Creating and Operating Security-Aware Code

Ph.D. Thesis Proposal

Vinod Ganapathy
Computer Sciences Department
University of Wisconsin, Madison
Madison, WI-53706
vg@cs.wisc.edu

Proposal date & time: 9:00AM on Friday, January 20, 2006

1 Motivation and Thesis Statement

A central problem in systems with shared resources is to protect these resources from unauthorized access. Access to these resources must be mediated so that the system's *security goals* are not violated. For example, a common security goal on most UNIX-like systems is to protect the confidentiality and integrity of sensitive files, such as `/etc/passwd`, and configuration files and log files of critical services. Thus, security goals outline high-level security requirements of a system. Popular security goals that have appeared in the literature include Biba integrity [10], Clark-Wilson integrity [15], and Bell-LaPadula confidentiality [9]

A popular way to meet security goals is by formulating and enforcing appropriate *authorization policies*, also referred to in the literature as access control policies [38, 55]. These policies govern the set of *security-sensitive operations*, that a set of *subjects*, typically users and processes, can perform on *objects*, which are shared resources on the system. Often the set of subjects and objects is not disjoint, for *e.g.*, a process may be classified as both a subject and an object when the security-sensitive operation in question is inter-process communication. An appropriately formulated authorization policy achieves system security goals via *enforcement*. One tool used for authorization policy enforcement is the concept of a reference monitor [3]. A reference monitor is an entity which observes execution of a program, and decides whether program execution is in accordance with an authorization policy. Most UNIX-like systems implement reference monitors to enforce discretionary access control (DAC [20, 38]), using access control lists [55].

Meeting more sophisticated end-to-end system security goals, such as information flow goals, requires more expressive policy languages and enforcement mechanisms, such as the ability to enforce mandatory access control (MAC [9, 10]). These policies were initially conceived for use in the military to implement multi-level security (MLS) goals, such as Biba integrity and Bell-LaPadula confidentiality. However, they are gaining popularity and acceptance in commodity operating systems. For example, security-enhanced Linux (SELinux [60, 61, 82]), which enables enforcement of authorization policies to meet security goals such as MLS, is now part of the Linux kernel, and is shipped and supported by several vendors (*e.g.*, Fedora Core [29] and hardened Gentoo [36]). Because MAC policies are centralized and system-wide, as opposed to DAC policies, which are based upon resource ownership, they can mitigate the threat posed by malware (*e.g.*, trojan horses, viruses and worms) and malicious insiders by restricting the damage an attack can cause [64].

Enforcement of end-to-end security goals requires creating *security-aware code*. By security-aware code, I mean code which is aware of, and has mechanisms to enforce security policies, specifically authorization policies, for the purpose of this proposal. Traditionally, security-aware code has been restricted to operating system code, which centrally manages authorization for the entire system, and certain specialized applications, such as database servers.

Definition 1 (Security-Aware Code) *Code which has mechanisms to enforce security policies, in particular, authorization policies, is called security-aware code.*

I argue that for effective enforcement of end-to-end system security goals, *security-aware code must not be restricted to operating system code alone, and must encompass all code which manages shared resources*. These include server applications which manage multiple clients simultaneously (database servers are just one example of such servers), middleware, and hypervisors (*e.g.*, Xen [7, 75]). The need to expand the security-aware code base to include entities other than the operating system is best demonstrated using a simple example.

Consider the X Window system. It is architected using a client/server model, where the X Window server, called X server [89], accepts connections from X clients, such as `xterms`, and displays windows on the console. The X server is responsible for bitmapped display, and handling client connections; X clients communicate with the X server via the X protocol. In the absence of authorization policy enforcement on client interaction, several well-documented attacks [25, 53, 87] are possible:

1. An X client can arbitrarily control and manipulate the contents of windows belonging to other X clients (violating integrity).
2. An X client can read the contents displayed on another X client window by simply reading the bitmapped display (violating confidentiality).
3. An X client can repeatedly grab control of the input stream, such as mouse and keyboard events, thus preventing other clients from getting access to these system resources (affecting availability).

The situation is akin to an operating system without access control, where any user can read from, and write to, files belonging to other users. Another attack serves to demonstrate that end-to-end enforcement of security goals cannot be achieved if security-aware code only includes the operating system.

Consider the X server running on an operating system, such as SELinux, which is capable of enforcing policies to meet multi-level security (MLS), such as Biba integrity and Bell-LaPadula confidentiality. On such an operating system, users, and thus resources and X clients belonging to these users will be associated with *security labels*, such as *Top-secret*, or *Unclassified*. The authorization policy, whose enforcement is meant to meet these end-to-end security goals, is expressed in terms of the security-labels of subjects and objects on this system. Consider an authorization policy (enforced by the operating system) which prevents *Unclassified* users from reading files belonging to *Top-secret* users (akin to Bell-LaPadula confidentiality). Note that the operating system is powerful-enough to enforce this policy even in the presence of a malicious *Top-secret* insider, *i.e.*, the insider cannot violate system security using operating system primitives alone.

However, a malicious *Top-secret* user can use the X server to violate end-to-end security as follows: he simultaneously opens a *Top-secret* `xterm` and a *Unclassified* `xterm` (with a colluding *Unclassified* user). A "cut" operation from the *Top-secret* window, followed by a "paste" operation to the *Unclassified* window violates end-to-end system security.

The reader may ask why the existing policy enforcement in the security-enhanced operating system, upon which the X server runs, is insufficient to enforce authorization policies on the X clients—after all, the "cut" and "paste" operations have kernel footprints, and thus, the authorization policy must be enforceable within the operating system. While this is certainly true, it must be noted that "cut" and "paste" are

X server-specific channels of communication, and are thus more readily visible within the X server (than they are within the operating system) 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 the 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 which will be affected by the requested operation. It has been argued that this is impractical [53].

The root of the problem is that in applications such as servers, which manage their own shared resources (for example, Windows, Fonts, buffers, etc., implemented within the X server), clients use operating system resources, such as files, and often store information related to these operating system resources on shared resources of the server (*e.g.*, the cut-buffer in the X server). If the server is not security-aware, and does not offer mechanisms to enforce authorization policies on client interaction, then end-to-end system goals cannot be met. In summary, I make the following claim:

CLAIM

All code which manages shared resources must be made security-aware for effectively meeting end-to-end security goals.

1.1 Existing Techniques For Creating and Operating Security-Aware Code

A natural question to ask is: what techniques exist to create security-aware code? A possible technique is to *design for security*, *i.e.*, *proactively* integrate security as a core design component, along with functionality and performance. Unfortunately, this is rarely followed in practice outside the design of operating system code, the notable exception being database servers, where authorization is considered paramount, and is integrated into the design of the server. However, even with these systems, changes to built-in security mechanisms are hard to make for two reasons.

1. **Additional security may be needed in the future.** For example, while Linux provides access-control lists to enforce DAC, the need was felt to incorporate mechanisms to enforce MAC policies, to meet sophisticated end-to-end security goals. Further, the need was also felt for mechanisms to enforce authorization policies on other shared resources, such as socket-based communication and shared memory, resources for which Linux provided no mechanisms to enforce authorization policies. Thus, the Linux kernel had to be retrofitted to enable enforcement of such policies.

   A second example is that of the X server, which has a primitive mechanism, called the X security extension [87], to enforce authorization policies. It statically partitions clients into *Trusted*, and *Untrusted*, and enforces policies on interactions between these two classes of clients. However, this framework is not powerful enough to enforce arbitrary authorization policies when multiple clients connect to the X server. 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 (*i.e.*, either *Trusted* or *Untrusted*), and will not enforce policies on the interaction between clients in the same category.
2. **Functional and performance enhancements may be needed in the future.** *Ad hoc* implementation techniques to add functionality and performance often break security assumptions, thus creating security vulnerabilities, as was shown by Karger and Schell in their security evaluation of Multics [52].

In each of the cases discussed above, legacy code (namely, the Linux kernel and the X server) was retrofitted manually. For example, in the case of the Linux kernel, this process required reasoning about the security-sensitive operations to be protected, and the locations where they were performed within the kernel.

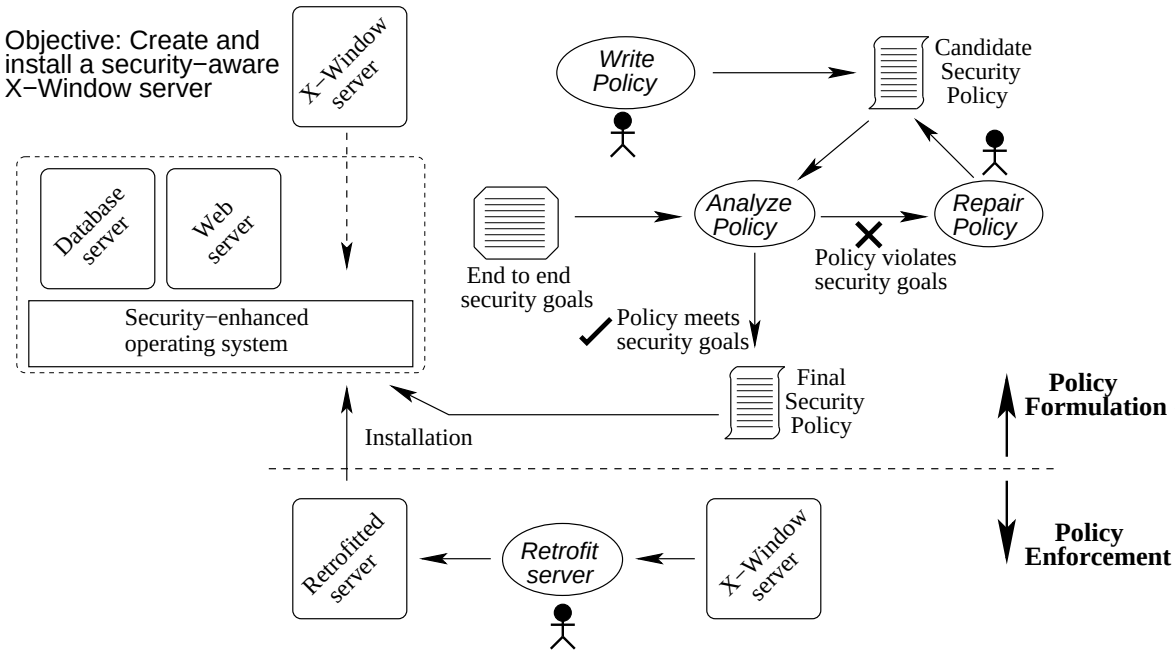

Figure 1: State-of-the-art in creating security-aware code from legacy code consists of several steps, shown in the ovals. This figure shows how a legacy server, in this case the X server, is made security-aware. Several steps are manual and *ad hoc*, as shown in the figure. My proposal seeks to largely automate the tasks of authorization policy formulation and enforcement.

Mechanisms to mediate security-sensitive operations with authorization policies were then introduced in the Linux Security Modules (LSM) framework [88], which incorporates a set of generic reference monitor calls (called *hooks*) at several locations in the kernel. The LSM framework is now part of the Linux distribution, and serves as the vehicle of choice for enforcing several policies, including SELinux [81].

The NSA has expressed interest in securing several user-space server applications, including the X server, and making them capable of enforcing authorization policies, using an architecture similar to the LSM framework, namely, by placing hooks to a generic reference monitor at appropriate locations in the server code [53]. A retrofitted version of the X server was recently announced [80].

In each of these cases, the manual effort involved was significant. For example, development of the LSM framework took over two years—the LSM framework places around 200 hooks in the Linux kernel, and the size of the generic reference monitor is just about 21,000 lines of code. Similarly, while availability of a security-aware X server was announced recently, the NSA first expressed interest in retrofitting the X server in early 2003 [53]. A significant fraction of this time was spent on deciding where to place reference monitor hooks, and on writing the generic (*i.e.*, policy-independent) code of the reference monitor.

Proactive techniques for creating security-aware code are desirable, but they are not always used in practice for economic reasons—for example, it would have been impractical to redesign the Linux kernel or the X server to incorporate the additional mechanisms discussed above. Code producers typically focus on providing functionality and performance, which are tangible to customers, rather than security, which is typically an afterthought. Further, as argued above, even if proactive techniques are used, changes needed in the future may break certain assumptions, and the changed system may no longer be secure. Consequently,

there is a need for techniques to *retroactively create security-aware code from legacy code*.

Unfortunately, there is little prior work on systematic techniques to retrofit legacy-code to create security-aware code. Most research in this area has focused on finding and fixing existing vulnerabilities (*e.g.*, [23, 70, 74, 78]). *Almost no work exists on systematic techniques to retroactively add extra security*. Contrast this with the relatively rich body of literature on retroactively adding the other two facets of software design, namely functionality and performance (for *e.g.*, see [14, 45, 71, 73, 83]).

As shown in Figure 1, there are two components to creating security-aware code, namely *authorization policy formulation*, and *authorization policy enforcement*. State-of-the-art techniques for each of these components is largely manual, as described below.

1. **Authorization Policy Enforcement** is the task of retrofitting legacy code to make it security-aware, *i.e.*, capable of enforcing authorization policies. This is typically a policy-independent step, and involves modifying legacy code by adding calls to a generic reference monitor at appropriate locations. While Figure 1 shows this in the context of retrofitting an application, namely the X server, the same was also done to retrofit the Linux kernel (in the context of the LSM framework).

   To the best of my knowledge, there are no systematic techniques to retrofit legacy code to enable it to enforce authorization policies. The process currently followed is informal, manual and time-consuming. A developer typically comes up with a proposal to place a reference monitor call at a certain code location, and puts this proposal up for discussion with other developers. A discussion on the pros and cons of placing the hook (security implications, performance implications) follows, and the proposal is accepted if there is consensus. No tools or techniques are currently available for evaluating the impact (security or otherwise) of the placement.

2. **Authorization Policy Formulation** is the task of writing an authorization policy that will be enforced by security-aware code, so that the end-to-end security goals of the system are satisfied by the policy. As Figure 1 shows, this is done in several steps. In the following, I describe how authorization policies are written for the Linux kernel, *i.e.*, authorization policies enforced by the Linux kernel on applications. Nearly the same set of steps are used to write policies for other security-aware code, for *e.g.*, authorization policies enforced by the X server on X clients.

   - **Write policy.** A security administrator first writes a candidate security policy, typically using *ad hoc* techniques. For example, a popular technique is to run the application under a null policy, *i.e.*, one that denies the application all access to shared resources, and collect an audit log of access violations that are generated [62, 79]. The hope is that this audit log contains all the accesses to shared resources that it needs for normal operation. Each statement in the audit log is carefully examined, and converted into a policy statement. Tools such as `audit2allow` (in the SELinux policy development toolkit [85]) enable automatic conversion of audit logs to authorization policies.
   - **Analyze policy.** The resulting authorization policy is then checked against end-to-end system security goals. Policies are typically complex, and manual analysis of policies is infeasible. Several policy analysis tools, that check for different kinds of end-to-end system security goals, have been developed to ease the task of the policy writer. For example, SLAT [40] uses model checking to check that SELinux policies satisfy information-flow goals. Similarly Gokyo [48, 50] checks that SELinux policies satisfy Biba integrity [10]. If the policy is found to violate these system security goals, these tools typically also inform the policy writer about the violation. SLAT does so by reporting counter-examples produced by the model checker, while Gokyo does so by reporting conflicts between the policy and Biba integrity.
   - **Repair policy.** The policy writer examines the violations produced by the policy analysis tool, and

repairs the policy to remove the reported violations. Often, the violation may be because the end-to-end security goal is too strict. For example, consider enforcing Biba integrity, which requires a *Top-secret* process to only read from *Top-secret* resources (or resources classified higher than *Top-secret*). This is unachievable in practice, for example, with applications like OpenSSH, which *must* read, perhaps untrusted (*e.g., Unclassified*), input from the network in order to remain functional. In such cases, if deemed appropriate, the end-to-end goals are appropriately relaxed. Recently, some techniques have been developed to aid policy repair [49]. However, they have so far only been applied to a limited class of end-to-end security goals (namely Biba integrity), and there is need for further research in this area.

Existing techniques, described above, are unsatisfactory, for two reasons:

1. **They do not scale.** As I argued earlier, for effectively meeting end-to-end security goals, security-aware code must encompass all code that manages shared resources. These include hypervisors, operating system code, and server code. Security-aware code will thus be present at all layers of the software stack. Further, new servers will continue to appear, providing new functionality. Requiring manual intervention to (i) retrofit each of these bodies of code for authorization policy enforcement, and (ii) write authorization policies for each security-aware application is impractical, does not scale, and if not automated, will be an administrative nightmare.
2. **They are error-prone.** As I described earlier, the process of retrofitting legacy code is largely informal. This is worrisome, because the code being added affects the security of the application being retrofitted—any mistakes in this process will result in improper authorization policy enforcement, thus resulting in security holes. Not surprisingly, researchers have found security holes in the placement of reference monitor hooks in the LSM framework [47, 93]. In particular, they found that the design of the reference monitor hook interface left room for time-of-check to time-of-use (TOCTTOU) bugs [11], and that the placement of these hooks in the Linux kernel violated the complete mediation principle [76].

   Similarly, the process of formulating authorization policies for security-aware code is repetitive and error-prone. If the same end-to-end security goals are to be achieved by two pieces of security-aware code, such as the operating system and a security-aware user-space server, then it is desirable to have techniques that enable *reuse* of policies written for one piece of security-aware code as policies for the other piece of security-aware code. Moreover, the authorization policies formulated using existing techniques may be incomplete. This is because they are based upon running the application: as a result, statements that govern resource accesses by the application in code branches that are not exercised during runtime will not be in the policy.

I argue that retroactive security will become more important as the body of legacy code without security mechanisms continues to increase, and that research into systematic techniques for (i) retrofitting legacy code to create security-aware code, and (ii) writing appropriate authorization policies for security-aware code must be initiated. I will address the need for such systematic techniques in my dissertation. Specifically, my thesis addresses the following problem:

PROBLEM STATEMENT

Devise techniques to automate authorization policy formulation and enforcement to ease the creation of security-aware code from legacy code.

1.2 Thesis Proposal

My proposal seeks to help the creation and operation of security-aware code by providing automated tools for authorization policy formulation and enforcement. I intend to do this by building tools to retrofit legacy

code with appropriately inserted calls to a reference monitor [3], and to write appropriate authorization policies for the reference monitor to enforce. Note that there are two facets to this work:

1. **Policy Formulation**, namely the task of writing an appropriate authorization policy, $\mathcal{A}$, one that ensures that end-to-end system security goals are not violated.
2. **Policy Enforcement**, namely the task of modifying a legacy software package, $\mathcal{X}$, so that the modified software can enforce the authorization policy $\mathcal{A}$ when its clients request access to shared resources that it maintains. To do so, requires statically inserting calls to a reference monitor (which encapsulates the policy $\mathcal{A}$) at each location in $\mathcal{X}$ where shared resources are accessed.

I will address both facets in my work. My thesis is the following:

THESIS STATEMENT

Program analysis techniques can largely automate the process of:

- formulating authorization policies for security-aware code.
- retrofitting legacy code for authorization policy enforcement.

In the rest of this document, I describe how I believe program analysis can help with the above mentioned tasks. I will describe steps that I have already pursued, and the steps that I intend to pursue in support of my thesis.

1.3 Outline of Tasks to Support my Thesis

To describe my thesis in more detail, and how I intend to support it, in this section, I present a list of tasks that I will address in my dissertation. Some formal definitions are in order before I can describe my contributions.

Formally, an authorization policy, $\mathcal{A}$, 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*, which denote the equivalence class to which they belong. 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, $\mathcal{M}$, is defined as $\langle \Sigma, \mathcal{S}, u, f \rangle$, where Σ is a set of *security events*, $\mathcal{S}$ is the state of $\mathcal{M}$, u: $\Sigma \times \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is a state transformer, and f: $\Sigma \times \mathcal{S} \times \mathcal{A} \rightarrow$ Bool is a *policy consulter*. For authorization policies, Σ is a set of triples of the form $\langle sub, obj, op \rangle$; $\mathcal{S}$ is a set storing the security-labels associated with each subject and object tracked by $\mathcal{M}$, and u is a set of rules denoting how subject and object security-labels may change in response to policy decisions. An *enforcer* $\mathcal{E}$, capable of controlling the execution of $\mathcal{X}$, observes events in Σ generated by $\mathcal{X}$, and passes them on to $\mathcal{M}$. Any violations of the policy $\mathcal{A}$, will result in f returning False, following which $\mathcal{E}$ will take appropriate action. To enable a legacy software package $\mathcal{X}$ ensure that end-to-end security goals are not violated in accesses to its shared resources, secure implementations of $\mathcal{E}$ and $\mathcal{M}$ are necessary. Further, the policy $\mathcal{A}$ must be appropriately chosen as to meet security goals.

1.3.1 Implementing $\mathcal{E}$

The enforcer $\mathcal{E}$ must have the ability to monitor all security events generated by $\mathcal{X}$, and have the ability to take appropriate action if a request for access to a shared resource in $\mathcal{X}$ is found to violate $\mathcal{A}$. The challenge here is to expose the set of security events generated by $\mathcal{X}$ to the enforcer. This is easy if the security events

are easily observable, such as system calls. However, access to shared resources of a user-level application can be made without invoking a system call. Thus, to enable authorization policy enforcement of user-level applications, such as the X server, techniques must be designed to identify locations in the code of $\mathcal{X}$ which are security-sensitive, *i.e.*, where access to shared resources may be made. The security event generated at a location where an operation op is potentially performed is then $\langle sub, obj, op \rangle$, where sub is the subject requesting the operation, and obj is the shared resource upon which the operation is to be performed.

Task 1:	Identify and expose locations in the code of $\mathcal{X}$ where security-sensitive operations, namely those that are governed by $\mathcal{A}$, are performed.
Status:	Completed.
Result:	AID, a tool to find root-causes of security-sensitive operations.

In my work to-date to validate my thesis on authorization policy enforcement, I have worked with two large, real-world systems, namely Linux, and the X server. As an initial case study, to explore the use of program analysis to retrofit legacy code, I investigated static analysis algorithms to place authorization hooks within the Linux kernel to the SELinux reference monitor (see [33]). This case study was appealing because of two reasons: (i) authorization hooks had been placed manually in the Linux kernel to the same reference monitor, in the context of the LSM framework, and (ii) placement of these authorization hooks had been extensively verified; so there is reason to believe that the placement is largely correct. The purpose of this case study was twofold: (i) Develop an algorithm to automatically infer locations where a large legacy application performs security-sensitive operations, and (ii) Evaluate the effectiveness of this algorithm. The task of evaluating the effectiveness of the algorithm is made easy because of the availability of manually-placed authorization hooks.

This exercise was a moderate success. I designed a static analysis algorithm to place hooks in the Linux kernel to the SELinux reference monitor. In designing the static analysis algorithm, I encountered the need for some manual input, similar to annotations used in several program analyses (*e.g.*, [24, 78]), to the static analysis algorithm, which enabled the analysis algorithm identify locations where security-sensitive operations are performed by the kernel. While these annotations were not hard to write, precision of the analysis results crucially depended on the precision of the manual input. In order for this technique to apply to other pieces of legacy code, *techniques were needed to eliminate, or at least ameliorate the effort of producing this input needed by the algorithm.*

I have addressed these needs by designing, implementing, and evaluating, AID. AID is a novel root-cause-finding tool, which uses dynamic program analysis to largely automate the process of generating the input needed by the static analysis algorithm (I describe this more precisely in Section 2).

1.3.2 Implementing $\mathcal{M}$

The Linux study was a feasibility study—my goal there was to reconstruct, using automated tools, the manual placement of authorization hooks to the SELinux reference monitor. The static analysis algorithm (refered to above) that I designed for that purpose was a matchmaking algorithm, which matched reference monitor functions in the SELinux reference monitor to locations where they must be invoked from the reference monitor. I soon realized that reference monitor implementations, like the SELinux reference monitor, will be unavailable for other bodies of legacy code. Thus, I saw the need to automate, as much as possible, the construction of the reference monitor.

Task 2:	Automate construction of the reference monitor.
Status:	Mostly Completed.
Result:	ALPEN, a tool that generates a template reference monitor.

I have largely addressed this challenge by designing and implementing ALPEN, a tool which largely automates reference monitor construction (described in detail in Section 2). The reference monitor generated by ALPEN is policy-independent. In future work, I intend to modify ALPEN to generate reference monitors capable of enforcing authorization policies written in the SELinux policy language, using the recently released user-space policy management server [84].

Together, AID and ALPEN empower a developer with automated tools to reason about, and retrofit legacy code for authorization policy enforcement. I have used AID and ALPEN to retrofit the X server and enable it to enforce authorization policies on X clients [34].

However, there is an important limitation in my work so far. My algorithms to retrofit legacy code to enforce authorization policies require as input a set of security-sensitive operations, *i.e.,* the set of accesses to shared resources that must be protected. A set of security-sensitive operations was derived manually by the NSA for both the Linux kernel as well as for the X server—I used these set of security-sensitive operations as inputs to my algorithm. If my work is to extend beyond these two applications, I need to develop a technique to identify shared resources in a legacy application, and identify the set of primitive operations on them that must be protected.

