Leveraging "Choice" to Automate Authorization Hook Placement

Divya Muthukumaran
Pennsylvania State University
muthukum@cse.psu.edu

Trent Jaeger
Pennsylvania State University
tjaeger@cse.psu.edu

Vinod Ganapathy
Rutgers University
vinodg@cs.rutgers.edu

ABSTRACT

When servers manage resources on behalf of multiple, mutually-distrusting clients, they must mediate access to those resources to ensure that each client request complies with an authorization policy. This goal is typically achieved by placing authorization hooks at appropriate locations in server code. The goal of authorization hook placement is to completely mediate all security-sensitive operations on shared resources.

To date, authorization hook placement in code bases, such as the X server and postgresql, has largely been a manual procedure, driven by informal analysis of server code and discussions on developer forums. