Retrofitting Legacy Code for Authorization Policy Enforcement

V G
University of Wisconsin
Madison, WI-53706
vg@cs.wisc.edu

T J
Pennsylvania State University
University Park, PA-16802
tjaeger@cse.psu.edu

S J
University of Wisconsin
Madison, WI-53706
jha@cs.wisc.edu

C S D , U W -M

T R # 1544
November 15, 2005

Abstract

Researchers have long argued that the best way to construct a secure system is to *proactively* integrate security into the design of the system. However, this tenet is rarely followed because of economic and practical considerations. Instead, security mechanisms are added as the need arises, by retrofitting legacy code. Unfortunately, existing techniques to do so are manual and adhoc, and often result in security holes in the retrofitted code.

We show that program analysis techniques can be used to securely, and largely automatically, retrofit legacy code for authorization policy enforcement. Our techniques are applicable to a large class of legacy servers, namely those that simultaneously manage multiple clients, possibly with different security labels. It is important for such servers to ensure that client interaction is governed by an authorization policy.

We demonstrate our ideas using two program analysis tools we built, A and A , which work together to automate the process of retrofitting legacy servers with mechanisms for authorization policy enforcement. We show that an X server retrofitted using these tools securely enforces authorization policies on its X clients.

NOTE: This report is superseded by our paper that appears in *Proceedings of the 2006 IEEE Symposium on Security and Privacy.* Please read that paper instead; it is available at the following URL `http://www.cs.wisc.edu/~vg/papers/ieee-sp2006`

1 I

Extensive research has been done to retrofit legacy code to add functionality and performance (*e.g.,* [9, 21, 32, 33, 39]). In contrast, relatively little research has focused on retrofitting legacy code to add security mechanisms, in particular, the ability to enforce authorization policies. Most existing research has focused on finding and fixing existing vulnerabilities (*e.g.,* [12, 31, 34, 37]). Further, researchers have traditionally argued against retrofitting existing code to add security mechanisms; rather they have advocated *proactive* security, *i.e.,* including security into the design of the software. While proactive security is unquestionably the best way to produce secure code, this tenet is often not followed, because of economic and practical considerations. As a result, there are large bodies of legacy code with little or no security mechanisms built in. Security is retroactively added as the need arises, often using adhoc, manual techniques, as was done in the case of the Linux Security Modules (LSM) framework [45]. Not surprisingly, these techniques result in security holes in retrofitted code [22, 49]. Even if proactive security is a design goal, adhoc techniques used in implementation can result in vulnerabilities [24]. Thus, it is desirable to have automated techniques that can securely retrofit legacy code.

We address the problem of retroactively adding security to legacy software systems. In particular, we focus on retrofitting a large class of legacy servers for authorization policy enforcement. The servers we consider simultaneously manage multiple clients. In such servers, it is paramount to ensure that the interaction between clients be governed by an authorization policy. Popular examples of such servers are the X Window server, called the *X server* [47], web-servers, middle-ware, and database servers.

We show that program analysis techniques can largely automate the process of retrofitting these servers for authorization policy enforcement. The key idea in our work is to add policy enforcement code to the servers, so that security-sensitive operations performed by the server are *completely mediated* [35] by authorization policy lookups. To do so, two important problems need to be addressed: (1) Identify all locations where the server performs security-sensitive operations, and (2) Instrument these locations, such that the appropriate policy statements are consulted before the operation is permitted. We equip the developer with two tools which largely automate these tasks:

1. **A** (assistant for root-cause identification), a hybrid static/dynamic analysis tool, which helps the developer identify all locations in the server where security-sensitive operations are performed. The key idea behind A is that each security-sensitive operation is typically characterized by certain canonical code-patterns being executed by the server. Formally, the execution of these code-patterns is the *root-cause* for the security-sensitive operation. The main challenge then is to find succinct root-causes for security-sensitive operations, *i.e.,* a small set of code-patterns that must be executed for a security-sensitive operation to be performed. We identify root-causes by making a novel observation: security-sensitive operations are typically associated with tangible side-effects. Thus, by tracing the server as it performs a side-effect, and analyzing the code-patterns in the trace, we can extract the root-cause of the security-sensitive operation associated with the side-effect.

   A operates in two phases. In the first phase, it traces the server, and identifies root-causes for security-sensitive operations, using the observation discussed above. In the second phase, it employs static program analysis to find all locations in the code of the server where these patterns occur; each of these locations is deemed to perform the security-sensitive operation.

   For example, consider the X server: the security-sensitive operation Window_Create creates a window (a tangible side-effect) for an X client. By analyzing the trace generated by the X server as it opens a client window on the screen, A identifies that a call to the function `CreateWindow`, which is implemented in the X server, is the root-cause of Window_Create. Indeed, this function allocates memory for, and initializes, a variable of type `Window` in response to a client request. Thus, each call to `CreateWindow` in the X server results in Window_Create.