Task 3:	Identify the set of shared resources, and primitive operations on these resources, in legacy code, access to which must be mediated by an authorization policy.
Status:	To be addressed.
Proposal:	See Section 2.3 for proposed approach.

1.3.3 Writing $\mathcal{A}$

My work to date has largely ignored the question of choosing an appropriate policy to enforce. Because a reference monitor is only as good as the policy it enforces, it is important to write authorization policies that meet system security goals. That is, the policy enforced must be *secure*.

Secure policies are easy to write—a policy that denies all access to shared resources is secure. The challenge is to ensure that the policy is also *functional, i.e.,* it must allow useful functionality. For example, a policy that meets Biba integrity goals [10], enforced by the X server, will disallow "cut"s from an *Unclassified* window to be "pasted" into *Top-secret* windows, but will allow all other "cut" and "paste" operations. I propose to design techniques to enable authorization policy formulation for security-aware code with the aim of producing secure and functional policies.

Task 4:	Design tools and techniques to help write *secure* and *functional* authorization policies.
Status:	To be addressed.
Proposal:	See Section 3 for proposed approach.

1.4 Timeline

I intend to complete tasks 1-4 by Spring 2007. Figure 2 presents a timeline that I will follow.

1.5 Threat Model and Scope of Work

In pursuing the above mentioned tasks, I will make several simplifying assumptions. I state them here.

1. **Availability of source code.** I will assume that the source of the legacy application, $\mathcal{X}$, is available. Several key steps of my work depend on the ability to analyze $\mathcal{X}$. Analyzing binary executables is harder

Time	Milestones	Paper Submission Goals
October 2004	Project started	
Spring 2005	Authorization policy enforcement for Linux	
May 2005		ACM CCS 2005 (see [33])
Summer 2005 Fall 2005	Authorization policy enforcement for general-purpose applications (Tasks 1 & 2)	
November 2005		IEEE S&P 2006 (see [34])
Fall 2005	AID++: Finding security-sensitive operations	
January 2006		Thesis Proposal
Spring 2006	AID++ Formalization and Case Studies (Task 3)	
May 2006		ACM SIGSOFT FSE 2006
Summer 2006	Policy formulation for security-aware code (Task 4)	
November 2006		IEEE S&P 2007
Winter 2007	Policy formulation followup work (Task 4)	
Spring 2007	Write dissertation and search for job	
May 2007		ACM CCS 2007
June 2007	Finish dissertation and defend	

Figure 2: Proposed timeline for completing dissertation.

than analyzing source code, and I do not want to lose focus by concerning myself with the issues that arise when analyzing binary executables. Once the algorithms and techniques for producing security-aware code are mature, they can be transitioned to work with binary executables—there are no fundamental limitations that prevent this.

2. **Legacy code is not malicious.** The legacy code to be retrofitted, $\mathcal{X}$, itself should not be adversarial, *i.e.*, it should not written with malicious intent, and must not actively try to defeat retroactive instrumentation. Thus, I will assume that $\mathcal{X}$ does not automatically remove, or modify the instrumentation. This can be ensured by the operating system as it loads $\mathcal{X}$ for execution, by comparing a hash of the executable against a precomputed value. This also entails that $\mathcal{X}$ be non-self-modifying, thus precluding the possibility that the instrumentation is modified at runtime. Note that this property can be enforced as well, by making code pages write-protected.
3. **Covert channels.** The goal of my work is to retrofit legacy code to enable enforcement of authorization policies on *explicit* ways to access shared resources. The presence of covert channels which allow partial access to shared resources cannot be ruled out. Covert channels are beyond the scope of my work, and I will not address them in my dissertation.
4. **Security vulnerabilities are detected appropriately.** Secure code is a prerequisite to create security-aware code. Existing vulnerabilities, such as buffer-overflow vulnerabilities in the legacy code, could possibly be exploited by malicious hackers to bypass the instrumentation. Such cases are currently beyond the scope of my techniques. While I cannot hope to eliminate these vulnerabilities statically, one way to defend against them is to protect $\mathcal{X}$ using techniques such as CCured [70], Cyclone [51], or runtime execution monitoring (*e.g.*, [1, 26, 30, 32, 59, 77, 86]), which terminate execution when the behavior of $\mathcal{X}$ differs from its expected behavior.
5. **Cooperation from the environment.** The environment that $\mathcal{X}$ runs in cooperates with $\mathcal{X}$ to enforce authorization policies, and is not malicious in intent. For example, a retrofitted user-space server $\mathcal{X}$ relies on the operating system for several policy enforcement tasks. First, it requires that operating system ensure that the policy $\mathcal{A}$ is tamper-proof. Second, because clients typically connect to the server via

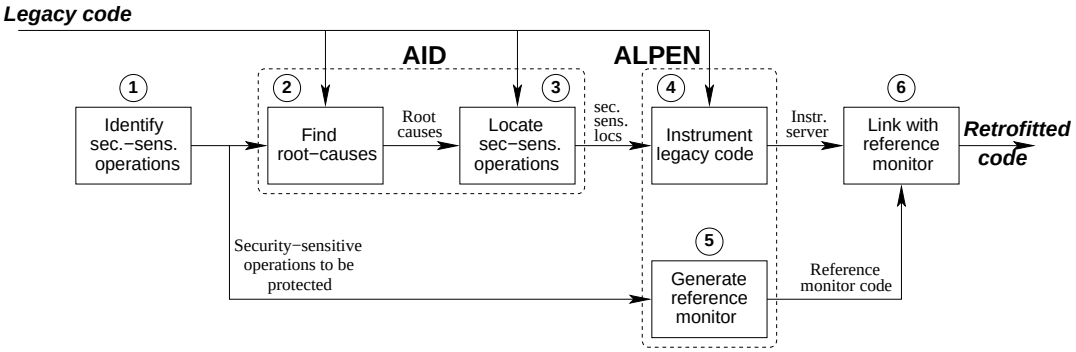

Figure 3: Steps involved in retrofitting legacy code for authorization policy enforcement.

the operating system, the server relies on the operating system for important information, such as the security-labels associated with the clients.

Under this model, it suffices to include in the trusted computing base, the lowest layer of software upon which other code runs. For example, in a traditional system, this is the operating system; more recently, with the emergence of virtual machines, this role has shifted to the hypervisor. This software layer bootstraps security by checking that retrofitted code running on it has not been tampered with. It also manages client connections to the code managing shared resources, and can thus bootstrap their security-labels. Note that clients themselves are not trusted, and can be malicious in intent—the goal of retrofitting code that manages shared resources is to ensure that client accesses to shared resources are mediated by an authorization policy.

2 Authorization Policy Enforcement

In this section, I demonstrate the use of program analysis for authorization policy enforcement. I have designed techniques, and have implemented them in two tools AID and ALPEN, for authorization policy enforcement. I will begin with a high-level informal overview of the approach, and present details later. Figure 3 shows the steps involved in retrofitting legacy code for authorization policy enforcement. Where applicable, I will illustrate the technique using an example from the X server.

1. **Find security-sensitive operations to be protected.** The first step is to determine the shared resource that must be protected, and to determine the operations that can be performed on them. I call these *security-sensitive operations*. More precisely,

   Definition 2 (Security-sensitive operation) *Any operation which manipulates a data structure of the server $\mathcal{X}$, and must be mediated in order to meet end-to-end security goals is security-sensitive.*

   In my work so far, I have relied on security-sensitive operations that have manually been identified by others.

   For instance, about 500 security-sensitive operations were manually identified for the Linux kernel [81], and 59 security-sensitive operations were manually identified for the X server [53]. In the rest of this document, I will represent these operations using sans-serif font as Resource_Operation. For example, in the case of Linux, shared resources included files, directories, sockets, and so on, and the security-sensitive operations identified for Linux included File_Write, File_Read, File_Execute, Dir_Rmdir, Dir_Mkdir, Socket_Create and Socket_Listen, each with their intuitive meanings. Similarly, for the X server, shared

resources include Windows, Fonts and Drawables, and include security-sensitive operations such as Window_Create, Window_Map, and Window_Enumerate.

In the case of both the Linux kernel and the X server, a design team (at NSA) manually identified the set of security-sensitive operations. These security-sensitive operations are often only accompanied by an informal English-language description of their meaning (as in [81, 53]), and a precise definition is not given. This is worrisome, because the legacy code is later retrofitted manually to protect these security-sensitive operations. Without a formal description of security-sensitive operations, there can be no security-guarantees on the retrofitted code.

My work on AID is an attempt to alleviate this situation, and formalize security-sensitive operations in terms of the actual code patterns that perform the said security-sensitive operations on shared resources. While AID is parameterized on the set of security-sensitive operations, and additions or deletions from this set do not affect its algorithms, *the problem of identifying security-sensitive operations and shared resources in legacy code* is still open. I describe my proposal to solve this problem in Section 2.3—this is Task 3 from Section 1.3.

2. **Infer root-cause of security-sensitive operations.** The second step identifies the *root-cause* of each security-sensitive operation, as a set of canonical code-patterns which are executed when the legacy code performs a security-sensitive operation (the execution of these code-patterns is the root-cause of the operation). However, the association between each security-sensitive operation, and the code-patterns that are executed is not known *a priori*, and the goal of this step is to recover the association.

   Two key observations help with this goal. The first observation is that each security-sensitive operation is typically associated with a tangible side-effect. 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. Thus, if we induce the server to perform the tangible side-effect associated with a security-sensitive operation, and trace the server as we do so, the code-patterns that characterize the security-sensitive operation *must* be in the trace.

   However, program traces are typically long, and it is still challenging to identify the code-patterns that characterize a security-sensitive operation from several thousand entries in the program trace. The second observation addresses this challenge—to identify the code-patterns for a security-sensitive operation, it suffices to compare the program trace of a tangible side-effect associated with the operation against those that are not. For example, displaying a visible X client window, which involves mapping the window on the screen, is associated with Window_Map; closing and moving an X client window are not. Thus, to identify the code-patterns canonical to Window_Map, it suffices to compare the trace generated by opening an X client window against the trace generated by closing, or moving, a window. Similarly, closing a browser window is associated with closing all child windows, which involves Window_Enumerate, while typing to a window does not.

   With these two observations, identifying root-causes reduces to studying fewer than 10 entries, on average, in a program trace. Using this technique, I identified, amongst others, the root-causes of Window_Create as *Call* `CreateWindow`, of Window_Map as writes of `MapRequest` and `MapNotify` to the field `type` of a variable of type `xEvent` and that of Window_Enumerate as *Read* `WindowPtr->firstChild` and *Read* `WindowPtr->nextSib` and `WindowPtr` $\neq$ `0`, which are intuitively performed during linked-list traversal. Note that this technique can express code-patterns at the granularity of reads and writes to individual fields of data structures. I will discuss the tracing infrastructure, and algorithms to compare traces to identify root-causes in more detail in Section 2.1.1.

3. **Find all locations which are security-sensitive.** The third step uses the results of root-cause analysis to

statically identify all locations in the server where code-patterns that characterize a security-sensitive operation occurs; each of these locations performs the operation. Consider Figure 4, which shows a snippet of code from `MapSubWindows`, a function in the X server. It contains writes of `MapRequest` 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 and Window_Enumerate. Static analysis identifies the set of security-sensitive operations performed by each function call, as described in Section 2.1.3.

```
MapSubWindows(pParent, pClient) {
  pWin = pParent->firstChild;
  for (;pWin; pWin = pWin->nextSib)
  { event.u.u.type = MapRequest;...
    event.u.u.type = MapNotify;...
  }
}
```

Figure 4: `MapSubWindows`

In addition to identifying the locations where security-sensitive operations occur, in this step I also identify the subject and object associated with the operation. To do so, I identify the variables corresponding to subject and object data types (such as `Client` and `Window`) in scope. In most cases, this heuristic precisely identifies the subject and the object. In Figure 4, the subject is the client requesting the operation (`pClient`), and the object is the window whose children are to be mapped (`pParent`), both of which are formal parameters of `MapSubWindows`, and are thus in scope.

Steps 2 and 3 together identify all locations where the server performs security-sensitive operations, and at each location, also help identify the subject and object associated with the operation (recall that these are part of the security-event description). These steps are realized in AID. While several enhancements can be made to the basic techniques used by AID, I believe that I have advanced the state-of-the-art in understanding the fundamental issues involved in exposing application-specific security-events for authorization policy enforcement. Thus, these steps, and their realization in AID, satisfactorily addresses Task 1 from Section 1.3.

4. **Instrument the server.** Once AID has 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 AID determines that a statement **`Stmt`** is security-sensitive, and that it generates the security event $\langle sub, obj, op \rangle$, it is instrumented as shown below. Note that if **`Stmt`** is a call to a function `foo`, the query can alternately be placed in the function-body of `foo`.

   `if` (`query_refmon`($\langle sub, obj, op \rangle$) == False) `then` **`handle_failure`**; `else` **`Stmt`**;

   For example, because AID determines that `MapSubWindows` performs both the security-sensitive operations Window_Map and Window_Enumerate, it protects calls to `MapSubWindows` as follows:

   ```
   if (query_refmon(⟨pClient, pParent, Window_Map⟩) == False) then handle_failure;
   elsif (query_refmon(⟨pClient, pParent, Window_Enumerate⟩) == False)
   then handle_failure; else MapSubWindows(pParent,pClient)
   ```

   The statement **`handle_failure`** 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. As mentioned earlier, autho-

rization policies are expressed in terms of security-labels of subjects and objects. Security-labels can be stored in a table within the reference monitor (generated in step 5), or alternately, 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 both the subject and the object labels are passed to the reference monitor using `query_refmon`, the reference monitor can lookup the corresponding security-labels, and consult the policy.

5. **Generate reference monitor code.** This step generates code for the `query_refmon` function. I generate a template for this function, omitting two details that must be filled-in manually by a developer. First, the developer must specify how the policy is to be consulted, *i.e.*, he must implement f using an appropriate policy management API (such as the SELinux policy management server [84]). Second, he must implement the state update function, u, by specifying how the state of the reference monitor is to be updated. For example, when a security-event $\langle$`pClient`, `pWin`, Window_Create$\rangle$ 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 which copies data from $\texttt{pWin}_1$ to $\texttt{pWin}_2$ may entail updating the security-label of $\texttt{pWin}_2$. Because security-labels are either stored 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 Section 2.2. Note that while steps 2-4 are policy independent, step 5 requires implementation of f and u, which depend on the specific policy to be enforced. Steps 4 and 5 together ensure complete mediation of security-sensitive operations identified by AID, are realized in the tool ALPEN.

   I believe that ALPEN realizes the need to automate reference monitor construction. While there are still certain manual steps in ALPEN and I have not yet linked it up with off-the-shelf policy management APIs, I have a basic understanding of the research issues that will arise in doing so. Automating the tasks that are currently manual in ALPEN is also related to Task 4 from Section 1.3. I believe that ALPEN satisfactorily addresses Task 2 from Section 1.3.

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.

2.1 AID: A Tool to Locate Security-sensitive Operations

AID analyzes legacy servers and identifies locations where they perform security-sensitive operations. As discussed earlier, this is done in two phases: identifying code-patterns, the execution of which is the root-cause of security-sensitive operations, followed by a static analysis phase, which identifies all locations in the code where these code-patterns occur. In the following sections, I will discuss these steps in detail.

2.1.1 Root-cause Identification via Analysis of Program Traces

Recall that the ultimate goal is to retrofit a legacy server to ensure complete mediation of security-sensitive operations by policy lookups. A necessary step in this process is to identify the code-patterns executed by the server when it performs a security-sensitive operation. Protecting these code-patterns then achieves complete mediation.

Formally, a code-pattern is defined to be a function call, a read or a write to a field of a data-structure, or a comparison of two values, as shown in Figure 5. Note that code-patterns are expressed in terms of abstract-syntax-trees (ASTs); this allows expressing code-patterns more generically, in terms data-structures, rather than individual variables. *A root-cause of a security-sensitive operation is defined as the conjunction of one*

CodePat	:=	*Call* AST | *Read* AST
		| *Write* Value *to* AST
		| *Compare*(Value, Value)
Value	:=	constant | AST
AST	:=	(`type-name->`)*field

Figure 5: Code-pattern definition

or more code-patterns.

For example, in the X server, the root-cause of Window_Create is *Call* `CreateWindow`, while one root-cause of the operation Window_Enumerate, which enumerates all the children of a window is (`WindowPtr` $\neq$ `0` $\wedge$ *Read* `WindowPtr->firstChild` $\wedge$ *Read* `WindowPtr->nextSib`), which intuitively denotes the code-patterns used to traverse the list of children of a window. A security-sensitive operation can have several root-causes, corresponding to different ways of performing the operation. Both forward and backwards traversal of the linked list of children of a window constitute root-causes for Window_Enumerate, for instance.

The key challenge, however, is to discover root-causes of security-sensitive operations, as this is often not known *a priori*—this is especially the case with legacy and third-party code. Further, the root-cause must be *succinct*, *i.e.*, it must contain a small combination of code-patterns which, when executed, result in the security-sensitive operation. I addressed this challenge by making two observations.

Observation 1 (Tangible side-effects) *Security-sensitive operations are associated with tangible side-effects.*

Thus, if the server is induced to perform a tangible side-effect associated with a security-sensitive operation, then the server *must* perform the security-sensitive operation. Thus, identifying root-causes reduces to tracing the server as it performs the tangible side-effect, and recording the code-patterns from Figure 5 that it executes in the process. However, the program trace generated by the tangible side-effect may be huge. Using a tracing infrastructure, I found that 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. It is impossible to identify succinct root-causes by studying this trace. My second observation addresses this problem.

Observation 2 (Comparing traces generated by tangible side-effects) *Comparing a trace associated with a security-sensitive operation, against traces that are not associated with the operation, yields succinct root-causes.*

The key idea underlying this observation is that a tangible side-effect that does not perform a security-sensitive operation will not contain the code-patterns that characterize the operation. For example, the trace T_{open} that opens an X client window on the X server will contain the root-cause 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 root-cause 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 I was able to reduce, on average, the size of the trace by several orders of magnitude using this technique (Figure 6), whittling down the search for root-causes to fewer than 10 functions.

A technical difficulty must be addressed before I discuss how to compare traces of tangible side-effects. 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).

I manually identified the security-sensitive operations generated by each tangible side-effect. Because the side-effects I 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 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 the they are not labeled correctly. However I will show that, somewhat surprisingly, root-causes can be identified succinctly and precisely, *inspite of errors in labeling*. Because traces can have multiple labels, I formulate *set-equations* for each label (recall that a label is just a security-sensitive operation) in terms of label-sets of all the traces.

Definition 3 (Set equation) *Given set S, a set $B \subseteq S$, and a collection $C=\{C_1, C_2, \ldots, C_n\}$ 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 $\cup$ or $\cap$.*

Algorithm: FIND_ROOT-CAUSE($\mathcal{X}$, S, Seff)
Input : (i) $\mathcal{X}$: Server to be retrofitted, (ii) S: A set of security-sensitive operations {op_1, ..., op_n}, and (iii) Seff: A set of tangible side-effects {$seff_1$, ..., $seff_m$}.
Output : RC_1, ..., RC_n : Each RC_i is the root-cause of the security-sensitive operation op_i.

1 $\mathcal{X}'$:= $\mathcal{X}$ instrumented to perform tracing;
2 **foreach** (tangible side-effect $seff_i \in$ Seff) **do**
3 T_i := Trace generated by $\mathcal{X}'$ when induced to perform $seff_i$;
4 *label*(T_i) := Set of operations (from S) involved in $seff_i$;
5 **foreach** ($op_i \in S$) **do**
6 SE_i := Set-equation for op_i in terms of *label*(T_1), ..., *label*(T_m);
7 $CPset_i$:= Set of code-patterns in T_i;
8 RC_i := Result when operations in SE_i are performed on $CPset_1$, ..., $CPset_m$;

Algorithm 1: **Algorithm to find root-causes of security-sensitive operations.**

To find a succinct root-cause for an operation op, I 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 generated 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 root-cause for op.

For example, if T_1 is a trace of side-effect $seff_1$, which performs op and op′, and T_2 is a trace of side-effect $seff_2$, which 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 root-cause of op, I let $B = \{op\}$, and observe that $B = C_1 - C_2$. I 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 root-cause of op. This process is formalized in Algorithm 1.

Finding 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 '$\cap$' and '$\cup$' as the conjunction and disjunction operators, respectively. Each disjunct in the equation is a *clause*. It can be shown that the *CNF-set-equation problem*, which is a restricted version of the general problem of finding set-equations, is NP-complete.

Trace name	A	B	C	D	E	F	G	H	I
Side-effect ⟶ Operation ↓	open xterm	close xterm	open browser	close browser	type to window	move window	open & close twm menu	switch windows	open menu (browser)
Create	✓		✓				✓		✓
Destroy		✓	★	✓			✓	★	
Map	✓		✓				✓		✓
Unmap		✓	★	✓			✓	★	
Chstack	✓		✓				✓	✓	✓
Getattr	✓		✓			●	●		✓
Setattr	✓		✓			✓	●	★	✓
Move			★			✓		★	★
Enumerate	★	★	✓	✓		✓	★	✓	✓
InputEvent					✓	✓	✓	✓	✓
DrawEvent	✓	✓	✓	✓		✓	✓	✓	✓
Distinct Functions	669	140	1117	233	132	95	382	516	145

Figure 6: Labeled traces of tangible side-effects obtained from the X server.

Definition 4 (CNF-set-equation problem) *Given a set S, a set $B \subseteq S$, a collection C of subsets of S (as in Definition 3), and an integer k, does B have a CNF-set-equation with at most k clauses?*

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

2.1.2 Evaluation of Root-Cause-Finding Algorithm

I have implemented Algorithm 1 in AID. I used a modified version of `gcc` (created by Jaeger *et al.* [47]) to compile the server. During compilation, instrumentation is inserted statically at statements which read and write to fields of critical data structures. I used this to log the field and the data structure that was read from, or written to, and the function name, file name, and the line number at which this occurs. I then induced the modified server to perform a set of tangible side-effects, and proceed as in Algorithm 1 to find root-causes.

I applied this to find root-causes of security-sensitive operations in the X server. In particular, I recorded reads and writes to fields of data structures such as `Client`, `Window`, `Font`, `Drawable`, `Input`, and `xEvent`. Figure 6 shows a portion of the result of performing lines (1)-(4) of Algorithm 1. Columns represent 9 tangible side-effects, and rows represent 11 security-sensitive operations on the `Window` data structure. I manually labeled each tangible side-effect with the security-sensitive operations it performs. These entries are marked in Figure 6 using ✓ and ●. 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

Operation	Set Equation	|RC|	Root-cause
Create	∩(A, C, G) - D - H	12	*Call* `CreateWindow`
Destroy	∩(B, D) - A	7	*Call* `DeleteWindow`
Map	∩(A, C, G) - D - H	12	*Write* `MapRequest` *to* `xEvent->union->type` ∧ *Write* `MapNotify` *to* `xEvent->union->type`
Unmap	∩(B, D) - A	7	*Write* `UnmapNotify` *to* `xEvent->union->type`
Chstack	∩(A, C, G, H, I) - D	4	*Call* `MoveWindowInStack`, *Call* `ReflectStackChange`, *Call* `WhereDoIGoInStack`
Getattr	∩(A, C, F, I)	5	*Call* `GetWindowAttributes`
Setattr	∩(A, C, F, I)	5	*Call* `ChangeWindowAttributes`
Move	F - G - A	11	*Call* `ProcTranslateCoords`
Enumerate	∩(C, D, F, H, I)	5	*Read* `WindowPtr->firstChild` ∧ *Read* `WindowPtr->nextSib` ∧ `WindowPtr≠0`, *Read* `WindowPtr->lastChild` ∧ *Read* `WindowPtr->prevSib`
InputEvent	E - C	12	*Call* `CoreProcessPointerEvent`, *Call* `CoreProcessKeyboardEvent`, . . .
DrawEvent	∩(A, B, C, D) - E	13	*Call* `DeliverEventsToWindow`

Figure 7: Root-causes obtained using labeled traces from Figure 6.

contains 669 calls to distinct functions in the X server, as shown in the last row of Figure 6.

Figure 7 shows 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 root-causes, the size of the resulting set, and the set of root-causes is shown. Note that each security-sensitive operation can have more than one root-cause, as for example, is the case with Window_Enumerate and Window_Chstack.

To find errors in manual labeling of traces, I did the following. After finding root-causes of security-sensitive operations, we checked each trace for the presence of these root-causes. Presence of a root-cause of a security-sensitive operation in a trace which is not labeled with that security-sensitive operation shows an error in manual labeling; such entries are marked ★ in Figure 6. For example, I did not label the trace generated by opening an `xterm` with Window_Enumerate. On the other hand, absence of root-causes of a security-sensitive operation in a trace which is labeled with the security-sensitive operation also shows an error in manual labeling; such entries are marked ● in Figure 6. Thus for example, I 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.

