Published in Proceedings of the 2006 IEEE Symposium on Security and Privacy, Oakland, California, May 2006

Retrofitting Legacy Code for Authorization Policy Enforcement

Vinod Ganapathy [*] Trent Jaeger [†] Somesh Jha [‡]

Abstract

Researchers have 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. Existing techniques to do so are manual and ad hoc, and often result in security holes.

We present program analysis techniques to assist the process of retrofitting legacy code for authorization policy enforcement. These techniques can be used to retrofit legacy servers, such as X window, web, proxy, and cache servers. Because such servers manage multiple clients simultaneously, and offer shared resources to clients, they must have the ability to enforce authorization policies. A developer can use our techniques to identify security-sensitive locations in legacy servers, and place reference monitor calls to mediate these locations. We demonstrate our techniques by retrofitting the X11 server to enforce authorization policies on its X clients.

1. Introduction

Researchers have traditionally argued that the best way to construct secure systems is to proactively design them for security. While this is unquestionably the best way to construct secure systems, economic and practical considerations force developers to choose functionality and performance over security. As a result, commodity systems often ship with inadequate security mechanisms built in, and security is retroactively added, as the need arises. For example, this was done in the case of the Linux Security Modules (LSM) framework [39], where the Linux kernel was retrofitted with mechanisms to enforce mandatory access control policies. Similarly, several popular server applications lack mechanisms to enforce authorization policies on their clients, and there is growing interest to retrofit these servers to add such mechanisms [25, 33].

Unfortunately, existing techniques to retrofit legacy code with security mechanisms, such as the ability to enforce authorization policies, are manual and ad hoc. Not surprisingly, security holes have been found in manually-retrofitted code [22, 43]. Thus, it is desirable to have automated techniques to retrofit legacy code.

In this paper, we address the problem of retroactively adding security mechanisms to legacy software systems. We focus on techniques to retrofit a class of legacy servers for authorization policy enforcement. Examples of servers to which our techniques are applicable include window servers, such as the X server [41], middleware, web, proxy, cache, and database servers. Because these servers offer shared resources to their clients, and manage multiple clients simultaneously, they must have the ability to enforce authorization policies on their clients. For example, an X server must be able to prevent an unauthorized client from reading the contents of other client windows.

The main challenge in retrofitting a legacy server is in identifying locations where *security-sensitive operations*, *i.e.,* primitive operations on critical server resources, are performed. The idea is that having identified these locations, authorization policy lookups can be added to the server code so as to *completely mediate* these locations [32]. We develop techniques to assist (1) identification of locations in server code where security-sensitive operations are performed, and (2) instrumentation of these locations, such that the operation is performed only if allowed by an authorization policy. We have prototyped these techniques in two tools, Aid and Arm, discussed below.

1. **Aid** (assistant for fingerprint identification) is a hybrid static/dynamic analysis tool, which helps a developer identify locations in server code where security-sensitive operations are performed. The key idea behind Aid is that each security-sensitive operation is typically characterized by certain canonical code-patterns being executed by the server. We call these code-patterns the *fingerprint* of the security-sensitive operation—just as a human fingerprint identifies an individual, these code-patterns identify the security-sensitive operation. The challenge

[*] Computer Sciences Department, University of Wisconsin, Madison, WI-53706, USA; vg@cs.wisc.edu.

[†] Computer Science & Engineering Department, Pennsylvania State University, University Park, PA-16802, USA; tjaeger@cse.psu.edu.

[‡] Computer Sciences Department, University of Wisconsin, Madison, WI-53706, USA; jha@cs.wisc.edu.

is to find fingerprints for security-sensitive operations.

We identify fingerprints using 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 code-patterns in the trace, we can extract fingerprints of security-sensitive operations associated with the side-effect.

For example, consider the X server: the security-sensitive operation Window_Create creates a window (window creation is 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, Aid identifies that a call to the function `CreateWindow`, which is implemented in the X server, is the fingerprint 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.

Aid is a two-phase tool. In the first phase, it traces the server and identifies fingerprints for security-sensitive operations, as discussed above. In the second phase, it statically identifies locations in the code of the server where these fingerprints occur; each of these locations is deemed to perform the security-sensitive operation.

2. **Arm** (assistant for reference monitoring) is a tool to instrument locations discovered by Aid. In particular, Arm adds calls to a reference monitor, which encapsulates the authorization policy to be enforced. These calls, which perform authorization policy lookups, completely mediate security-sensitive locations, thus ensuring that a security-sensitive operation is performed only if allowed by the authorization policy.

While Aid and Arm are not yet fully automatic, we feel that they are an improvement over existing techniques, which are completely manual. We also note that for our techniques to be applicable, legacy code must satisfy certain assumptions, which we lay out in Section 2.

1.1. Case study: Retrofitting the X server

The X server accepts connections from multiple X clients, and manages resources (*e.g.,*windows, buffers) that it offers to these clients. Thus, it is important for the X server to enforce authorization policies on its X clients. A manual effort to retrofit the X server with authorization policy enforcement mechanisms was initiated by the NSA in early 2003 [25], and a retrofitted X server was produced only recently [35], taking approximately two years.

We demonstrate that our techniques can assist with, and potentially reduce the turnaround time of efforts to retrofit legacy servers, by performing a case study with the X server. Specifically, we retrofitted the X server to enforce mandatory access control policies on window operations requested by X clients. Using Aid and Arm, we were able to identify security-sensitive locations in the X server, and add reference monitoring code, with a few hours of manual effort. We ran the retrofitted X server on a security-enhanced operating system (SELinux [28]), so that X clients have associated *security-labels*, such as Top-secret and Unclassified. The retrofitted X server enforced authorization policies on X clients based upon their security-labels.