2. **A** (assistant for legacy code policy enforcement), a static analysis tool, which instruments locations discovered by A . The key idea behind A is to instrument the server with calls to a reference monitor, which encapsulates the policy to be enforced. This ensures that a security-sensitive operation is performed only if allowed by the authorization policy. Formally, A ensures that security-sensitive operations are completely mediated by authorization policy lookups.

However, the above techniques are applicable only if the legacy code satisfies certain (easily verifiable) assumptions, which we lay out in Section 2.

Retrofitting the X server: We demonstrate our ideas by using these tools to retrofit the X server. The X server accepts connections from multiple X clients. We assume that the X server and X clients run on security-enhanced operating systems, such as SELinux [29] or Asbestos [11], which are capable of enforcing mandatory access control (MAC). Thus, the X clients will have associated *security-labels*, such as *Top-secret*, or *Unclassified*. The idea is that the authorization policy is expressed in terms of the security-labels of *subjects*, who request security-sensitive operations to be performed, and *objects*, which are the entities upon which the operations are to be performed. We show that the retrofitted X server enforces authorization policies on the X clients based upon their security-labels.

An astute reader may ask why the existing policy enforcement mechanism in the security-enhanced operating system, upon which the server runs, is insufficient to enforce authorization policies on the clients, and why the server needs to be retrofitted. The answer is that the server may provide channels of communication between clients which are not readily visible to the operating system. In the case of the X server, a "cut" operation from a *Top-secret* window, and a "paste" operation to an *Unclassified* window, violates the Bell-LaPadula policy [6]. "Cut" and "paste" are X server-specific channels for X client communication. While these operations have a kernel footprint, they are not as readily visible in the operating system, as they are within the X server, where they are primitive operations. It is not advisable in such cases to use the operating system to enforce authorization policies, because it must be modified to be made aware of kernel footprints of X server-specific operations, which introduces application-specific code into the operating system. In addition, the X server must also be modified to expose more information to the operating system, such as internal data structures which will be affected by the requested operation. It has been argued that this is impractical [25].

It is also worth noting that the X server has an existing mechanism, called the X security extension [44], 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.

Finally, we note that an effort to manually retrofit the X server was initiated by the NSA in early 2003 [25], and a retrofitted X server was produced only recently [38]. Similar efforts in the context of the LSM framework also took about two years. The techniques that we develop have the potential to reduce the turnaround time of such projects.

Contributions: To summarize, our main contribution is in showing that *program analysis techniques can be used to largely automate the process of securely retrofitting legacy servers for authorization policy enforcement*. We present two tools, A and A , and demonstrate their effectiveness by using them to retrofit the X server. A uses a novel technique for finding the root-cause (in our case, canonical code-patterns) of each security-sensitive operation, by analyzing traces produced by the server when a corresponding tangible side-effect is induced. A automatically instruments the server with calls to a reference monitor to achieve complete mediation. We show that the retrofitted X server can enforce authorization policies on X clients.

2 A O A

Our goal is to enforce an authorization policy $\mathcal{A}$ on the security-sensitive operations requested by a client C that connects to a server $\mathcal{X}$. In this section, we show how our techniques can be used to securely retrofit $\mathcal{X}$ to do so. We begin by defining our threat model.

2.1 T M

We make several assumptions:

1. The server $\mathcal{X}$ itself is not adversarial, *i.e.,* it is not written with malicious intent, and does not actively try to defeat retroactive instrumentation. Thus, we assume that $\mathcal{X}$ does not automatically remove, or modify the instrumentation that we insert. 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. We also require 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.

2. Existing vulnerabilities, such as buffer-overflow vulnerabilities, could possibly be exploited by malicious hackers to bypass our instrumentation. Such cases are currently beyond the scope of our techniques. While we cannot hope to eliminate these vulnerabilities statically, one way to defend against them is to protect $\mathcal{X}$ using techniques such as CCured [31], Cyclone [23], or runtime execution monitoring (*e.g.,* [1, 13, 16, 17, 28, 36, 42]), which terminate execution when the behavior of $\mathcal{X}$ differs from its expected behavior.
3. The environment that $\mathcal{X}$ runs in cooperates with $\mathcal{X}$ to enforce authorization policies, and is not malicious in intent. In particular, $\mathcal{X}$ 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 the operating system, the server relies on the operating system for important information, such as the security-labels associated with the clients.
4. Clients cannot communicate directly with each other, and their communication is mediated by the server $\mathcal{X}$ or the operating system. If client communication is mediated by the operating system, then the policy $\mathcal{A}$ can be enforced by the operating system itself, as is done in SELinux, and Asbestos. Thus, we restrict ourselves to the case where communication is mediated by the server $\mathcal{X}$. We also note that if the clients communicate via the operating system, they cannot avail of server-specific security-sensitive operations, such as "cut" and "paste" in the case of the X server. Thus our goal is to enforce authorization policies on server-specific security-sensitive operations requested by clients.
5. Client-server communication is not altered by any intervening software layers. For example, most commercial deployments of the X server are accompanied by a *window manager*, such as `gnome` or `kde`, in the interest of usability. Because the window manager controls how clients connect to the X server, it can in theory, alter any information exchanged between the X server and its clients. However, because window managers are few in number (unlike X clients), we assume that they can be verified, and certified, to satisfy the above assumption. Further, the operating system can ensure that only certified window managers are allowed to run with the X server.