I now discuss the evaluation of AID's root-cause finding algorithm by answering the following questions:

1. **How effective is AID at locating root-causes?** Raw-traces generated by tangible-side effects, have on average, 103967 code-patterns. However, AID first abstracts each trace to function calls: it first identifies root-causes at the function-call 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 Figure 6. The third column of Figure 7 shows, in terms of the number of function calls, the size of RC, which is the result obtained by computing the set-equation for each security-sensitive operation, to determine root-causes. Note that AID was able to achieve over two orders of magnitude reduction in terms of the number of distinct functions to be examined.

   I examined each of the functions in RC to determine if it is indeed a root-cause. In most cases, I found that for a security-sensitive operation, a single function in RC performs the operation. However, in some cases, multiple functions in RC seemed to perform the security-sensitive operation. For example, both

Call `MapWindow` and *Call* `MapSubWindow`, which were present in RC, performed Window_Map. In such cases, I (manually) examined the traces generated by AID 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* `MapRequest` *to* `xEvent->union->type` $\wedge$ *Write* `MapNotify` *to* `xEvent->union->type`). For security-sensitive operations such as Window_Chstack, where the traces generated by AID did not contain commonalities in the code-patterns exercised by different functions, I deemed each of the function calls in RC to be root-causes of the operation.

2. **How precise are the root-causes found?** For each of the root-causes recovered by AID for the X server, I manually verified that it is indeed a root-cause of the security-sensitive operation in question.

   However, in general, AID need not recover all the root-causes. Because AID is a runtime analysis, it can only capture the root-causes 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 root-causes collected by AID against these traces, confidence can be increased in the precision of root-causes obtained by AID. In the future, I plan to investigate static techniques to identify root-causes to overcome this limitation. To do so, I intend to explore the use of symbolic evaluation to replace actual execution.

3. **How much effort is involved in manual labeling of traces?** In all, I collected 15 traces for different tangible side-effects exercising different `Window`-related security-sensitive operations. It took me a few hours to manually label these traces with security-sensitive operations (contrast this with the 2 year effort by NSA to retrofit the X server).

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, I may miss security-sensitive operations that may be performed (★ entries in Figure 6), or label a trace with security-sensitive operations that are not actually performed by the trace (● entries). My 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 I was able to recover root-causes precisely *inspite of labeling errors*. If a security-sensitive operation is wrongly omitted from the labels of a tangible side-effect (the ★ case), then because the same security-sensitive operation often appears in the labels of other tangible side-effects, a set-equation can still be formulated for the operation, and the root-cause can be recovered. On the other hand, if a security-sensitive operation is wrongly added to the labels of a tangible side-effect (the ● case), none of the functions in RC will perform the tangible side-effect. In this case, trace labels are refined, and the process is iterated until a root-cause is identified.

2.1.3 Identification of Security-sensitive Locations using Static Analysis

Having identified root-causes of security-sensitive operations, AID employs static analysis to find all locations in the code of the server where these root-causes occur.

AID currently identifies security-sensitive locations at the granularity of function calls. Note that several, but not all, root-causes are function calls. AID considers root-causes that are not function calls, such as those of Window_Map, Window_Unmap, and Window_Enumerate, and identifies functions which contain these code-patterns. The idea is that by mediating calls to functions which contain these patterns, the corresponding security-sensitive operations are mediated as well. This is done using a flow-insensitive, intraprocedural analysis, as described in Algorithm 2. AID first identifies the set of code-patterns that appear in the body of a function, and then checks to see if the root-causes of a security-sensitive operation appear in this set. If

so, the function is marked as performing the security-sensitive operation. For a security-sensitive operation whose root-causes contain only function calls, Algorithm 2 marks each of these functions as performing the operation.

Algorithm: FIND_SECURITY-SENSITIVE_LOCATIONS($\mathcal{X}$, S, $\mathcal{RC}$)
Input : (i) $\mathcal{X}$: Server to be retrofitted, (ii) S: Set of security-sensitive operations {op_1, ..., op_n}, and (ii) $\mathcal{RC}$: Set of root-cause sets rc_1, ..., rc_n of op_1, ..., 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 $\mathcal{X}$.

```
1  /* Process root-causes with only function calls */;
2  foreach (root-cause set rc_i in RC) do
3      rcset_i := Set of code-patterns in rc_i;
4      if (rcset_i == {Call f_1, ..., Call f_m}) then
5          foreach (f ∈ {f_1, ..., f_m}) do
6              Opset(f) = Opset(f) ∪ {op_i};
7          RC = RC - {rc_i};
8  /* Process other root-causes */;
9  foreach (function f in X) do
10     Opset(f) := φ;
11     CP(f) := Set of code-patterns in f (as determined using the ASTs of statements in f);
12     foreach (root-cause set rc_i in RC) do
13         if (rcset_i ⊆ CP(f)) then Opset(f) := Opset(f) ∪ {op_i};
14 return Opset;
```

Algorithm 2: **Finding functions which contain code-patterns that appear in root-causes.**

Consider the function `MapSubWindows` in the X server (see Figure 4). This function maps all children of a given window (`pParent` in Figure 4) to the screen. Note that it contains code-patterns which constitute the root-cause of both Window_Enumerate and Window_Map. Thus, Opset(`MapSubWindows`) = {Window_Map, Window_Enumerate}.

AID uses a slightly more powerful variant of code-pattern language in Figure 5 to match code-patterns in function bodies. In particular, it extends Figure 5 with the ability to specify relations between different instances of ASTs. Thus, for example, it can match patterns such as *Read* $\texttt{WindowPtr}_1\texttt{->firstChild} \wedge$ *Read* $\texttt{WindowPtr}_2\texttt{->nextSib}$ *Where* $\texttt{WindowPtr}_1 \neq \texttt{WindowPtr}_2$. Note that for traversing the linked list of child windows, as in Figure 4, the *parent*'s `firstChild` field is read, followed by `nextSib` of *child* windows. In the absence of this feature, AID will also match functions in which the `firstChild` and `nextSib` fields of the same window are read, which does not correspond to linked-list traversal, and will thus result in extra functions reported as performing Window_Enumerate.

Finally, AID also helps identify the subject requesting, and the object upon which, the security-sensitive operation is to be performed. To do so, it identifies variables of the relevant types that are in scope. 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.

```
 bool query_refmon(Client *sub, Window *obj, Operation OP) {
switch (OP) {
   case WINDOW_CREATE:
     rc = policy_lookup(sub->label, NULL, WINDOW_CREATE);
     if (rc == success) {
       obj->label = sub->label;
       return True;
     } else { return False; }
   case WINDOW_MAP:
   ...
} }
```

Figure 8: Code fragment showing the implementation of `query_refmon` for Window_Create.

2.1.4 Evaluation of Security-sensitive Location-Finding Algorithm

I have implemented AID's static analysis algorithm as a plugin to the CIL [69] toolkit. I present an evaluation of AID's security-sensitive location finding algorithm by answering two questions:

1. **How precise are the security-sensitive locations found?** Algorithm 2 precisely identifies the set of security-sensitive operations performed by each function, with one exception. AID 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 17 functions reported as performing Window_Enumerate, only 11 actually do. I found that this was because of the inadequate expressive power of the code-pattern language. In particular, AID matches functions which contain the code-patterns `WindowPtr ≠ 0`, *Read* `WindowPtr->firstChild`, and *Read* `WindowPtr->nextSib`, but do not perform linked-list traversal. These false positives can be eliminated by enhancing the code-pattern language with more constructs (in particular, loop constructs).
2. **How easy is it to identify subjects and objects?** As mentioned earlier, AID also helps identify subjects and objects by identifying variables of relevant data types in scope. This simple heuristic is quite effective: out of 24 functions, calls to which were identified as security-sensitive for `Window` operations, the subject and object were the unique variables of the relevant types (`Client` and `Window`, respectively) in scope for 20 of them. In the remaining functions, local variables of type `Window` were declared for manipulating the object within the function. However, even in this case, manual inspection quickly revealed the object `Window` easily.

2.2 ALPEN: A Tool to Protect Security-sensitive Operations

Locations identified as performing security-sensitive operations by AID are protected by ALPEN using instrumentation. Because AID helps recover the complete description of security-events, adding instrumentation is straightforward, and calls to `query_refmon` are inserted as described earlier. If the function to be protected is implemented in the server itself (as opposed to a library call), as is the case with all the security-sensitive function calls in the X server, calls to `query_refmon` can be placed within the function body itself. Because the same variables that constitute the security-event are also passed to `query_refmon` (*i.e.*, if $\langle sub, obj, op \rangle$ is the security event, then the corresponding call is `query_refmon`$(\langle sub, obj, op \rangle)$), and the data structures used to represent subjects and objects are internal to the server, ALPEN avoids TOCTTOU bugs [11] by construction.

ALPEN also generates a template implementation of `query_refmon`, as shown in Figure 8. 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 `query_refmon`, generated by ALPEN. We impose no restrictions on the policy language, or the policy management framework. Figure 8 shows an example: it shows a snippet of code that is automatically generated by ALPEN. Subject and object labels are stored as fields (`label`) in the data structures representing them. The italicized statement, a function-call `policy_lookup`, 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 server for user-space applications [84], which manages policies written in the SELinux policy language. If this tool is used, the relevant API call to replace `policy_lookup` 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 u. Note that u depends on the policy to be enforced; for example, the Chinese Wall policy [13] requires labels to be updated if a client is authorized for an operation. Because the state update function u is dependent on the policy, policy-management tools such as Polserver also provide functionality to determine how security-labels must change based upon whether the authorization request succeeds or fails, thus relieving the developer of this task.

   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 bold in Figure 8 shows a simple example of 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 I place the call to `query_refmon` 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, free the memory that was allocated, and return a NULL window (*i.e.*, **`handle_failure`** is implemented as `return NULL;`).

Finally, it remains to explain how I bootstrap security-labels in the server. I assume that the server runs on a machine with a security-enhanced operating system. I use operating system support to bootstrap security-labels based upon how clients connect to the server. For example, in an SELinux system, all socket connections have associated security-labels, and X clients connect to the X server using a socket. Thus, I use the security-label of the socket (obtained from the operating system) as the security-label of the X client. I then propagate X client security-labels as they manipulate resources on the X server, as shown in Figure 8, where the client's security-label is used as the security-label for the newly-created window. While this trick works with the X server, a principled approach to obtain and propagate labels is still needed. I intend to address this as part of my work on authorization policy formulation (Section 3).

Using AID and ALPEN, I have retrofitted the X server to enforce authorization policies. I have demonstrated authorization policies that prevent previously published attacks [34]. Two previously-known attacks, and the policies that prevent them are given below:

1. **Attack.** An X client can change properties of windows belonging to other X clients, for *e.g.*, by changing their background or content. This is used as the basis for other attacks [53].

   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 will 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 will 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 opening a new window involves exercising the Window_Chprop security-sensitive operation.

2. **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 [9]). 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 'cut' from a high-security X client window can only be 'pasted' into X client windows with equal or higher security". Note that the existing security mechanism in the X server (the X security extension [87]) cannot enforce this policy if there are more than two security-levels.

2.3 A Unified Approach to Identify Security-Sensitive Operations and their Root-Causes

In both case studies that I have worked on so far, namely retrofitting the Linux kernel, and retrofitting the X server, I have relied on manually-determined security-sensitive operations, in both cases, a list identified by the NSA. These were typically accompanied by an English-language description of their intended meaning. My work on AID formalizes security-sensitive operations in terms of their root-causes, which are conjunctions of code-patterns (where code-patterns are as defined in Figure 5).

The next logical step is to design heuristics to automate the identification of security-sensitive operations. Recall that security-sensitive operations are defined as operations which manipulate data structures of the server $\mathcal{X}$. For example, the security-sensitive operation Dir_Rmdir in Linux, which performs directory removal results in `inode->i_size` being set to 0, and `inode->i_nlink` being decremented, corresponding, intuitively, to setting the `i_size` field of the inode of the removed directory to 0, and decrementing the number of links (`i_nlink` field) of both the directory that is removed, and its parent. Several such examples are presented elsewhere [33, 46]. Thus, one possible technique to identify security-sensitive operations is to mark each *Read*/*Write*/*Call* to each (`type-name->`*)`field`, where `type-name` denotes the type name of a data structure in the server $\mathcal{X}$ as security-sensitive.