A question that may arise is why the server itself needs to be retrofitted to enforce authorization policies on its clients. In particular, why can't existing policy enforcement mechanisms in a security-enhanced operating system (*e.g.,* SELinux), upon which the server runs, be used to enforce these policies? The answer is that the server may provide channels of communication between clients that are not readily visible to the operating system. For example, consider enforcing a policy in the X server that disallows a cut operation from a Top-secret window followed by a paste operation into an Unclassified window. Cut and paste are X server-specific channels for X client communication. While these operations do have a kernel footprint, they are not as readily visible in the operating system as they are within the X server, where they are primitive operations. It is not advisable in such cases to use the operating system to enforce authorization policies, because it must be modified to be made aware of kernel footprints of X server-specific operations, which introduces application-specific code into the operating system. In addition, the X server must also be modified to expose more information to the operating system, such as internal data structures that will be affected by the requested operation. It has been argued that this is impractical [25].

1.2. Contributions

To summarize, our main contributions are:

- Program analysis techniques to identify security-sensitive locations in legacy code, and retrofit these locations with reference monitor calls for authorization policy enforcement.
- Prototype implementations of these techniques in two tools. Aid (Section 3) uses a novel approach based upon program tracing to find fingerprints of security-sensitive operations, and uses these fingerprints to statically find security-sensitive locations. Arm (Section 4) retrofits these locations with reference monitor calls.
- Application of these tools to retrofit the X server to enforce authorization policies on its X clients (Section 5).

More broadly, we feel that it is valuable to have *retroactive* techniques and tools, such as the ones presented in this paper, to add security mechanisms to legacy code.

2. Overview of our approach

Our goal is to enforce an authorization policy on the security-sensitive operations requested by a client that connects to a server. In this section, we show how our techniques can be used to securely retrofit the server to do so. We begin by stating our assumptions.

2.1. Assumptions

Server not adversarial. We assume that the server 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 the server does not remove, or modify the instrumentation that we insert. This can be ensured by the operating system as it loads the server for execution, by comparing a hash of the executable against a precomputed value. We also require that the server be non-self-modifying, to preclude the possibility that instrumentation is modified at runtime. This property can be enforced by making code pages write-protected.

Defense against control-hijacking exploits. Existing vulnerabilities, such as buffer-overflow vulnerabilities, could possibly be exploited by malicious hackers to bypass our instrumentation. Because we cannot hope to eliminate these vulnerabilities statically, we assume that the server is protected using techniques such as CCured [30], Cyclone [23], or runtime execution monitoring and sandboxing, which terminate execution when the behavior of the server differs from its expected behavior.

Cooperation from environment. The environment that the server runs in must cooperate with it to enforce authorization policies, and must not be malicious in intent. In particular, the server relies on the operating system for several policy enforcement tasks. First, it requires that operating system ensure that the authorization policy 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.

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

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

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. Client security information, in particular its security-label, 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 requests for security-sensitive operations are mediated by the instrumentation that we add, thus enabling enforcement of authorization policies on clients.

2.2. Basic tools

We enforce authorization policies by retrofitting a server to ensure that security-sensitive operations requested by clients are mediated and approved by an authorization policy. We do so using a reference monitor [2].

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

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

- Σ is a set of *security events*, where each security event is a triple ⟨*sub*, *obj*, *op*⟩;
- $\mathcal{S}$ is the *state* of the reference monitor, and is a set storing current associations of security-labels with subjects and objects;
- $\mathcal{U}$: $\Sigma \times \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is a *state update* function, which denotes how subject and object security-labels change in response to policy decisions;
- $\mathcal{R}$: $\Sigma \times \mathcal{S} \times \mathcal{A} \rightarrow \mathsf{Bool}$ is a *policy consulter*, which returns True if and only if a security event is permitted by the reference monitor.

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

The enforcer. An implementation of the enforcer must have the (1) ability to monitor all security events generated in response to client requests, and (2) ability to take action if a security event results in authorization failure. The action may be to terminate the client whose request resulted in the authorization failure.

1. To monitor security events, the enforcer must be able to infer the security-sensitive operation requested, the security-label of the subject that requests the operation (typically the client), and the object upon which the operation is to be performed.
2. To take preventive action if the security event is not permitted by the authorization policy, the enforcer must be able to control the execution of clients of the server, or audit the failure appropriately.

The reference monitor. An implementation of the reference monitor must ensure that the state of the reference monitor and the authorization policy are tamper-proof. In addition, the state of the reference must be updated appropriately in response to security events, using $\mathcal{U}$. Implementing $\mathcal{R}$ entails looking up the policy, and can be achieved using off-the-shelf policy management libraries, such as the SELinux policy development toolkit [36].

2.3. Our approach

In this section, we present a high-level, informal overview of our approach, and describe how we implement the enforcer and the reference monitor. 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, is manual. Typically, a design team considers security requirements for the server, and determines security-sensitive operations based upon these requirements. This approach was followed in the case of the LSM framework [39] and the X server [25], where security-sensitive operations were identified for kernel resources, and X server resources, respectively. The design team typically considers a wide range of policies to be enforced by the server. Because security-sensitive operations are typically the granularity at which authorization policies are written (a policy $\mathcal{A}$ is a set of triples of the form $\langle sub_i$, obj_i, $op_i\rangle$), the set of operations $\{op_i\}$ can be identified.

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 the `Client`, `Window`, `Font`, `Drawable`, `Input`, and `xEvent` data structures. Of these, 22 security-sensitive operations are for the `Window` data structure, such as Window_Create, Window_Map, and Window_Enumerate (we will denote security-sensitive operations in this paper using suggestive names, like the ones above).

However, only an informal description of these security-sensitive operations is provided by Kilpatrick *et al.*, and a precise code-level description of these operations is needed for enforcement. Aid, described in the next two steps, achieves this by identifying fingerprints of these operations. It must be noted that 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: Find fingerprints of security-sensitive operations. The second step identifies fingerprints of security-sensitive operations. As mentioned in the introduction, the server executes certain canonical code-patterns when it performs a security-sensitive operation, and these code-patterns are the fingerprint 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 a tangible side-effect associated with a security-sensitive operation, and trace the server as we do so, the code-patterns that form the fingerprint of the security-sensitive operation *must* be in the trace.