In summary, it suffices to ensure that the operating system is in the trusted computing base. It then bootstraps security by ensuring that the instrumentation inserted in the server is not tampered with. The clients are not trusted, and could be malicious. Because the operating system handles client connections, it oversees any interaction that the clients may have at the operating system level. Similarly, client security information is bootstrapped by the operating system during client connection, and is stored within the server, thus ensuring that clients cannot tamper with their security information after connection has been established. As we will describe in the rest of this paper, client interaction at the server level is mediated by the instrumentation we add, thus ensuring that interactions between the clients are in accordance with the authorization policy.

2.2 S R

We enable authorization policy enforcement by retrofitting a server $\mathcal{X}$ to ensure that each security-sensitive operation performed by $\mathcal{X}$ is mediated, and approved, by $\mathcal{A}$. We do so using a reference monitor [3].

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 as mentioned earlier, 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: $\Sigma\times\mathcal{S}\times\mathcal{A}\rightarrow\mathsf{Bool}$ 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

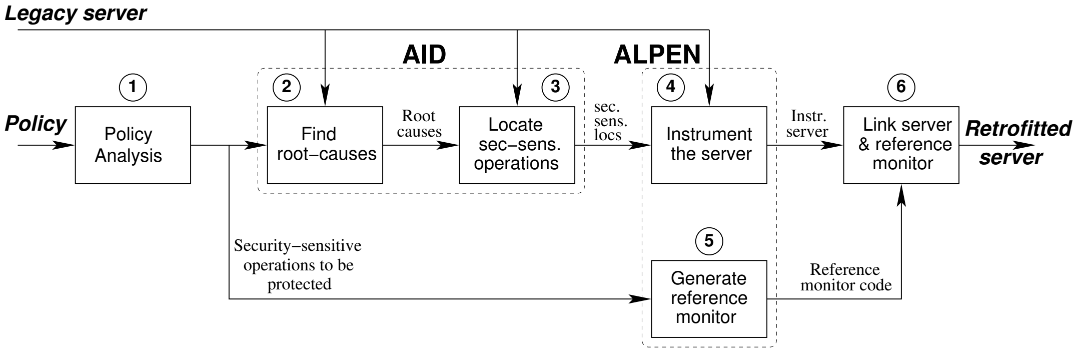

Figure 1: Steps involved in retrofitting a server for authorization policy enforcement.

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. Thus, enforcing authorization policies entails implementing $\mathcal{E}$ and $\mathcal{M}$.

- ***Requirement 1: security of the enforcer.*** The enforcer must meet two goals: (1) Ability to monitor all security events generated by $\mathcal{X}$, and (2) Ability to take action if a security event generated by $\mathcal{X}$ results in authorization failure. The action may be to terminate $\mathcal{X}$, or to audit the failure.
  - ***Requirement 1.1:*** To monitor security events, the enforcer must be able to intercept the security-sensitive operation to be performed by $\mathcal{X}$, the security-label of the subject which requests the operation, and the object upon which the operation is to be performed.
  - ***Requirement 1.2:*** To take preventive action if the security-sensitive operation is not permitted by the authorization policy, the enforcer must be able to control the execution of clients of $\mathcal{X}$.
- ***Requirement 2: security of the reference monitor.*** The main task of the reference monitor is to ensure that the state $\mathcal{S}$ of the reference monitor is not tampered with. In addition, $\mathcal{S}$ must be updated appropriately using u. Implementing f is easy, because it merely entails looking up the policy. Commercial implementations of policy management libraries are available, such as Tresys Polserver [40]; thus implementing f reduces to calling the appropriate function from the API of the library. Finally, the reference monitor must also ensure that the policy $\mathcal{A}$ is tamper proof.

2.3 O A

We present a high-level, informal overview of our approach, and show how it meets the security requirements above. Details omitted from this section appear in Section 3 and Section 4. Our approach proceeds in six steps, as shown in Figure 1. Where applicable, we illustrate the technique using an example from the X server.

Step 1: Find security-sensitive operations to be protected. The first step, that of determining the security-sensitive operations to be protected, can proceed in one of two ways. Suppose that the server $\mathcal{X}$ must be retrofitted to enforce a particular authorization policy $\mathcal{A}$. Recall that $\mathcal{A}$ is a set of triples $\langle sub, obj, op\rangle$. In this case, the set of operations can be recovered automatically from $\mathcal{A}$ as $\{op_i\}$, where $\langle sub_i, obj_i, op_i\rangle \in \mathcal{A}$.