Recall, however, that an authorization policy is a set of triples $\{\langle sub, obj, op\rangle\}$, where op denotes a security-sensitive operation. Creating and maintaining authorization policies in which security-sensitive operations are represented at the granularity of *Read*s, *Write*s and *Call*s of individual fields of data structures of the server $\mathcal{X}$ has two disadvantages: (i) it is tedious because it increases the number of security-sensitive operations for which authorization policies must be written, and (ii) in retrofitting the server $\mathcal{X}$ to monitor these security-sensitive operations, a larger number of `query_refmon` calls will have to be placed, potentially increasing the runtime overhead of authorization. It is apparently for these reasons that authorization policies are not written at this level of granularity for either the Linux kernel or the X server. For example, authorization policies are not written at the granularity of `Read`s and `Write`s to individual fields (such as `i_size` and `i_nlink`) of the `inode` data structure. Instead, a security-sensitive operation is typically a *combination* of *Read*s, *Write*s and *Call*s to fields of data structures, and the combination of these manipulations achieves some *high-level goal*. For example,

1. In Linux, Dir_Rmdir is characterized by *Decrement* `inode->i_nlink` and *Write* 0 *to* `inode->i_size`. Together, manipulations of these fields of `inode` achieve directory removal, because they reset the size of the directory to be removed, and also unlink the directory from its parent directory.
2. In Linux, Dir_Write is characterized by *Write* `inode->i_ctime` and *Call* `address_space_ops->prepare_write()`. The former sets the modification time of the directory, and the latter is a low-level operation to write to the directory.
3. In the X server, mapping a window to the screen is characterized by a request by the client to map to

the screen (*Write* `MapRequest` *to* `xEvent->union->type`), followed by a notification by the X server to the X client that the request was successful (*Write* `MapNotify` *to* `xEvent->union->type`), as shown in Figure 7.

Note that in each of these cases, a particular combination of *Read*s, *Write*s and *Call*s achieves a high-level goal, such as directory removal, writing to a directory, or mapping a client window to the screen. In summary, I make the following observation:

Observation 3 (Security-sensitive Operations) *Security-sensitive operations are typically a combination of Reads, Writes, and Calls of fields of data-structures of the server $\mathcal{X}$, where the combination achieves some high-level goal.*

AID uses the fact that these high-level goals typically are associated with tangible side-effects. Thus, given the high-level goal to be achieved (and the associated tangible side-effect), AID helps to recover root-causes of security-sensitive operations as a combination of code-patterns which achieve the high-level goal.

I would like to extend AID *to identify security-sensitive operations in terms of their root-causes even when high-level goals and their associated tangible side-effects are not known a priori.* The reason is because for a user-space server, high-level goals may not be defined precisely, or may not be documented properly. This was the case with the Linux kernel as well as with the X server, and a design team manually identified them. However, to reduce manual effort in retrofitting legacy code, an automated technique to identify security-sensitive operations (and their root-causes) is desirable. My approach will be based upon the following hypothesis:

Hypothesis 1 (Frequency and Proximity of associated data-structure manipulations) Because each security sensitive operation is a combination of Reads, Writes, and Calls of fields of data-structures of the server $\mathcal{X}$, the Reads, Writes and Calls that are associated with a particular security-sensitive operation must appear in close proximity in any runtime trace of the server $\mathcal{X}$. Further, if the security-sensitive operation is exercised several times during the trace, these occurrences will be frequent.

This suggests an approach based upon data-mining to find the *Reads*, *Writes* and *Calls* which typically appear together, and are thus most likely associated with a security-sensitive operation. For example, consider a trace generated by the X server: if a window is mapped several times onto the screen, this trace will likely contain several occurrences of both *Write* `MapRequest` *to* `xEvent->union->type` and *Write* `MapNotify` *to* `xEvent->union->type` in close proximity to each other. I intend to formalize the problem of identifying security-sensitive operations as *mining frequent itemsets*. Using this approach, security-sensitive operations will be identified in terms of their root-causes. For example, for the X server, (*Write* `MapRequest` *to* `xEvent->union->type`) $\wedge$ (*Write* `MapNotify` *to* `xEvent->union->type`) will be reported as a security-sensitive operation (representing Window_Map). I intend to obtain traces by running the server on regression test inputs. To explain how I intend to use frequent itemset mining, I first present a brief background on the problem, borrowed from [57].

An *itemset* is a set of *items*. A *sub-itemset* is a subset of an itemset. Suppose we are presented with a database consisting of several itemsets. A sub-itemset S is said to appear *frequently* if it appears as a subset of more than *min_support* itemsets, and the number of occurrences is called its *support*. The itemsets that contain S are called its *supporting itemsets*. For example, given a database of itemsets:

$$\{\{a,b,c,d,e\},\{a,b,d,e,f\},\{a,b,d,g\},\{a,c,h,i\}\}$$

The support of sub-itemset $\{a, b, d\}$ is 3. One variant of the problem of frequent itemset mining, the one I am most interested in, is that of mining *closed sub-itemsets*. The problem here is to identify frequently-occurring closed sub-itemsets, where a closed sub-itemset is one whose support is different from that of its super-itemsets. I am interested only in closed sub-itemsets because, intuitively, I want to find the itemsets of maximal size, which are also most frequent. That is, if both $\{a\}$ and $\{a, b\}$ have the same support, then I am only interested in $\{a, b\}$. *FPClose* [39] is an efficient algorithm for the problem of mining closed sub-itemsets. Note however, that because an itemset is an unordered set of items, we lose information about the sequencing of items in the database. Closed subsequence mining is a variant of the closed sub-itemset mining problem that respects the sequencing of data; *CloSpan* [90] is an efficient algorithm designed for closed subsequence mining.

The intuition behind using closed sub-itemset mining for identifying security-sensitive operations is as follows. Suppose we have a trace generated by the server $\mathcal{X}$. Suppose that we have divided the trace into a set of itemsets, thus representing a database of itemsets, where each itemset is a set of code-patterns from Figure 5. Frequent closed sub-itemsets then correspond to frequently appearing code-patterns, and all the code-patterns in a closed sub-itemset are most likely associated with a single security-sensitive operation. For example, because *Write* `MapRequest` *to* `xEvent->union->type` and *Write* `MapNotify` *to* `xEvent->union->type` always happen when a window is mapped to the screen, a trace generated by the X server will likely contain these two code-patterns frequently. Further, the trace will contain these two code-patterns in close proximity. Thus, by modifying *FPClose* and *CloSpan* to mine frequent itemsets which contain items that are in close proximity, we will identify code-patterns which are most likely associated with a security-sensitive operation. This intuition has recently been used to mine programming idioms [57] and duplicated code [56]—these papers have also modified the basic *FPClose* and *CloSpan* algorithms to include the notion of proximity. This relationship is not surprising, because security-sensitive operations are also performed in certain idiomatic ways, and the problem of finding root-causes of security-sensitive operations can be mapped to the problem of finding programming idioms.

It remains to explain how a trace generated by the server $\mathcal{X}$ is converted into a database of itemsets. To do so, several heuristics can be employed, three of which I list below:

1. Split the trace into equal-length partitions, treating each partition as an itemset.
2. Instead of equal-length partitions, have random-length partitions.
3. Partition the trace based upon functions in the server $\mathcal{X}$ that generated the portion of the trace. That is, events in the trace generated by a function f in $\mathcal{X}$ are treated as an itemset.

Algorithm: FIND_SECURITY-SENSITIVE_OPERATIONS($\mathcal{X}$, I)
Input : (i) $\mathcal{X}$: Server to be retrofitted, (ii) I: Set of inputs for the $\mathcal{X}$ (for *e.g.*, regression-testing inputs)
Output : A set of code-pattern-sets, where each code-pattern-set denotes a security-sensitive operation in terms of its root-cause.
1 Trace $\mathcal{X}$ as it runs on each input from I, and concatenate traces;
2 Split trace into database of itemsets (using heuristics);
3 Mine frequent sub-itemsets in database, and report frequent sub-itemsets as root-causes;

Algorithm 3: **Algorithm to find security-sensitive operations in terms of their root-causes.**

The algorithm for identifying security-sensitive operations in terms of their root-causes is presented in Algorithm 3. Note that this technique has three advantages over AID:

1. It does not require the set of security-sensitive operations to be known in advance.
2. It identifies security-sensitive operations in terms of their root-causes, *i.e.*, it reports combinations of code-patterns as security-sensitive operations, which is by definition the root-cause of the security-sensitive operation.
3. It is completely automated. In AID, manual intervention was necessary to run the server $\mathcal{X}$ and force it to perform a tangible side-effect. On the other hand, Algorithm 3 uses a set of regression-test inputs I, and automatically identifies frequent itemsets, and is thus is completely automatic.

While I believe that the data-mining-based approach will help identify security-sensitive operations, I see a few challenges that I may encounter, which I hope to address as I begin work on this problem:

1. **Frequency of security-sensitive operations.** Algorithm 3 makes a key assumption: that security-sensitive operations are exercised often by the regression testing inputs I. Because this algorithm mines *frequent* itemsets, a security-sensitive operation that is not exercised, or is exercised infrequently will not be found by the algorithm, thus resulting in *false negatives*. A possible option here to overcome this limitation is to augment Algorithm 3 with code-coverage metrics, which report the frequency with which different portions of the code in $\mathcal{X}$ were exercised.
2. **Number of spurious itemsets.** The approach may report a large number of spurious itemsets (for *e.g.*, singleton itemsets), which may not achieve any high-level goals, and are thus not security-sensitive. The challenge here will be to somehow rank the results obtained, so that itemsets which are most likely to be root-causes of security-sensitive operations are reported first.
3. **False negatives and Accuracy.** I am still not clear how I will evaluate the completeness of this technique. Completeness of the above algorithm is defined as the ability to identify all security-sensitive operations. While evaluating completeness for the X server and the Linux kernel is relatively easy—by comparing the output of the algorithm against manually-identified security-sensitive operations—there is a need for metrics to evaluate the completeness of the algorithm.

3 Authorization Policy Formulation

In this section, I describe my ideas on authorization policy formulation. An appropriate authorization policy must accompany security-aware code created using the techniques described in Section 2. Enforcement mechanisms inserted in security-aware code can only prevent security holes if they are used in conjunction with a *secure* authorization policy, *i.e.*, one that satisfies end-to-end system security goals, such as Biba or Bell-LaPadula.

Intuitively, an authorization policy $\mathcal{A}$ is an entity, which if enforced properly, constraints the execution of a program $\mathcal{X}$. Let $\mathcal{S}$ represent the state-space of a program $\mathcal{X}$, denoting all possible executions of $\mathcal{X}$ (the state space description also includes the name of the subject executing $\mathcal{X}$). A policy $\mathcal{A}$ splits the state space into two disjoint sets $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\bar{\mathcal{A}}}$, denoting, respectively, portions of the state space allowed and disallowed by $\mathcal{A}$. As described earlier, a policy is expressed as a set of triples $\langle sub, obj, op \rangle$. A *security goal* is the specification of a security property to be achieved by the policy. Thus, a policy that satisfies a security goal $\mathcal{G}$ will be such that $\mathcal{S}_{\mathcal{A}} \models \mathcal{G}$ (in model checking terminology). Policy analysis tools such as SLAT [40], in fact, express security goal $\mathcal{G}$ using a formal specification language, such as LTL, and use model checking to check that $\mathcal{S}_{\mathcal{A}} \models \mathcal{G}$.

In policy formulation, the objective is to solve the dual problem, *i.e.*, given $\mathcal{G}$ and $\mathcal{X}$, generate a policy $\mathcal{A}$, such that $\mathcal{S}_{\mathcal{A}} \models \mathcal{G}$ by construction. The analogy between policy analysis and policy formulation is somewhat akin to the analogy between program verification and program synthesis. In program synthesis, the objective

is to generate a program, given a formal specification, such that the program models the specification by construction. Unfortunately, policy formulation is hard to automate for several reasons:

1. **Security Goals are Partial Specifications.** Often the security-goal $\mathcal{G}$ is only a *partial specification*, and not a complete description of the policy. It only specifies certain properties that a policy must satisfy. For example, consider a UNIX system with two users *Alice* and *Bob* with security labels *Top-Secret* and *Unclassified*. Further, suppose that `nuclear-secret` and `news` are two files labeled *Top-Secret* and *Unclassified*, respectively. Consider the policy shown in Figure 9.

   1 ⟨*Alice*, `nuclear-secret`, File_Read⟩
   2 ⟨*Alice*, `news`, File_Read⟩
   3 ⟨*Bob*, `news`, File_Read⟩

   Figure 9: A simple authorization policy

   This policy clearly satisfies Bell-LaPadula confidentiality goals, because it does not allow an *Unclassified* user to read *Top-Secret* files. However, so does the policy without line **2**. In fact, any subset of the above policy satisfies Bell-LaPadula confidentiality. Thus, given a security goal, it is not possible to uniquely formulate an authorization policy.