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

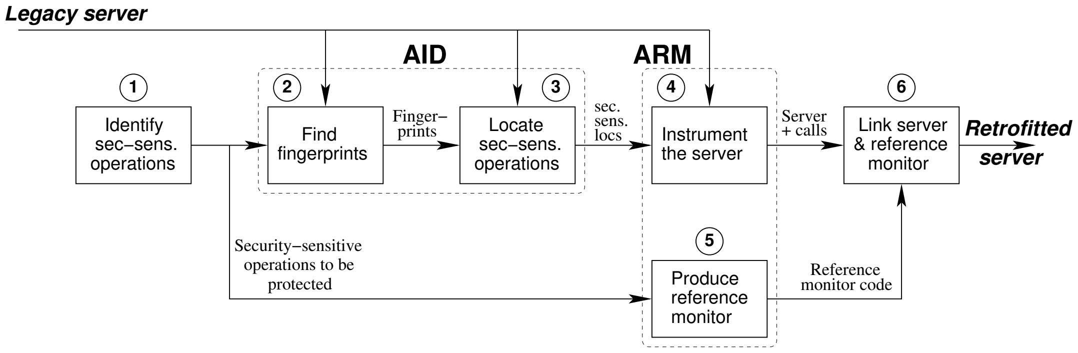
Legacy server
AID
ARM
1
Identify sec–sens. operations
2
Find fingerprints
Finger–prints
3
Locate sec–sens. operations
sec. sens. locs
4
Instrument the server
Server + calls
6
Link server & reference monitor
Retrofitted server
Security–sensitive operations to be protected
5
Produce reference monitor
Reference monitor code

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

or typing to the window. Similarly, closing a browser window is associated with closing all child windows, which involves Window_Enumerate, while typing to a window is not.

With these two observations, identifying fingerprints reduces to studying about 15 entries, on average, in a program trace. Using this technique, we identified, for example, the fingerprints of Window_Create as *Call* `CreateWindow`; of Window_Map as writes of `True` to the field `mapped` of a variable of type `Window` and `MapNotify` to the field `type` of a variable of type `xEvent`; and of Window_Enumerate as *Read* `WindowPtr->firstChild` and *Read* `WindowPtr->nextSib` and `WindowPtr` ≠ `0`, which are intuitively performed during linked-list traversal. Note that code-patterns are expressed at the granularity of reads and writes to individual fields of data structures. We discuss the tracing infrastructure, and algorithms to compare traces to identify fingerprints in more detail in Section 3.1.

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

Figure 2. X server function MapSubWindows

Step 3: Find all locations that are security-sensitive. The third step uses the results of fingerprint analysis to statically identify all locations in the server where code-patterns that form the fingerprint of a security-sensitive operation occur; each of these locations performs the operation. Consider Figure 2, which shows a snippet of code from `MapSubWindows`, a function in the X server. It contains writes of True to `pWin->mapped`, and `MapNotify` to `event.u.u.type`, as well as a traversal of the children of the window pointer `pParent`. Thus, a call to the function `MapSubWindows` performs both the operations Window_Map and Window_Enumerate. We identify the set of security-sensitive operations performed by each function call using static analysis, as described in Section 3.3.

In addition to identifying the locations where security-sensitive operations occur, in this step we also try to identify the subject and object associated with the operation. To do so, we identify the variables corresponding to subject and object data types (such as `Client` and `Window`) in scope. In most cases, this heuristic is good-enough to identify the subject and the object. In Figure 2, the subject is the client requesting the operation (`pClient`), and the object for Window_Enumerate is the window whose children are enumerated (`pParent`), and the object for Window_Map is the variable denoting the child windows (`pWin`), which are mapped to the screen.

Steps 2 and 3 together identify all locations where the server performs security-sensitive operations. These steps are realized in Aid.

Step 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 ⟨*sub*, *obj*, *op*⟩, 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(⟨ sub, obj, op⟩))
then handle_failure; else Stmt;
```

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

```
if (¬query_refmon(⟨pClient, pParent, Window_Enumerate⟩))
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, 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 pointers to 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 queries. 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 $\mathcal{R}$ using an appropriate policy management API (*e.g.,* [36]). Second, he must implement the state update function, $\mathcal{U}$, by specifying how the state of the reference monitor is to be updated. For example, when a security-event ⟨`pClient`, `pWin`, Window_Create⟩ succeeds, corresponding to creation of a new window, the security-label of `pWin`, the newly-created window, must be initialized appropriately. Similarly, a security-event that copies data from `pWin`$_1$ to `pWin`$_2$ may entail updating the security-label of `pWin`$_2$ (*e.g.,* under the Chinese-Wall policy [10]). 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 $\mathcal{R}$ and $\mathcal{U}$, 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 Arm.

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 the security of our approach.

- **The enforcer** is implemented using instrumentation inserted in Step 4. Because the subject, object, and operation are passed to the reference monitor, security-labels can be retrieved, and the authorization policy consulted. If the requested operation is not permitted by the policy, the instrumentation ensures that it will not be executed. Further, because the server controls client connections, it can use **`handle_failure`** to terminate the execution of malicious clients.
- **The reference monitor** is part of the server's address space, and is thus tamper-proof by our assumptions in Section 2.1. Alternately, the reference monitor can run as a separate process, and communicate with the server using IPC. The policy itself must be protected by storing it on the file-system with permissions such that it can be modified only by a privileged system user.

A noteworthy feature of our approach is its modularity. In particular, alternate implementations of fingerprint-finding (*e.g.,* using dynamic slicing [1, 26, 44]) and instrumentation (*e.g.,* using aspect weavers [3]) can be used in place of Aid and Arm, respectively. Thus, our technique benefits directly from improved algorithms for these tasks.