On the other hand, a design team can manually specify the set of security-sensitive operations. This is typically done by considering a wide range of policies that $\mathcal{X}$ must enforce, and determining the set of security sensitive operations based upon these policies. In this case as well, the set of security-sensitive operations can be recovered by studying each authorization policy, as above.

In this paper, we assume that the set of security-sensitive operations is given. For the X server, we used the set of operations identified manually by Kilpatrick *et al.* [25]. This set of operations, 59 in number, considers security-sensitive operations on several key X server resources, including `Client`, `Window`, `Font`, `Drawable`, `Input`, and `xEvent`. They identify 22 security-sensitive operations on the `Window` data structure, such as Window_Create, Window_Map, and Window_Enumerate. Our techniques are parameterized on the set of security-sensitive operations, and additions or deletions from this set do not affect any of our algorithms.

Step 2: Infer root-cause of security-senstive operations. The second step identifies the root-cause of each security-sensitive operation. As mentioned in the introduction, the server executes certain canonical code-patterns when it performs a security-sensitive operation, and 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 us achieve 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. Our 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, we 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 Window_Enumerate as *Read* `WindowPtr->firstChild` and *Read* `WindowPtr->nextSib` and `WindowPtr` $\neq$ `0`, which are intuitively performed during linked-list traversal. Note that our technique can express code-patterns at the granularity of reads and writes to individual fields of data structures. We discuss the tracing infrastructure, and algorithms to compare traces to identify root-causes in more detail in Section 3.1.

Step 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 2, 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. We automatically identify the set of security-sensitive operations performed by each function call using static analysis, as described in Section 3.2.

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

Figure 2: `MapSubWindows`

In addition to identifying the locations where security-sensitive operations occur, in this step we also identify the subject and object associated with the operation. To do so, we identify the variables corresponding to subject and object data types (such as `Client` and `Window`) in scope. In most cases, this heuristic precisely identifies the subject and the object. In Figure 2, 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. These steps are realized in A .

Step 4: Instrument the server. Once A 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 A 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 A 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, authorization policies are expressed in terms of security-labels of subjects and objects. Security-labels can be stored in a table within the reference monitor (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 we pass both the subject and the object to the reference monitor using `query_refmon`, the reference monitor can lookup the corresponding security-labels, and consult the policy.

Step 5: Generate reference monitor code. This step generates code for the `query_refmon` function. We 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 Polserver [40]). 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 ⟨`pClient`, `pWin`, Window_Create⟩ succeeds, corresponding to creation of a new window, the security-label of `pWin`, the newly-created window, must be initialized appropriately. Similarly, a security-event which copies data from $\texttt{pWin}_1$ to $\texttt{pWin}_2$ may entail updating the security-label of $\texttt{pWin}_2$ (for *e.g.,*, under the Chinese-Wall policy [8]). 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 4. 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 A , are realized in the tool A .

Step 6: Link the modified server and reference monitor. The last step involves linking the retrofitted server and the reference monitor code to create an executable that can enforce authorization policies.

We now examine how our approach meets the security requirements from Section 2.2. In our approach, the enforcer $\mathcal{E}$ is implemented using the instrumentation inserted in step 4. A identifies all locations where security-sensitive operations are performed, and step 4 inserts instrumentation at these locations. Further, the subject, object, and the operation that constitute the security-event at the location are passed to the reference monitor. Because the security-labels of the subject, and the object are stored in the data structures that represent them, our approach satisfies requirement 1.1. The design of our instrumentation ensures that if the security-event is not authorized, the server $\mathcal{X}$ handles failed authorizations appropriately. In particular, **handle_failure** can be used to terminate the execution of malicious clients. Thus, our approach satisfies requirement 1.2. Our approach meets requirement 2 because the retrofitted server runs as a separate process. In particular, because the state of the reference monitor is stored internally in data structures private to the server, it is tamper-proof. Finally, we ensure that the policy $\mathcal{A}$ is tamper-proof by storing it on the file-system with permissions such that only a privileged system user can modify it.

A noteworthy feature of our approach is its modularity. In particular, alternate implementations of root-cause analysis (such as dynamic slicing [2]) and instrumentation can be used in place of A and A , respectively. Thus, our technique benefits directly from improved algorithms for these tasks.

2.4 L A

While we have so far identified security requirements for retroactive enforcement, in the rest of this paper, we describe largely automatic techniques to meet these requirements. Our automated techniques however, have limitations, partly fundamental, and partly artifacts of our current implementation. We discuss them here:

1. ***Binary executables.*** A key step in our analysis is that of identifying the root-cause of each security-sensitive operation. In our case, the root-cause is the set of canonical code-patterns that characterize each security-sensitive operation. Currently these code patterns are expressed in terms of abstract syntax trees (ASTs) at the source code level, thus constraining our analysis to work with source code. However, this is not a fundamental limitation, and can be overcome by modifying the language in which code patterns are expressed.
2. ***Obfuscated code.*** Identifying individual code-patterns might be possible in obfuscated code. However, the root-cause of the security-sensitive operation may be the execution of several of these code-patterns together. Because of obfuscation, it will be harder to identify if all the relevant code-patterns are executed together, without doing a deeper control-flow analysis.

   In addition, a security event also consists of the subject and the object associated with the security sensitive operation. Our analysis does so by first identifying security-sensitive operations, and then locating variables in scope, which could possibly be subjects or objects. Identifying these entities becomes harder if code is obfuscated.
3. ***Encrypted code.*** Encrypted code does not lend itself to program analysis.
4. ***Self-checksumming code.*** Finally, the code we analyze must be amenable to modification, because retrofitting code involves instrumentation. Self-checksumming code may preclude modification, except by the code producer, who can instrument the code, and recompute the checksums.

3 A : A T L S - O

A 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. We discuss these steps in detail.

3.1 R - I A P T

Recall that our 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 3. Note that code-patterns are expressed in terms of abstract-syntax-trees (ASTs); this allows us to express code-patterns more generically, in terms data-structures, rather than individual variables. The root-cause of a security-sensitive operation is a conjunction of one or more code-patterns.

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

Figure 3: Code-pattern definition

For example, in the X server, the root-cause of Window_Create is *Call* `CreateWindow`, while one root-cause of 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. We address this challenge by making two novel observations.

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

Thus, if we induce a server 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 3 that it executes in the process. However, the program trace generated by the tangible side-effect may be huge. Using our tracing infrastructure, the

X server generates a trace of length 10459 when the following experiment is performed: start the X server, open an `xterm`, close the `xterm`, and close the X server. It is impossible to identify succinct root-causes by studying this trace. Our 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 we were able to reduce, on average, the size of the trace by several orders of magnitude using this technique (Figure 4), whittling down the search for root-causes to fewer than 10 functions.

A technical difficulty must be addressed before we 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).

We manually identify the security-sensitive operations generated by each tangible side-effect. Because the side-effects we consider are *tangible*, programmers typically have an intuitive understanding of the operations involved in performing the side-effect. The trace generated by the tangible side-effect is then assigned a *label* with the set of security-sensitive operations 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 we show that, somewhat surprisingly, root-causes can be identified succinctly and precisely, *inspite of errors in labeling*. Because traces can have multiple labels, we formulate *set-equations* for each label (recall that a label is just a security-sensitive operation) in terms of label-sets of all our traces.

Definition 1 (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   : F _R -C (X, S, Seff)
Input       : (i) 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.
X' := X instrumented to perform tracing;
foreach (tangible side-effect seff_i ∈ Seff) do
    T_i := Trace generated by X' when induced to perform seff_i;
    label(T_i) := Set of operations (from S) involved in seff_i;
foreach (op_i ∈ S) do
    SE_i := Set-equation for op_i in terms of label(T_1), ..., label(T_m);
    CPset_i := Set of code-patterns in T_i;
    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, we do the following: Let S be the set of all security-sensitive operations, and B = {op}. Let C_i denote the label (*i.e.,*, the set of security sensitive operations performed) of trace T_i, which is 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, we let $B = \{\text{op}\}$, and observe that $B = \mathrm{C}_1 - \mathrm{C}_2$. We perform the *same* set-operations on the set of code-patterns in T_1 and T_2 to obtain $\{p_1\}$, which is then reported as the 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.

Definition 2 (CNF-set-equation problem) *Given a set S, a set $B \subseteq S$, a collection C of subsets of S (as in Definition 1), and an integer k, does B have a CNF-set-equation with at most k clauses?*

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

3.1.1 E R -C -F A

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	115	148	251	161	68	148	96	93	166

Figure 4: Labeled traces of tangible side-effects obtained from the X server.

We have implemented Algorithm 1 in A . We use a modified version of `gcc` to compile the server. During compilation, instrumentation is inserted statically at statements which read and write to fields of critical data structures. We 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. We then induce the modified server to perform a set of tangible side-effects, and proceed as in Algorithm 1 to find root-causes.

We applied this to find root-causes of security-sensitive operations in the X server. In particular, we recorded reads and writes to fields of data structures such as `Client`, `Window`, `Font`, `Drawable`, `Input`, and `xEvent`. Figure 4 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. We manually labeled each tangible side-effect with the security-sensitive operations it performs. These entries are marked in Figure 4 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 settting its attributes (Window_Getattr, Window_Setattr), and drawing the contents of the window (Window_DrawEvent). This trace of operations contains 115 calls to distinct functions in the X server, as shown in the last row of Figure 4.

Figure 5 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.