   A natural question is whether the solution to the above problem is to generate a "maximal" policy that satisfies system security goals. While this is certainly a solution, it must be noted that in real-world policy management, statements are added and deleted from the policy. Policy analysis is used to ensure that the modified policy still satisfies certain partial specifications. For example, the need may be felt to prevent user *Alice* from reading *Unclassified* files, and line **2** may be deleted from the policy in Figure 9 without violating Bell-LaPadula confidentiality. Note that preventing *Alice* from reading *Unclassified* files is an additional security goal. Unfortunately, changes to authorization policies to reflect changing security requirements are often not documented formally in terms of changes to security goals.

   As a result, while it is reasonable to assume that security-goals describing desirable properties (*i.e.*, partial specifications) of a security policy will be available, it is unrealistic to assume that a complete specification of end-to-end system security goals will be available.

2. **Application Functionality Requirements must be met.** Authorization policies must also satisfy *functional* requirements. That is, in addition to meeting end-to-end system security goals, they must allow an application to perform useful functionality. For example, Apache typically requires File_Read access to configuration files and File_Write access to log files for proper functioning. An empty authorization policy satisfies all system security goals (because $\mathcal{S}_{\mathcal{A}} = \phi$), but also does not allow the application to perform useful functionality. To the best of my knowledge, there are no formal languages to express application functionality requirements.

Because of the above problems, I intend to follow a different approach to formulate authorization policies. My approach is based upon the following hypothesis.

Hypothesis 2 (Operating System-level Policies are Well-Tested) Authorization policies written for, and enforced by, the operating system on user-space applications are typically well-tested, and meet end-to-end system security goals as well as useful application functionality.

To see the validity of this hypothesis, consider the SELinux example policy, which is available with standard distributions of SELinux. This example policy contains over 50, 000 policy statements [40, 48] for different kinds of user-space applications, including several servers, such as web-servers and the X server,

and allows these applications to function usefully. The SELinux example policy does not embody any particular security goals, but is instead supposed to serve as an example policy, from which system administrators can create their own site-specific policies (based upon site-specific security goals). The example policy is modified to meet site-specific system security goals—policy analysis tools such as Gokyo [48], SLAT [40], and tools from the SELinux policy development toolkit [85] can be used to determine whether the modified policy indeed meets these goals.

While it is realistic to expect example policies to be developed for the operating system, it is unrealistic to expect the same for arbitrary user-space applications, because of the sheer number of such applications. Thus, there is a need for techniques to formulate authorization policies for user-space applications, such as servers. My approach to formulate authorization policies for user-space servers is based upon *policy translation*. That is, I intend to *convert* an authorization policy written for, and enforceable by the operating system, into an authorization policy that is enforceable by a security-aware user-space server.

To understand the intuition behind the policy translation approach, consider the original motivation of the need for security-aware code. Our objective is to protect unauthorized access to *operating system-level resources*, such as files, shared memory and inter-process communication. Unfortunately, user-space servers also contain resources, such as buffers and caches, which are shared between clients of the server. If the server does not enforce authorization policies on the way clients use these shared resources, it can result in unauthorized access to operating system-level resources as well. Thus, *the root of the problem is that user-space servers may store information related to operating system resources in the shared resources that they offer to clients*. Because operating system-level policies meet security goals (by hypothesis 2), it suffices to translate these policies into ones enforceable by a security-aware user-space server in order to protect unauthorized access to operating system-level resources.

An ideal policy translator that converts an operating system-level policy into a user-space application-level policy should satisfy two properties:

1. **Soundness,** which says that if a subject *sub* is not permitted to perform operation *op* on object *obj* in the operating system-level policy, he should not be permitted to do so in the application-level policy as well. Soundness is necessary to maintain the security goals as an invariant.
2. **Completeness,** which says that if a subject *sub* is permitted to perform operation *op* on object *obj* in the operating system-level policy, he should be permitted to do so in the application-level policy as well. Completeness is necessary to ensure that the application-level policy does not constrain functionality of the application.

Thus, a sound and complete policy translator would convert the operating system-level authorization policy from Figure 9 into one enforceable by the X server as shown in Figure 10

1 ⟨*Alice*, `nuclear-secret`, File_Read⟩		**1** ⟨*Alice_Window*, *nuclear-secret-buffer*, Window_Paste⟩
2 ⟨*Alice*, `news`, File_Read⟩	→	**2** ⟨*Alice_Window*, *news-buffer*, Window_Paste⟩
3 ⟨*Bob*, `news`, File_Read⟩		**3** ⟨*Bob_Window*, *news-buffer*, Window_Paste⟩

Figure 10: Translating the authorization policy from Figure 9 for a security-aware X server.

Here, *Alice_Window* and *Bob_Window* denote the security-labels of X client windows belonging to *Alice* and *Bob*, respectively, and *nuclear-secret-buffer* and *news-buffer* denote security-labels of the cut-buffer of the X server, when it stores the `nuclear-secret` and `news` files, respectively. Note that the converted authorization policy also satisfies Bell-LaPadula confidentiality.

I intend to design and implement such a policy translator. Recall that an authorization policy is a set of

triples $\langle sub, obj, op \rangle$. Thus, translating a policy written for the operating system into one for a security-aware server entails solving the following sub-problems:

1. **The security-sensitive operation mapping problem** is the problem of determining how the security-sensitive operations of the security-aware server map to security-sensitive operations of the operating system. For example, in case of the X server, this is the problem of determining that a `Window_Paste` requested by a subject *sub* can result in a `File_Read` operation of the file presently contained in the cut-buffer.

   I intend to address this problem by understanding the kernel footprint of each security-sensitive operation of the security-aware server. To do so, I intend to use dynamic program analysis: I will trace the kernel as the security-aware server performs a security-sensitive operation, and relate the security-sensitive operation to its kernel footprint.

   The problem of mapping security-sensitive operations in one vocabulary to their equivalents in another vocabulary also arises in intrusion detection. Garfinkel and Rosenblum [35] discuss this problem in the context of constructing an intrusion-detection system within a virtual machine monitor. In this case, the set of events visible to the virtual machine monitor are at a much lower level (*e.g.*, at the granularity of individual machine instructions) than are typically specified by a security analyst (*e.g.*, at the system-call level). Thus, they too felt the need for creating a mapping between events at one level of granularity (*e.g.*, system-call level) to events at another level of granularity. However, no systematic or automated solutions have yet been proposed for this problem.

2. **Obtaining and propagating security-labels for subjects and objects.** This is the problem of determining how security-aware server-level subjects and objects map to operating-system level subjects and objects. For example, in the case of the X server, this is the problem of determining the security-labels of X clients, `Windows`, the cut-buffer, and so on.

   Recall that in ALPEN, the reference monitor state-update function, u, as well as the process of bootstrapping security-labels in the security-aware server were still *ad hoc* manual processes. A solution to the problem of mapping security-labels for subjects and objects of the server will help automate these aspects of ALPEN as well.

   I intend to address this problem by working with a simplifying assumption: that the set of subjects in the operating system are the same as the set of subjects of the security-aware server, *i.e.*, only operating-system-level users are clients of the server. With this assumption, the problem of mapping subject labels becomes trivial, and the problem reduces to that of finding object-labels. To solve the object label mapping problem, I intend to use static analysis to identify information-flows within the server. My hypothesis is that objects which are affected along an information-flow path should then get the same security-label as the operating-system-level object that the information-flow path is initialized with.

   I will then consider the case where the subjects of a security-aware server are not necessarily operating system-level subjects, as can be expected, for *e.g.*, with a web-server.

A sound and complete policy translator is just a good starting point for further exploration in the area of policy translation. In particular, sound and complete policy translation only assures that a security-aware server will enforce the same policies on operating system-level resources as the operating system does itself. However, several open questions remain (discussed below), which I intend to explore as I begin work on policy translation:

1. **Is sound and complete policy translation possible?** It is not yet apparent to me whether sound and complete policy translation is possible. For example, it will be impossible to track individual file labels within a security-aware X server unless we observe and interpret each action of an X client. For example,

in Figure 10 it may not be possible to ascertain that the file `news` or the file `nuclear-secret` is stored in the cut-buffer. Instead, it may only be possible to determine that an *Unclassified* or `Top-Secret` file is stored in the cut-buffer.

My focus in solving the policy translation problem will be on soundness, which will guarantee security of operating system-level resources, with completeness (which ensures that the policy is functional) being a secondary concern.

2. **Policy translation does not address server-specific authorization problems.** A policy translator will not produce authorization policies to address server-specific attacks. For example, an attack in the X server where an X client can alter the window settings of another X client will not be prevented by a policy obtained via translation (even if the policy translator is sound and complete). As a result, the authorization policy obtained via translation will have to be extended to prever server-specific attacks.
3. **Server-specific users.** I will also have to address the case where the subjects using resources of a server are not necessarily defined as users of the operating system, as for instance with a web-server, where arbitrary users from the web can connect and use web-server resources (such as caches in the web-server). In this case, one possible (secure) solution is to produce an authorization policy that isolates one subject's data from another subject's data.

4 Related Work

To the best of my knowledge, there is no prior work directly related to systematically creating security-aware code from legacy code. Prior work has noted the difficulty of retofitting legacy code to add security [63], and in general, the approach has been discouraged because it introduces security as an afterthought. There is, however, a rich body of work on several aspects of authorization policies, including enforcement, formulation and analysis, as well as on the program analysis techniques that I propose to use.

4.1 Foundations of Authorization

Authorization was formalized by Lampson using the notion of an access control matrix [55]. Each column of the matrix corresponds to a system resource, and each row corresponds to a system user. The matrix entry (*sub*, *obj*) denotes the *rights* (*e.g.*, read, write, create, own) that system user *sub* has resource *obj* (the term "rights", as used in [42] is synonymous with the term "security-sensitive operation" used in this document as well as with the term "permission" used in the literature [40, 48, 79]). Given an access control matrix, it is natural to ask the following *safety* question: does a a subject *sub* have a right *r* on a resource *obj*? This problem was shown to be undecidable [42].

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. In the Discretionary Access Control (DAC) [38] model, 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 Mandatory Access Control (MAC) [9, 10] model, access rights that a user has on system resources are decided by a 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 [60, 61], have enabled the deployment of MAC in commodity operating systems as well.

4.2 Authorization Policy Enforcement

The most popular way to enforce authorization policies is using a reference monitor [3]. The reference monitor is a policy-independent entity, which observes events of interest from a monitored program. For each event, it consults the authorization policy to determine if the event is allowed or not. If allowed, the reference monitor updates its state appropriately. Otherwise, it takes appropriate action, either by terminating execution of the monitored application, or by auditing the failed authorization request.

The reference monitor model has been adopted by the Linux security modules (LSM) framework [88], which provides a generic framework to write and enforce mandatory access control policies for the Linux kernel. LSM places *hooks* to a reference monitor at several pre-defined locations (where security-sensitive operations are performed) in the kernel. The reference monitor itself is implemented as a loadable kernel module.

Hooks are placed manually in LSM using *ad hoc* techniques. Not surprisingly, vulnerabilities were found in hook placement [47, 93]. The hook placement was found to violate complete mediation [76], and the design of the hook interface left room for TOCTTOU vulnerabilities [11, 93]. This example underscores the need for systematic techniques to retrofit legacy code for reference monitoring. As I described earlier in this document, the first study that I conducted was on the feasibility of automatic placement of authorization hooks in the Linux kernel [33]. That study served as a vehicle to understand the viability of static analysis to retrofit legacy code for authorization policy enforcement.

However, in this study I identified root-causes of kernel-specific security-sensitive operations manually. I was soon convinced that for the technique to scale to user-space server applications, automated root-cause identification was necessary—AID is a response to this need. Extensive research has been conducted in the area of root-cause analysis. Most existing work has focused on the root-cause of bugs [5, 16, 41, 58, 92]. These techniques differentiate between "good" and "bad" executions of a program to find the root-cause of the bug. The most important difference between these techniques and AID is that AID *uses a much richer set of labels*, namely, arbitrary tangible side-effects, rather than just "good" or "bad", to classify the traces it generates. In contrast to prior work, which typically uses a program crash as the only tangible side-effect, AID traces use a variety of application-specific tangible side-effects. Another technique used for root-cause analysis is dynamic slicing [2, 54, 94]. Using data-flow analysis, dynamic slicing techniques can be used to work backwards from the effect of a vulnerability, such as a program crash, to the root-cause of the vulnerability. AID can also be adapted to use dynamic slicing; however, dynamic slicing requires construction of program dependence graphs, which AID currently does not do.