3. Locating 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 fingerprints of security-sensitive operations, in our case, combinations of code-patterns that identify an operation 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. Identifying fingerprints using analysis of program traces

Recall that our ultimate goal is to retrofit a legacy server to ensure that policy lookups completely mediate security-sensitive operations. A necessary step in this process is to locate where security-sensitive operations are performed. We use fingerprints of security-sensitive operations for this task.

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 generically in terms data-structures, rather than individual variables. The fingerprint of a security-sensitive operation is defined to be 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 fingerprint of Window_Create is *Call* `CreateWindow`, while one fingerprint of Window_Enumerate, which enumerates all the children of a window is (`WindowPtr` ≠ `0` ∧ *Read* `WindowPtr->firstChild` ∧ *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 fingerprints, corresponding to different ways of performing the operation. Both forward and backwards traversal of the linked list of children of a window constitute fingerprints for Window_Enumerate, for instance.

The key challenge, however, is to discover fingerprints of security-sensitive operations, as this is often not known *a priori*—this is especially the case with legacy and third-party code. Further, fingerprints must be succinct, *i.e.,* a fingerprint must be a small combination of code-patterns that identifies 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.*

Tangible side-effects help us determine whether a server has performed a security-sensitive operation. Thus if we induce the server to perform a security-sensitive operation—the occurrence of a tangible side-effect denotes that the operation is performed—then the code-patterns in a fingerprint of that security-sensitive operation *must* be in the trace generated by the server. Thus, identifying fingerprints reduces to tracing the server as it performs a tangible side-effect, and recording the code-patterns from Figure 3 that it executes in the process. However, the program trace generated by the server as it performs a 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 (each of these is a tangible side-effect). It is impossible to identify succinct fingerprints of security-sensitive operations (*e.g.,* those of Window_Create and Window_Destroy) by studying this trace. Our second observation addresses this problem.

Observation 2 (Comparing traces) *Comparing a trace associated with a security-sensitive operation, against traces that are not associated with the operation yields succinct fingerprints.*

The key idea underlying this observation is that if a run of the server does not perform a security-sensitive operation, then the trace produced by the server will not contain a fingerprint of that operation. For example, the trace T_{open} that opens an X client window on the X server will contain the fingerprint of Window_Create, but the trace T_{close} that closes a window will not. Thus, T_{open} - T_{close}, a shorter trace, still contains the fingerprint of Window_Create. Continuing this process with other traces that do not perform Window_Create reduces the size of the trace to be examined even further. In fact, for the X server we were able to reduce the size of the trace several-fold using this technique (Figure 4), whittling down the search for fingerprints to about 15 functions, on average.

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

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

Definition 1 (Set equation) *Given set S, a set $B \subseteq S$, and a collection $\mathcal{C}$=$\{C_1, C_2, \ldots, C_n\}$ 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 $\mathcal{C}$, and '*' is $\cup$ or $\cap$.*

```
Algorithm : FIND_FINGERPRINT($\mathcal{X}$, $S$, Seff)
Input     : (i) $\mathcal{X}$: Server to be retrofitted, (ii) $S$: A set of
            security-sensitive operations $\{op_1, \ldots, op_n\}$, and (iii) $Seff$:
            A set of tangible side-effects $\{seff_1, \ldots, seff_m\}$.
Output    : $FP_1, \ldots, FP_n$ : Each $FP_i$ is the fingerprint of the
            security-sensitive operation $op_i$.
$\mathcal{X}' := \mathcal{X}$ instrumented to perform tracing;
foreach (tangible side-effect $seff_i \in Seff$) do
    $T_i$ := Trace generated by $\mathcal{X}'$ when induced to perform $seff_i$;
    $label(T_i)$ := Set of operations (from $S$) involved in $seff_i$;
foreach ($op_i \in S$) do
    $SE_i$ := Set-equation for $op_i$ in terms of $label(T_1), \ldots, label(T_m)$;
    $CPset_i$ := Set of code-patterns in $T_i$;
    $FP_i$ := Result when the set operations in $SE_i$ are performed on
    $CPset_1, \ldots, CPset_m$;
```

Algorithm 1: Algorithm to find fingerprints of security-sensitive operations.

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

For example, if T_1 is a trace 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 $\mathrm{C}_1 = \{\mathsf{op}, \mathsf{op}'\}$, and $\mathrm{C}_2 = \{\mathsf{op}'\}$. Say T_1 contains the set of code-patterns $\{p_1, p_2\}$, and T_2 contains the set of code-patterns $\{p_2\}$. Then to find the fingerprint of op, we let $B = \{\mathsf{op}\}$, and observe that $B = \mathrm{C}_1$ - C_2. We perform the *same* set-operations on the set of code-patterns in T_1 and T_2 to obtain $\{p_1\}$, which is then reported as the fingerprint of op. This process is formalized in Algorithm 1.

Finding set-equations is, in general, a hard problem. More precisely, define a CNF-set-equation as a set-equation expressed in conjunctive normal form, with '∩' and '∪' as the conjunction and disjunction operators, respectively. Each disjunct in the equation is a *clause*. 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).

3.2. Evaluation of fingerprint-finding algorithm

We have implemented Algorithm 1 in Aid. We use a modified version of `gcc` to compile the server. During compilation, instrumentation is inserted statically at statements that read and write to fields of 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 fingerprints.

We applied this to find fingerprints of security-sensitive operations in the X server. In particular, we recorded reads and writes to fields of data structures such as `Client`, `Window`, `Font`, `Drawable`, `Input`, and `xEvent`. Figure 4 shows a portion of the result of performing lines (1)-(4) of Algorithm 1. Columns represent traces of 9 tangible side-effects, and rows represent 11 security-sensitive operations on the `Window` data structure. We manually labeled each trace with the security-sensitive operations it performs. These entries are marked in Figure 4 using ✓ and $\times_2$. For example, opening an `xterm` on the X server includes creating a window (Window_Create), mapping it onto the screen (Window_Map), placing it appropriately in the stack of windows that X server maintains (Window_Chstack), getting and setting its attributes (Window_Getattr, Window_Setattr), and drawing the contents of the window (Window_DrawEvent). This trace of operations contains 115 calls to distinct functions in the X server, as shown in the last row of 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 fingerprints, the size of the resulting set, and the set of fingerprints is shown. Note that each security-sensitive operation can have more than one fingerprint, as for example, is the case with Window_Enumerate and Window_InputEvent.