Operation	Set Equation	|RC|	Root-cause
Create	∩(A, C, G) - D - H	9	*Call* `CreateWindow`
Destroy	∩(B, D) - A	7	*Call* `DeleteWindow`
Map	∩(A, C, G) - D - H	9	*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 - E	6	*Call* `MoveWindowInStack`, *Call* `ReflectStackChange`, *Call* `WhereDoIGoInStack`
Getattr	∩(A, C, I) - (B, D, E, F)	25	*Call* `GetWindowAttributes`
Setattr	∩(A, C, F, I) - (B, D, E)	15	*Call* `ChangeWindowAttributes`
Move	F - (A, B, D, E, G)	38	*Call* `ProcTranslateCoords`
Enumerate	∩(C, D, F, H, I)	21	*Read* `WindowPtr->firstChild` ∧ *Read* `WindowPtr->nextSib` ∧ `WindowPtr≠0`, *Read* `WindowPtr->lastChild` ∧ *Read* `WindowPtr->prevSib`
InputEvent	E - C	19	*Call* `CoreProcessPointerEvent`, *Call* `CoreProcessKeyboardEvent`, ...
DrawEvent	∩(A, B, C, D, E, F, G, H, I)	12	*Call* `DeliverEventsToWindow`

Figure 5: Root-causes obtained using labeled traces from Figure 4.

To find errors in manual labeling of traces, we 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 4. For example, we 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 4. Thus for example, we did label the trace generated by moving a window with Window_Getattr, whereas in fact, this operation is not performed when a window is moved.

We now evaluate A 's root-cause finding algorithm by answering the following questions:

1. ***How effective is* A *at locating root-causes?*** Raw-traces generated by tangible-side effects, have on average, 103967 code-patterns. However, A 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 4. The third column of Figure 5 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 A was able to achieve over one order of magnitude reduction in terms of the number of distinct functions to be examined.

   We examined each of the functions in RC to determine if it is indeed a root-cause. In most cases, we 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, we examined the traces generated by A 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` ∧ *Write* `MapNotify` *to* `xEvent->union->type`). For security-sensitive operations such as Window_Chstack, where the traces generated by A did not contain commonalities in the code-patterns exercised by different functions, we 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 A for the X server, we have manually verified that it is indeed a root-cause of the security-sensitive operation in question.

   However, in general, A need not recover all the root-causes. Because A 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 A against these traces, confidence can be increased in the precision of root-causes obtained by A . In the future, we plan to investigate static techniques to identify root-causes to overcome this limitation.

3. ***How much effort is involved in manual labeling of traces?*** In all, we collected 15 traces for different tangible side-effects exercising different `Window`-related security-sensitive operations. It took us a couple of hours to

manually label these traces with security-sensitive operations.

4. ***How effective is manual labeling of traces?*** In most cases, it is easy to reason about the security-sensitive operations that are performed if a tangible side-effect is induced. However, because this process is manual, we may miss security-sensitive operations that may be performed (★ entries in Figure 4), or label a trace with security-sensitive operations that are not actually performed by the trace (● entries). Our experience of manually labeling traces for the X server shows that this process has an error rate of approximately 15%.

   However, it must be noted that we were able to recover 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.

3.2 I S - L S A

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

A currently identifies security-sensitive locations at the granularity of function calls. Note that several, but not all, root-causes are function calls. A 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, intraprodecural analysis, as described in Algorithm 2. A 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 : F _S - _L (X, S, RC)
Input     : (i) X: Server to be retrofitted, (ii) S: Set of security-sensitive operations {op_1, ..., op_n}, and (ii) RC: Set of root-cause sets rc_1, ...,
            rc_n of op_1, ..., op_n, respectively.
Output    : Opset: X → 2^S, where Opset(f) denotes the set of security-sensitive operations performed by a call to f, a function of X.
/* Process root-causes with only function calls */;
foreach (root-cause set rc_i in RC) do
    rcset_i := Set of code-patterns in rc_i;
    if (rcset_i == {Call f_1, ..., Call f_m)} then
        foreach (f ∈ {f_1, ..., f_m}) do
            Opset(f) = Opset(f) ∪ {op_i};
        RC = RC - {rc_i};
/* Process other root-causes */;
foreach (function f in X) do
    Opset(f) := φ;
    CP(f) := Set of code-patterns in f (as determined using the ASTs of statements in f);
    foreach (root-cause set rc_i in RC) do
        if (rcset_i ⊆ CP(f)) then Opset(f) := Opset(f) ∪ {op_i};
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 2). This function maps all children of a given window (`pParent` in Figure 2) 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}.

A uses a slightly more powerful variant of code-pattern language in Figure 3 to match code-patterns in function bodies. In particular, it extends Figure 3 with the ability to specify relations between different instances of ASTs. Thus, for example, it can match patterns such as *Read* `WindowPtr`$_1$`->firstChild` ∧ *Read* `WindowPtr`$_2$`->nextSib`

Where WindowPtr$_1$ ≠ WindowPtr$_2$. Note that for traversing the linked list of child windows, as in Figure 2, the *parent*'s firstChild field is read, followed by nextSib of *child* windows. In the absence of this feature, A 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, A 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.

3.2.1 E S - L -F A

We have implemented A 's static analysis algorithm as a plugin to the CIL [30] toolkit. We evaluate A '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. A 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.

