can be used to refactor legacy software (*e.g.,* convert non-object-oriented programs into object-oriented ones [24]). Another recent use of concept analysis is in the context of debugging mined specifications [3]. Automatically mined temporal specifications may often be buggy, and the problem here is for an analyst to classify each mined specification as correct or buggy. Similar traces can be clustered using concept analysis, so the analyst can decide en-masse whether an entire cluster is buggy.

8 Summary

We presented an approach to reduce the manual effort involved in mining security-sensitive operations in legacy servers. Our approach uses concept analysis to mine fingerprints, which are code-level descriptions of security-sensitive behavior. Our experiments with three complex real-world servers show that our approach is efficient and effective at finding security-sensitive operations.

Acknowledgments. We thank the anonymous reviewers for their insightful comments.

References

[1] `http://www.cs.wisc.edu/~vg/papers/icse2007`.
[2] PennMUSH multi-user dungeon. `http://pennmush.org`.
[3] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus. Debugging temporal specifications with concept analysis. In *PLDI*, 2003.
[4] J. P. Anderson. Computer security technology planning study, volume II. Technical Report ESD-TR-73-51, Deputy for Command & Mgmt. Systems, Bedford, MA, 1972.
[5] Aspect-oriented software development glossary. `http://aosd.net/wiki/index.php?title=Glossary`.
[6] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *PLDI*, 2005.
[7] J. Berger, J. Picciotto, J. Woodward, and P. Cummings. Compartmented mode workstation: Prototype highlights. *IEEE TSE*, 16(6), 1990.
[8] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. A quantitative comparison of three aspect mining techniques. In *13th Wkshp. Prog. Comprehension*, 2005.
[9] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE TSE*, 29(3), 2003.
[10] 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. *Jour. Computer Security*, 2(2-3), 1993.
[11] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell, 2004.
[12] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symp. on Security & Privacy*, 1999.
[13] B. Fletcher. Case study: Open source and commercial applications in a Java-based SELinux cross-domain solution. In *2nd Security-enhanced Linux Symp.*, 2006.
[14] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux security modules framework. In *ACM Conf. Comp. & Comm. Security*, 2005.
[15] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *IEEE Symp. on Security & Privacy*, 2006.
[16] R. Godin, R. Missaoui, and H. Alaoui. Incremental concept formation algorithms based on Galois (concept) lattices. *Computational Intelligence*, 11(2), 1995.
[17] M. Hocking, K. Macmillan, and D. Shankar. Case study: Enhancing IBM Websphere with SELinux. In *2nd Security-enhanced Linux Symp.*, 2006.

[18] T. Jaeger, A. Edwards, and X. Zhang. Consistency analysis of authorization hook placement in the Linux security modules framework. *ACM TISSEC*, 7(2), 2004.
[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, 1997.
[20] D. Kilpatrick, W. Salamon, and C. Vance. Securing the X Window system with SELinux. Technical Report 03-006, NAI Labs, 2003.
[21] C. Lindig and G. Snelting. Assesing modular structure of legacy code based on mathematical concept analysis. In *ICSE*, 1997.
[22] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX Annual Technical Conf. (FREENIX)*, 2001.
[23] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conf. Compiler Construction*, 2002.
[24] M. Siff. *Techniques for Software Renovation*. PhD thesis, University of Wisconsin-Madison, 1998.
[25] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *ACM SIGSOFT FSE*, 1998.
[26] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *11th Conf. on Reverse Engineering*, 2004.
[27] T. Tourwe and K. Mens. Mining aspectual views using formal concept analysis. In *4th Wkshp. on Source code Analysis & Manipulation*, 2004.
[28] A. van Duersen and T. Kuipers. Identifying objects using cluster and concept analysis. In *ICSE*, 1999.
[29] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. *Ordered Sets*, 1982.
[30] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symp.*, 2002.
[31] The X11 Server, version X11R6.8 (X.Org Foundation).
[32] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *USENIX Security Symp.*, 2002.