To find errors in manual labeling of traces, we did the following. After finding fingerprints of security-sensitive operations, we checked each trace for the presence of these fingerprints. Presence of a fingerprint of a security-sensitive operation in a trace that is not labeled with that security-sensitive operation shows an error in manual labeling; such entries are marked $\times_1$ in Figure 4. For example, we did not label the trace generated by opening a browser (`htmlview`) with Window_Unmap. On the other hand, absence of fingerprints of a security-sensitive operation in a trace that is labeled with the security-sensitive operation also shows an error in manual labeling; such entries are marked $\times_2$ 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 Aid's fingerprint finding algorithm by answering the following questions:

1. **How effective is Aid at locating fingerprints?** Raw-traces generated by tangible-side effects, have on average, 53829 code-patterns. However, Aid abstracts each trace to function calls: it first identifies fingerprints 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 FP, which is the result obtained by computing the set-equation for each security-sensitive operation, to determine fingerprints. Aid was able to achieve about one order of magnitude reduction in terms of the number of distinct functions to be examined for fingerprints.

   We examined each of the functions in FP to deter-

Trace name	A	B	C	D	E	F	G	H	I
Side-effect ⟶ **Sec.-sens. Operation** ↓	open xterm	close xterm	open browser	close browser	type to window	move window	open & close twm menu	switch windows	open menu (browser)
Window_Create	✓		✓				✓		✓
Window_Destroy		✓	$\times_1$	✓			✓	$\times_1$	
Window_Map	✓		✓				✓		✓
Window_Unmap		✓	$\times_1$	✓			✓	$\times_1$	
Window_Chstack	✓		✓				✓	✓	✓
Window_Getattr	✓		✓			$\times_2$	$\times_2$		✓
Window_Setattr	✓		✓			✓	$\times_2$	$\times_1$	✓
Window_Move			$\times_1$			✓		$\times_1$	$\times_1$
Window_Enumerate	$\times_1$	$\times_1$	✓	✓		✓	$\times_1$	✓	✓
Window_InputEvent					✓	✓	✓	✓	✓
Window_DrawEvent	✓	✓	✓	✓	$\times_1$	✓	✓	✓	✓
Distinct Functions	115	148	251	161	68	148	96	93	166

Figure 4. Examples of labeled traces of tangible side-effects obtained from the X server. A "✓" entry in (*row*, *column*) denotes that the trace represented by *column* performs the security-sensitive operation represented by *row*. A "$\times_1$" or a "$\times_2$" entry denotes a mistake in manual labeling.

Operation	**Set Equation**	**|FP|**	**Fingerprint**
Window_Create	∩(A, C, G) - D - H	9	*Call* `CreateWindow`
Window_Destroy	∩(B, D) - A	7	*Call* `DeleteWindow`
Window_Map	∩(A, C, G) - D - H	9	*Write* True *to* `Window->mapped` ∧ *Write* `MapNotify` *to* `xEvent->union->type`
Window_Unmap	∩(B, D) - A	7	*Write* `UnmapNotify` *to* `xEvent->union->type`
Window_Chstack	∩(A, C, G, H, I) - D - E	6	*Call* `MoveWindowInStack`
Window_Getattr	∩(A, C, I) - B - D - E - F	25	*Call* `GetWindowAttributes`
Window_Setattr	∩(A, C, F, I) - B - D - E	15	*Call* `ChangeWindowAttributes`
Window_Move	F - A - B - D - E - G	38	*Call* `ProcTranslateCoords`
Window_Enumerate	∩(C, D, F, H, I)	21	*Read* `WindowPtr->firstChild` ∧ *Read* `WindowPtr->nextSib` ∧ `WindowPtr≠0`, *Read* `WindowPtr->lastChild` ∧ *Read* `WindowPtr->prevSib`
Window_InputEvent	E - C	19	*Call* `CoreProcessPointerEvent`, *Call* `CoreProcessKeyboardEvent`, *Call* `xf86eqProcessInputEvents`
Window_DrawEvent	∩(A, B, C, D, E, F, G, H, I)	12	*Call* `DeliverEventsToWindow`
	Average value of |FP|:	**15.3**	

Figure 5. Fingerprints obtained by applying Algorithm 1 to the labeled traces from Figure 4.

mine if it is indeed a fingerprint. In most cases, we found that for a security-sensitive operation, a single function in FP performs the operation. However, in some cases, multiple functions in FP seemed to perform the security-sensitive operation. For example, both *Call* `MapWindow` and *Call* `MapSubWindow`, which were present in FP, performed Window_Map. In such cases, we examined the traces generated by Aɪᴅ to determine common code-patterns exercised by the call to these functions. Doing so for Window_Map reveals that the common code-patterns in `MapWindow` and `MapSubWindow` are (*Write* True *to* `Window->mapped` ∧ *Write* `MapNotify` *to* `xEvent->union->type`). For security-sensitive operations such as Window_InputEvent, where we did not find common code-patterns exercised by candidate functions from FP, we deemed each of these function calls to be fingerprints of the operation.

2. **How precise are the fingerprints found?** For each of the fingerprints recovered by Aɪᴅ for the X server, we manually verified that it is indeed a fingerprint of the security-sensitive operation in question.