Java's security mechanism [37] is also conceptually similar to the LSM framework, and uses a reference monitor for authorization policy enforcement; the reference monitor is implemented by an object of type `AccessController`, and `AccessController.checkPermission()` calls are manually inserted at appropriate locations within the code to enforce authorization policies. The techniques presented in this paper are applicable to secure legacy Java applications as well.

Reference monitoring and code retrofitting techniques have also been used to enforce safety policies in legacy code. Inlined reference monitors (the PoET/PSLang framework) [26], Naccio [28], and Polymer [8] are three such frameworks, which have been used to enforce several policies on legacy code, including stack inspection [27] and control-flow integrity [1]. The most important difference between my work and these tools is that *they require the code-patterns that must be protected to be specified in the policy*. For example, the PoET/PSLang framework requires the names of security-sensitive Java methods to be mentioned in the policy. Our work does not require code-patterns to be known *a priori*; it uses AID to recover them.

Several host-based intrusion detection techniques (HIDs) also use reference monitoring to compare the execution of the application against an expected execution model (*e.g.*, [1, 30, 32, 59, 77, 86]). Our tech-

niques are applicable here as well, to infer the event interface to be monitored (unless the event interface is obvious, *e.g.*, system calls).

An alternative to reference monitoring is to ensure that code conforms to a predefined policy at compile time. This approach is adopted by language-based techniques for policy enforcement. The central problem addressed by language-based techniques is tracking information-flow through a program [4, 6, 17, 18, 19]. A popular approach is to use the type system to track, in addition to the types of variables, their security classification, as for example with the SLam Calculus [43] and the Decentralized Label Model (DLM) [66, 67, 68], which I examine in detail below.

The basic idea behind DLM is to associate an authorization policy with the data object that is to be protected. To do so, DLM associates security-labels with program variables. Each security-label is a set, each of whose members is of the form *Subject:Permissions*, which determine the set of *Permissions* that a *Subject* has on the the contents of the program variable. In DLM, authorization policies are manually specified as type annotations—a small number of annotations to certain program variables are provided. A compiler then propagates these type annotations, and uses type checking to determine that a program satisfies an end-to-end security goal, such as confidentiality. If a program fails to type-check, then the security-goal is violated. Policy violation is dealt with in one of two ways: either by changing the policy, or by explicitly allowing the security-goal violation (using a technique called *declassification* [91]).

Note that if DLM can be implemented at the full system level, *i.e.*, operating system as well as applications running on it, then modulo the problem of writing appropriate authorization policies, it is easy to ensure that end-to-end security goals are met. However, because DLM relies heavily on type-checking, it has only been implemented for a type-safe language so far (namely Jif [65], a dialect of Java).

My work is an attempt to approximate DLM at the full-system level, and a more rigorous, formal comparison of the power afforded by my approach versus the DLM approach is a topic for future research. Asbestos [21] is a recent operating system that strives to emulate DLM by labeling virtual memory pages with security labels. The most important difference between Asbestos and my work is that while Asbestos is an operating system constructed with information-flow control as a principle design goal (the *proactive* approach), my work is an attempt at retrofitting existing, commodity code for authorization policy enforcement (the *retroactive* approach).

4.3 Authorization Policy Formulation

To the best of my knowledge there is little prior work on automating the formulation of authorization policies with the intent of meeting security goals as well as application functionality goals. No work exists on formulating authorization policies for security-aware code.

Most prior work has focused on formulating policies which 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 [76]. Systrace [72] and Polgen [44] 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 [85].

There is also prior work on expressive languages in which to express authorization policies. These include PSLang [26], Polymer [8], the SELinux policy language [79], and languages used by trust management systems, such as SPKI/SDSI [22] and KeyNote [12], which are used for authorization in distributed environments. Prior work on formal languages to express system security goals exists, albeit in the context of authorization policy analysis. LTL has previously been used to express information-flow goals [40]. Sim-

ilarly, role-based access control [31] allows the use of constraints to express invariants that must hold as an authorization policy evolves.

References

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity: Principles, implementations and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (November 2005).

[2] Agrawal, H., and Horgan, J. R. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (June 1990).

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

[4] Andrews, G., and Reitman, R. An axiomatic approach to information-flow in programs. *ACM Transactions on Programming Languages and Systems 2*, 1 (1980).

[5] Ball, T., Naik, M., and Rajamani, S. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the ACM Conference on the Principles of Programming Languages* (January 2003).

[6] Banatare, J., Bryce, C., and Metayer, D. L. Compile-time detection of information-flow in sequential programs. In *Proceedings of the 1994 European Symposium on Research in Computer Security* (1994).

[7] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles* (October 2003).

[8] Bauer, L., Ligatti, J., and Walker, D. Composing security policies with Polymer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2005).

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

[10] Biba, K. J. Integrity considerations for secure computer systems. Tech. Rep. MTR-3153, MITRE, 1977.

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

[12] Blaze, M., Feigenbaum, J., Ioannidis, J., and Keromytis, A. The KeyNote trust management system, version 2, 1999. Internet RFC 2704.

[13] Brewer, D. F. C., and Nash, M. J. The Chinese Wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy* (May 1989).

[14] Chilimbi, T., Hill, M., and Larus, J. Cache-conscious structure layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation* (1999).

[15] Clark, D. D., and Wilson, D. A comparison of military and commercial security policies. In *1987 IEEE Symposium on Security and Privacy* (May 1987).

[16] Cleve, H., and Zeller, A. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering* (May 2005).

[17] Denning, D. *Secure Information Flow in Computer Systems*. PhD thesis, Purdue University, 1975.

[18] Denning, D. A lattice model of secure information flow. *Communications of the ACM 19*, 5 (1976), 236–242.

[19] Denning, D., and Denning, P. Certification of programs for secure information flow. *Communications of the ACM 20*, 7 (1977), 504–513.

[20] Downs, D. D., Rub, J. R., Kung, K. C., and Jordan, C. S. Issues in discretionary access control. In *Proceedings of the 1985 IEEE Symposium on Security and Privacy* (April 1985).

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

[22] Ellison, C. M., Frantz, B., Lampson, B., Rivest, R. L., Thomas, B. M., and Ylonen, T. SPKI certificate theory, September 1999. Internet RFC 2693.

[23] Engler, D., and Ashcraft, K. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating System Principles* (October 2003).

[24] Engler, D., Chelf, B., Chou, A., and Hallem, S. Checking system rules using system-specific programmer-written compiler extensions. In *Proceedings of the 2000 USENIX Symposium on Operating System Design and Implementation* (December 2000), pp. 1–16.

[25] Epstein, J., and Picciotto, J. 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).

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

[27] Erlingsson, U., and Schneider, F. B. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy* (May 2000).

[28] Evans, D., and Twyman, A. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy* (May 1999).

[29] Fedora Core 4 Linux distribution. http://fedora.redhat.com.

[30] Feng, H. H., Giffin, J. T., Huang, Y., Jha, S., Lee, W., and Miller, B. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy* (May 2004).

[31] Ferraiolo, D. F., and Kuhn, D. R. Role-based access control. In *15th National Computer Security Conference* (1992).

[32] Forrest, S., Hofmeyr, S., Somayaji, A., and Longstaff, T. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (May 1996).

[33] Ganapathy, V., Jaeger, T., and Jha, S. Automatic placement of authorization hooks in the Linux security modules framework. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (November 2005).

[34] Ganapathy, V., Jaeger, T., and Jha, S. Retrofitting legacy code for authorization policy enforcement. Tech. Rep. TR-1544, Computer Sciences Department, University of Wisconsin-Madison, November 2005. Submitted to the 2006 IEEE Symposium on Security and Privacy (in review).

[35] Garfinkel, T., and Rosenblum, M. A virtual machine introspection-based architecture for intrusion detection. In *Proceedings of the 2003 ISOC Symposium on Networked and Distributed System Security* (February 2003).

[36] Hardened gentoo linux distribution. http://www.gentoo.org/proj/en/hardened.

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

[38] Graham, G. S., and Denning, P. J. Protection — principles and practice. In *AFIPS Spring Joint Computer Conference* (May 1972).

[39] Grahne, G., and Zhu, J. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations* (2003).

[40] Guttman, J. D., Herzog, A. L., Ramsdell, J. D., and Skorupka, C. W. 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).

[41] Hangal, S., and Lam, M. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering* (May 2002).

[42] Harrison, M., Ruzzo, W., and Ullman, J. D. Protection in operating systems. *Communications of the ACM* (Aug. 1976).

[43] Heintze, N., and Riecke, J. G. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th Symposium on Principles of Programming Languages* (January 1998).

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

[45] Hicks, M., and Nettles, S. Dynamic software updating. *ACM Transactions on Programming Languages and Systems* (September 2005).

[46] Examples of idioms used by TAHOE. http://www.cs.wisc.edu/~vg/papers/ccs2005a/idioms.html.

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

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

[49] Jaeger, T., Sailer, R., and Zhang, X. Resolving constraint conflicts. In *Proceedings of the 2004 ACM Symposium on Access Control Models and Technologies* (June 2004).

[50] Jaeger, T., Zhang, X., and Cacheda, F. Policy management using access control spaces. *ACM Transactions on Information and System Security (TISSEC) 6*, 3 (August 2003), 327–364.

[51] Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference* (June 2002).

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

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

[54] Korel, B., and Rilling, J. Application of dynamic slicing in program debugging. In *Automated and Algorithmic Debugging* (1997).

[55] Lampson, B. W. Protection. In *5th Princeton Conference on Information Sciences and Systems* (1971).

[56] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 2004 Symposium on Operating System Design and Implementation* (December 2004).

[57] Li, Z., and Zhou, Y. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the ACM SIGSOFT Conference on Foundations of Software Engineering* (September 2005).

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

[59] Linn, C. M., Rajagopalan, M., Baker, S., Collberg, C., Debray, S. K., and Hartmann, J. Protecting against unexpected system calls. In *Proceedings of the 14th USENIX Security Symposium* (August 2005).

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

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

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

[63] McDaniel, P., and Prakash, A. Security policy enforcement in the Antigone system. *Journal of Computer Security*. To Appear.

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

[65] Myers, A. C. Practical, mostly-static, information-flow control. In *Proceedings of the 1999 Symposium on Principles of Programming Languages* (January 1999).

[66] Myers, A. C., and Liskov, B. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles* (October 1997).

[67] Myers, A. C., and Liskov, B. Complete, safe information flow with decentralized labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy* (May 1998).

[68] Myers, A. C., and Liskov, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology 9*, 4 (October 2000), 410–442.

[69] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction* (April 2002), pp. 213–228. Available at: http://manju.cs.berkeley.edu/cil.

[70] Necula, G. C., McPeak, S., and Weimer, W. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the ACM Conference on the Principles of Programming Languages* (January 2002).

[71] Patterson, H. *Informed Prefetching and Caching*. PhD thesis, Carnegie Mellon University, October 1997.

[72] Provos, N. Improving host security with system call policies. In *Proceedings of the 2003 USENIX Security Symposium* (August 2003).

[73] Rubin, S., Bodik, R., and Chilimbi, T. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of the 2002 ACM Conference on the Principles of Programming Languages* (January 2002).

[74] Ruwase, O., and Lam, M. A practical dynamic buffer overflow detector. In *Proceedings of the 2004 ISOC Network and Distributed System Security Symposium* (February 2004).

[75] Sailer, R., Jaeger, T., Valdez, E., Caceres, R., Perez, R., Berger, S., Griffin, J. L., and van Doorn, L. Building a MAC-based security architecture for the Xen open-source hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference* (December 2005).

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

[77] Sekar, R., Bendre, M., Bollineni, P., and Dhurjati, D. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy* (May 2001).

[78] Shankar, U., Talwar, K., Foster, J., and Wagner, D. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium* (August 2001).

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

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

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

[82] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., and Lapreau, J. The Flask architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium* (August 1999), pp. 123–139.

[83] Tamches, A., and Miller, B. P. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 1999 Symposium on Operating System Design and Implementation* (February 1999).

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

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

[86] Wagner, D., and Dean, D. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy* (May 2001).

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

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

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

[90] Yan, X., Han, J., and Afshar, R. CloSpan: Mining closed sequential patterns in large datasets. In *Proceedings of the 2003 SIAM International Conference on Data Mining* (May 2003).

[91] Zdancewic, S., and Myers, A. Robust declassification. In *Proceedings of the Computer Security Foundations Workshop* (June 2001).

[92] Zeller, A. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering* (November 2002).

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

[94] Zhang, X., and Gupta, R. Cost-effective dynamic program slicing. In *Proceedings of the 26th International Conference on Software Engineering* (May 2004).