   We found that this was because of the inadequate expressive power of the code-pattern language. In particular, A 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, A 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.

4 A : A T P S - O

Locations identified as performing security-sensitive operations by A are protected by A using instrumentation. Because A helps recover the complete description of security-events, adding instrumentation is straightforward, and calls to query_refmon are inserted as described in Section 2. 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, A avoids TOCTTOU bugs [7] by construction.

A also generates a template implementation of query_refmon, as shown in Figure 6. 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 A . We impose no restrictions on the policy language, or the policy management framework. Figure 6 shows an example: it shows a snippet of code that is automatically generated by A . 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 Tresys' Polserver [40], 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.

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

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 [8] 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 6 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 we 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 we bootstrap security-labels in the server. As mentioned in the introduction, we assume that the server runs on a machine with a security-enhanced operating system. We use operating system support to bootstrap security-labels based upon how clients connect to the server. 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, we use the security-label of the socket (obtained from the operating system) as the security-label of the X client. We then propagate X client security-labels as they manipulate resources on the X server, as shown in Figure 6, where the client's security-label is used as the security-label for the newly-created window.

5 E A P X R X

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

Attack I: Several well-known attacks against the X server [25] rely on the ability of an X client to change properties of windows belonging to other X clients, for *e.g.,* by changing their background or content.

Policy I: "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.
Enforcement I: The call to `query_refmon` placed in the `ChangeProperty` function of the X server mediates Window_Chprop. To enforce this policy, we check that the security-label of the subject requesting the operation, and the security-label of the window whose properties are to be changed are equal.
Attack II: Operating systems can ensure that a file belonging to a *Top-secret* user cannot be read by an *Unclassified* user (the Bell-LaPadula policy [6]). However, if both the *Top-secret* and *Unclassified* users have `xterms` 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 II: "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 [44]) cannot enforce this policy if there are more than two security-levels.
Enforcement II: The cut and paste operations correspond to the security-sensitive operation Window_Chselection of the X server. A identifies the root-causes of Window_Chselection as calls to two functions, `ProcSetSelectionOwner` and `ProcConvertSelection` in the X server. It turns out that the former is responsible for the "cut" operation, and the latter for the "paste" operation. Calls to the `query_refmon` placed in these functions are used to mediate the cut and paste operations, respectively. We created three users on our machine with security-labels *Top-secret*, *Confidential* and *Unclassified*, in decreasing order of security. The X clients created by these users inherit their security-labels. We were able to successfully ensure that a cut operation from a high-security X client window (*e.g., Confidential*) can only result in a paste into X client windows of equal or higher security (*e.g., Top-secret* or *Confidential*).

5.1 P R X

We measured the runtime overhead imposed by instrumentation by running an X server retrofitted with our instrumentation, and without, on 25 `x11perf` [46] benchmarks. We ran the retrofitted X server with a null policy, *i.e.,* all authorization requests succeed, to measure overhead (defined as $\frac{Time\ in\ retrofitted\ server}{Time\ in\ vanilla\ server}\times 100 - 100$). Overhead ranged from 0% to 18% across the benchmarks, with an average overhead of 2%.

6 R W

The research presented here is related to work in several other areas.
Techniques for Authorization Policy Enforcement: Reference monitors [3] have traditionally been used to enforce authorization policies, both when enforcement mechanisms are added proactively, or retroactively.

The Asbestos operating system [11] is a good example of proactively adding security. Asbestos incorporates several mechanisms, including security-labels, and techniques to isolate user data, so as to contain the effects of exploits. The operating system acts as the reference monitor, and enforces an authorization policy using security-labels. Security-enhanced Linux (SELinux) [29] was also conceived with similar goals in mind, but started off by retrofitting the Linux kernel to enforce authorization policies. It is currently architected using the Linux Security Modules framework (LSM) [45]. LSM retrofits the Linux kernel to enforce mandatory access control policies. It consists of a reference monitor implemented as a loadable module, which encapsulates a policy, and presents a *hook* interface. These hooks are called from appropriate locations in the Linux kernel.

However, hooks are placed manually in LSM using adhoc techniques. Not surpisingly, vulnerabilities were found in hook placement [22, 49]. The hook placement was found to violate complete mediation [35], and the design of the interface left room for TOCTTOU vulnerabilities [7, 49]. This example underscores the need for systematic techniques to retrofit legacy code for reference monitoring.

Recently, Ganapathy *et al.* [18] have proposed automatic static analysis techniques to match-make hooks with kernel locations. However, their technique suffers from two important limitations. First, they assume that the reference monitor implementation is available, and use static analysis of the reference monitor to infer what operation each hook authorizes. Second, they require manually-written *idioms*, which are akin to root-causes of security-sensitive operations. We provide automatic techniques to generate the reference monitor, and to aid the developer in finding root-causes of security-sensitive operations. Further, A can be used to automatically identify idioms.