   However, in general, Aɪᴅ need not recover all fingerprints of a security-sensitive operation. Because Aɪᴅ is a runtime analysis, it can only capture the fingerprints of a security-sensitive operation exercised by the runtime traces, and may miss *other* ways to perform the operation. By collecting traces for a larger number of tangible side-effects, and verifying the fingerprints collected by Aɪᴅ against these traces, confidence can be increased in the precision of fingerprints obtained by Aɪᴅ. In the future, we plan to investigate static techniques to identify fingerprints 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 few hours to manually label 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 ($\times_1$ entries in Figure 4), or erroneously label a trace with security-sensitive operations that are not actually performed ($\times_2$ entries). Our experience of manually labeling traces for the X server shows that this process has an error rate of approximately 15%.

   However, it must be noted that we were able to recover fingerprints precisely *in spite of labeling errors*. If a security-sensitive operation is wrongly omitted from the labels of a trace that performs a tangible side-effect associated with that operation (the $\times_1$ case), then because the same security-sensitive operation often appears in the labels of other traces, a set-equation can still be formulated for the operation, and the fingerprint can be recovered. On the other hand, if a security-sensitive operation is wrongly added to the labels of a trace (the $\times_2$ case), none of the functions in FP will perform the tangible side-effect. In this case, trace labels are refined, and the process is iterated until a fingerprint is identified.

3.3. Identifying security-sensitive locations using static analysis

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

Aid currently identifies security-sensitive locations at the granularity of function calls. Note that several, but not all, fingerprints are function calls. Aid considers fingerprints that are not function calls, such as those of Window_Map, Window_Unmap, and Window_Enumerate, and identifies functions that contain these code-patterns. The idea is that by mediating calls to functions that contain these patterns, the corresponding security-sensitive operations are mediated as well. This is done using a flow-insensitive, 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 fingerprints 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 fingerprints contain only function calls, Algorithm 2 marks each of these functions as performing the operation. Aid also supports interprocedural search for code-patterns (for fingerprints that cross procedure boundaries); however, we did not encounter any such fingerprints for the X server.

```
Algorithm : FIND_SECURITY-SENSITIVE_LOCATIONS($\mathcal{X}$, $S$, $\mathcal{FP}$)
Input     : (i) $\mathcal{X}$: Server to be retrofitted, (ii) $S$: Set of
            security-sensitive operations $\{\mathsf{op}_1, \ldots, \mathsf{op}_n\}$, and (ii) $\mathcal{FP}$:
            Set of fingerprint sets $\mathrm{fp}_1, \ldots, \mathrm{fp}_n$ of $\mathsf{op}_1, \ldots, \mathsf{op}_n$,
            respectively.
Output    : Opset: $\mathcal{X} \rightarrow 2^S$, where Opset($f$) denotes the set of
            security-sensitive operations performed by a call to $f$, a
            function of $\mathcal{X}$.
 /* Process fingerprints with only function calls */;
foreach (fingerprint set $\mathrm{fp}_i$ in $\mathcal{FP}$) do
    $\mathrm{fpset}_i$ := Set of code-patterns in $\mathrm{fp}_i$;
    if ($\mathrm{fpset}_i$ == {Call $\mathtt{f}_1, \ldots,$ Call $\mathtt{f}_m$)} then
        foreach ($f \in \{\mathtt{f}_1, \ldots, \mathtt{f}_m\}$) do
            Opset($f$) = Opset($f$) $\cup$ $\{\mathsf{op}_i\}$;
        $\mathcal{FP} = \mathcal{FP}$ - $\{\mathrm{fp}_i\}$;

 /* Process other fingerprints */;
foreach (function $f$ in $\mathcal{X}$) do
    Opset($f$) := $\phi$;
    CP($f$) := Set of code-patterns in $f$ (as determined using the ASTs of
    statements in $f$);
    foreach (fingerprint set $\mathrm{fp}_i$ in $\mathcal{FP}$) do
        if ($\mathrm{fpset}_i \subseteq$ CP($f$)) then Opset($f$) := Opset($f$) $\cup$ $\{\mathsf{op}_i\}$;

return Opset;
```

Algorithm 2: Finding functions that contain code-patterns that appear in fingerprints.

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 that constitute the fingerprint of both Window_Enumerate and Window_Map. Thus, Opset(`MapSubWindows`) = {Window_Map, Window_Enumerate}.

Aid uses a slightly more powerful variant of the code-pattern language in Figure 3 to match code-patterns in function bodies. In particular, it extends Figure 3 with the ability to specify simple relations between different instances of ASTs. Thus, for example, it can match patterns such as *Read* $\mathtt{WindowPtr}_1$`->firstChild` $\wedge$ *Read* $\mathtt{WindowPtr}_2$`->nextSib` *Where* $\mathtt{WindowPtr}_1 \neq \mathtt{WindowPtr}_2$, which can be useful for matching code-patterns such as the ones in Figure 2, where the *parent*'s `firstChild` field is read, followed by `nextSib` of *child* windows.

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.

3.4. Evaluation of security-sensitive location-finding algorithm

We have implemented Aid's static analysis algorithm as a plugin to CIL [29]. We evaluate 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 20 functions reported as performing Window_Enumerate, only 10 actually do.

   We found that this was because of the inadequate expressive power of the code-pattern language. In particular, Aid matches functions that 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, we identify subjects and objects using variables of relevant data types in scope. This simple heuristic is quite effective: out of 25 functions that were identified as performing `Window` operations, the subject, of type `Client`, and object, of type `Window`, were available as formal parameters or derivable from formal parameters in 22 of them. In the remaining functions, specifically, those performing Window_InputEvent, the subject and object were derived from global variables. Even in this case, however, manual inspection quickly reveals the relevant global variables.