Java's security mechanism [19] is also conceptually similar to the LSM framework; 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.

Languages and Techniques for Safety Policy Enforcement: Reference monitoring and code retrofitting techniques have also been used to enforce safety policies in legacy code. Inlined reference monitors (the PoET/PSLang framework) [13], Naccio [15], and Polymer [5] are three such frameworks, which have been used to enforce several policies on legacy code, including stack inspection [14] and control-flow integrity [1]. The most important difference between our 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 A 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, 16, 17, 28, 36, 42]). Our techniques are applicable here as well, to infer the event interface to be monitored (unless the event interface is obvious, *e.g.,* system calls).

Root-cause Analysis: Extensive research has been conducted in the area of root-cause analysis. Most existing work has focused on the root-cause of bugs [4, 10, 20, 27, 48]. 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 A is that A *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, A traces use a variety of application-specific tangible side-effects. Another technique used for root-cause analysis is dynamic slicing [2, 26, 50]. 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. A can also be adapted to use dynamic slicing; however, dynamic slicing requires construction of program dependence graphs, which A currently does not do.

Security of the X server: Information-flow attacks against the X server have long been known [43]. Several techniques have been proposed to prevent such attacks, including the X security extension [44], described in the introduction. Uppuluri *et al.*[41] propose a filtering technique to regulate information-flow between X clients—the filter is a layer which interposes X client communication with the X server, and enforces policies on client interaction. While this technique has the advantage that the X server does not have to be modified, as they note themselves, the filter can be bypassed by malicious clients. Finally, manual efforts are underway to retrofit the X server to enforce SELinux-like policies [25]. The techniques proposed in this paper can help by formalizing and automating such efforts, and in the process, reducing turnaround time.

7 C

It is becoming common practice to retrofit legacy code with extra security mechanisms. However, existing techniques to do so are adhoc, and often lead to security holes in retrofitted applications. We formalized the problem of retrofitting legacy server code for authorization policy enforcement, and presented program-analysis based techniques to do so largely automatically. We demonstrated the efficacy of our techniques by using them to retrofit the X server, and showed that the retrofitted server securely enforces authorization policies on its clients.

Acknowledgements

The authors thank Mihai Christodorescu, Jonathon Giffin, Boniface Hicks, Louis Kruger, Shai Rubin, and Hao Wang for useful comments.

R

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

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

[3] A , 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] B , T., N , M., R , S. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the ACM Conference on the Principles of Programming Languages* (January 2003).

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

[6] B , D. E., L P , 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.

[7] B , M., D , M. Checking for race conditions in file accesses. *Computer Systems 9*, 2 (Spring 1996).

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

[9] C , T., H , M., L , J. Cache-conscious structure layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation* (1999).

[10] C , H., Z , A. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering* (May 2005).

[11] E , P., K , M., V D B , S., F , C., Z , D., K , E., M , D., K , F., M , R. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating System Principles* (October 2005).

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

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

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

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

[16] F , H. H., G , J. T., H , Y., J , S., L , W., M , B. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy* (May 2004).

[17] F , S., H , S., S , A., L , T. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (May 1996).

[18] G , V., J , T., J , 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).

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

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

[21] H , M., N , S. Dynamic software updating. *ACM Transactions on Programming Languages and Systems* (September 2005).

[22] J , T., E , A., Z , 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.

[23] J , T., M , G., G , D., H , M., C , J., W , Y. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference* (June 2002).

[24] K, P. A., S, 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.

[25] K, D., S, W., V, C. Securing the X Window system with SELinux. Tech. Rep. 03-006, NAI Labs, March 2003.

[26] K, B., R, J. Application of dynamic slicing in program debugging. In *Automated and Algorithmic Debugging* (1997).

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

[28] L, C. M., R, M., B, S., C, C., D, S. K., H, J. Protecting against unexpected system calls. In *Proceedings of the 14th USENIX Security Symposium* (August 2005).

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

[30] N, G. C., M P, S., R, S. P., W, 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). Available at: http://manju.cs.berkeley.edu/cil.

[31] N, G. C., M P, S., W, W. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the ACM Conference on the Principles of Programming Languages* (January 2002).

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

[33] R, S., B, R., C, 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).

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

[35] S, J. H., S, M. D. The protection of information in computer systems. *Communications of the ACM* (July 1974).

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

[37] S, U., T, K., F, J., W, D. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium* (August 2001).

[38] S, S., 2005. Personal Communication.

[39] T, A., M, 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).

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

[41] U, P., D, V., R, V. A practical framework to manage information flow between X clients, May 2004. Manuscript available at: http://www.sce.umkc.edu/~uppulurip/research/xclients.pdf.

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

[43] W, 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.

[44] W, D. Security extension specification, version 7.1, X Consortium Inc., 1996.

[45] W, C., C, C., S, S., M, J., K-H, G. Linux security modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium* (August 2002).

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

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

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

[49] Z , X., E , A., J , T. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium* (August 2002).

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