4. Protecting security-sensitive locations

Locations identified as performing security-sensitive operations by Aid are protected by Arm 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 in Section 2. If the function to be protected is implemented in the server itself (and not within a library), 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, Arm avoids TOCTTOU bugs [9] by construction.

```
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.

Arm 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 Arm. We impose no restrictions on the policy language, or the policy management framework. Figure 6 shows an example: it shows a snippet of code generated by Arm. Subject and object labels are stored as fields (`label`) in the data structures representing them. The statement in italics, a call to the function `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 toolkit [36], 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 $\mathcal{U}$. Note that $\mathcal{U}$ depends on the policy to be enforced; different policies may choose to update security-labels differently. Functionality to determine how security-labels must change based upon whether an authorization request succeeds or fails must ideally be provided by the policy-management tool that is used (because how security-labels change is policy-dependent).

   However, if this functionality is not available in the policy-management tool used, the developer must update the state of the reference monitor manually. The fragment of code in bold in Figure 6 shows a simple example of $\mathcal{U}$: When a new window is created, its security-label is initialized with the security-label of the client that created it.

   It is worth noting for this example that a pointer to the window is created only after memory has been allocated for it (in the CreateWindow function of the X server). Thus we place the call to 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, we 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 earlier, we assume that the server runs on a machine with a security-enhanced operating system. We use operating system support to bootstrap security-labels based upon how clients connect to the server (as has been done by others [35]). 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. Enforcing authorization policies on X clients using a retrofitted X server

We demonstrate how an X server retrofitted using Aid and Arm 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 rely on the ability of an X client to set properties of windows belonging to other X clients, for *e.g.,* by changing their background or content [25].

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 setting window properties 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 [7]). 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". Existing security mechanisms for the X server (namely, the X security extension [38]) 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. Aid identifies the fingerprints of Window_Chselection as calls to two functions, ProcSetSelectionOwner and ProcConvertSelection in the X server. It turns out that the former is responsible for the cut operation, and the latter for the paste operation. Calls to 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).

Performance of the retrofitted X server. We measured the runtime overhead imposed by instrumentation by running a retrofitted X server and a vanilla X server on 25 x11perf [40] 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. Limitations

The techniques presented in this paper have limitations, some fundamental, and some artifacts of our current implementation.

First, because Aid uses analysis of runtime traces, it is ideally suited for cases where a security-sensitive operation has a unique, or a small number of fingerprints. While we have observed that this is typically the case in practice (specifically, with the X server, and in the context of a previous study, with the Linux kernel [20]), Aid could potentially miss fingerprints in paths not exercised by any of the runtime traces that it analyzes. Further research on code-coverage metrics and static fingerprint-finding algorithms is needed to address this shortcoming.

Second, our techniques are incapable of analyzing obfuscated code. While it may be possible to identify individual code-patterns in obfuscated code, it will be harder to identify fingerprints with multiple code-patterns (*e.g.,* to identify that all these patterns appear together in a function). In addition, identifying subjects and objects, and modifying code to insert instrumentation becomes harder with obfuscated code.

Third, our infrastructure is currently built to analyze C source code, and we cannot analyze binary executables. However, this is not a fundamental limitation. Analyzing executables requires two key enhancements:

- The ability to instrument executables, both for fingerprint-finding (*e.g.,* to trace reads and writes to key data structures, as discussed in Section 3.1), and for adding reference monitor calls. Both these objectives can be achieved using static binary rewriters, or dynamic rewriters, such as Dyninst [12].
- The ability to express code-patterns in terms of abstract syntax trees of executables. Currently, code-patterns are expressed in terms of abstract syntax trees at the source code level (see Figure 3), thus constraining our analysis to work with source code.

7. Related work

Techniques for authorization policy enforcement. Reference monitors [2] have been used as the standard vehicle for authorization policy enforcement. Historically, policy enforcement has been performed by the operating system. Linux, for example, provides mechanisms to enforce discretionary access control policies. Recent work on SELinux [28] aims to augment Linux with mechanisms to enforce mandatory access control policies. SELinux is currently architected using the LSM framework [39], which adds a reference monitor as a loadable kernel module. This kernel module encapsulates a policy to be enforced, and presents an *authorization hook* interface. These hooks are placed so as to mediate security-sensitive locations within the kernel. In the context of LSM, these authorization hooks were placed manually using an informal process. Unfortunately, this process resulted in vulnerabilities in hook placement [22, 43]. Hook placement was found to violate complete mediation [32], and the hook interface left room for TOCTTOU exploits [9, 43]. This example shows the need for automated techniques to retrofit legacy code.

In prior work, we evaluated the use of static analysis techniques to automate the placement of authorization hooks in LSM [20]. In particular, given an implementation of the hook interface, and a non-hook-placed version of Linux, we used static analysis to determine the set of hooks to protect security-sensitive kernel locations. However, in that work, our static analysis algorithm relied on manually-written fingerprints (called *idioms* in [20]) to identify security-sensitive kernel locations, and the hook placement depended on the accuracy of these idioms. Aid addresses this shortcoming by providing tool-support to write fingerprints.

Java's security mechanism [21] 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.

While SELinux was obtained by retrofitting the Linux kernel, there have also been efforts to proactively construct secure operating systems. For example, Asbestos [13] incorporates several mechanisms to isolate user data and contain the effects of exploits. It enforces security policies using security-labels, as in SELinux.

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 toolkit) [17], Naccio [18], and Polymer [6] are three such frameworks, which have been used to enforce several policies on legacy code. 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 Aid to recover them.

Aspect-oriented programming. An *aspect* is defined to be a concern, such as security or error-handling, that crosscuts a program [3]. In *aspect-oriented programming (AOP)* [24] languages, (*e.g.,* AspectJ [5], AspectC++ [4]) these concerns are developed independently, as *advice*. An *aspect-weaver* merges advice with the program at certain *join-points*, which are specified to the weaver using *pointcuts*. Pointcuts are patterns that serve to succinctly identify a set of join-points—the weaver matches these patterns with the program to identify join-points.

The techniques developed in this paper bear close resemblance to the aspect-oriented programming paradigm. In particular, each combination of code-patterns (*i.e.,* a fingerprint) written in the language shown in Figure 3 identifies several locations in server code where reference monitor calls must be inserted. Thus, a fingerprint is a pointcut that identifies security-sensitive locations, which are join-points. Arm, which inserts reference monitor calls, is a compile-time aspect-weaver, while the code of the reference monitor and the authorization policy serve as the advice. A key issue in AOP is how to identify join-points—in our context, this is the problem of identifying security-sensitive locations. As we have discussed, Aid assists with this task.

Root-cause analysis. Root-cause analysis techniques, developed primarily for debugging, typically use "good" and "bad" traces to localize the root-cause of a bug [11, 27, 42]. Aid is similar to these techniques because it classifies program traces, and uses this classification to find fingerprints of security-sensitive operations. The primary difference between these techniques and Aid is that Aid *uses a much richer set of labels*, namely, an arbitrary set of security-sensitive operations, rather than just "good" or "bad". Another approach for trace analysis (primarily for debugging) is dynamic slicing [1, 26, 44]. Dynamic slicers use data-flow analysis to work backwards from the effect of a vulnerability, such as a program crash, to the cause of the vulnerability. An interesting avenue for future research will be to adapt Aid to use dynamic slicing to work backwards from the effect of a security-sensitive operation (a tangible side-effect) to the fingerprint of the operation.

Security of window systems. The X server was historically developed to promote cooperation between X clients, and security (*e.g.,* isolation) of X clients was not built into the design of the server. The X protocol, which X clients use to communicate with the X server, has well-documented security flaws, too [37]. There is a rich body of work to rectify this situation, and identify security requirements for, and create secure versions of the X server. Most of this work was carried out in the context of the Compartmented Mode Workstation [8, 15, 31], and the Trusted X projects [14, 16], which built prototype windowing systems to meet the Trusted Computer System Evaluation Criteria. While these efforts focus on retrofitting the X server, there is also work on building X server-like window systems, with security proactively designed into the system [19, 34].

The X security extension [38] extends the X server by enabling it to enforce authorization policies. It does so by placing reference monitor calls at appropriate locations in the X server, as discussed in this paper. To the best of our knowledge, these calls were placed manually, and thus the techniques presented in this paper could have assisted in that effort. However, the X security extension is quite limited in the policies that it can enforce. It statically partitions clients into Trusted, and Untrusted, and only enforces policies on interactions between these two classes of clients. Thus for example, if three clients, with security-labels Top-secret, Confidential, and Unclassified connect to the X server simultaneously, the X security extension will group two of them into the same category, and will not enforce policies on clients in the same category.

8. Summary and future work

We have shown that program analysis can assist with the process of retrofitting legacy code for authorization policy enforcement. Using our prototype tools, Aid and Arm, we retrofitted the X server to enforce authorization policies on its X clients.

In addition to using these tools to retrofit more servers, we plan to explore several avenues of research to improve the basic techniques presented in this paper. First, identifying security-sensitive operations is a manual process, and we plan to develop tool-support to assist with this task. Second, because Aid uses runtime analysis, it can potentially miss fingerprints of security-sensitive operations, as discussed earlier. We plan to address this by exploring code-coverage and static fingerprint-finding algorithms. Last, by investigating how reference monitor calls can be added to running executables (*e.g.,* using the Dyninst API [12]), we plan to extend the techniques presented in this paper to cases where source code is unavailable.

Acknowledgments

We thank Mihai Christodorescu for his help in shaping the message of this paper. We benefited from insightful comments by Jon Giffin, Boniface Hicks, Louis Kruger, Shai Rubin, Hao Wang and the anonymous reviewers. We also thank members of the Secure Systems Department at IBM Research, Hawthorne, where this work was initiated. This work is supported in part by ONR contracts N00014-01-1-0708 and N00014-01-1-0796.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM Conference Programming Language Design and Implementation*, June 1990.

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

[3] Aspect-oriented software development. http://aosd.net.

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

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

[6] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *ACM Conference on Programming Language Design and Implementation*, June 2005.

[7] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and MULTICS interpretation. Technical Report ESD-TR-75-306, Deputy for Command and Management Systems, L. G. Hanscom Field, Bedford, MA, March 1976.

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

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

[10] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *IEEE Symposium on Security and Privacy*, May 1989.

[11] H. Cleve and A. Zeller. Locating causes of program failures. In *International Conference on Software Engineering*, May 2005.

[12] Dyninst API. http://www.dyninst.org.

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

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

[15] J. Epstein and J. Picciotto. Trusting X: Issues in building trusted X window systems -or- What's not trusted about X? In *Annual National Computer Security Conference*, October 1991.

[16] J. J. Epstein. A prototype for trusted X labeling policies. In *Annual Computer Security Applications Conference*, December 1990.

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

[18] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, May 1999.

[19] N. Feske and C. Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *Annual Computer Security Applications Conference*, December 2005.

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

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

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

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

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

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

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

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

[28] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *FREENIX track: USENIX Annual Technical Conference*, June 2001.

[29] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, April 2002. Available at: http://manju.cs.berkeley.edu/cil.

[30] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages*, January 2002.

[31] J. Picciotto. Towards trusted cut and paste in the X window system. In *Annual Computer Security Applications Conference*, December 1991.

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

[33] Summary of informal SELinux meeting, May 2004. http://www.selinux-symposium.org/meeting.php.

[34] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *USENIX Security Symposium*, August 2004.

[35] S. Smalley, June 2005. Personal Communication.

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

[37] D. Wiggins. 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.

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

[39] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symposium*, August 2002.

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

[41] The X11 Server, version X11R6.8 (X.Org Foundation).

[42] A. Zeller. Isolating cause-effect chains from computer programs. In *ACM Symposium on Foundations of Software Engineering*, November 2002.

[43] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *USENIX Security Symposium*, August 2002.

[44] X. Zhang and R. Gupta. Precise dynamic slicing algorithms. In *International Conference on Software Engineering*, May 2003.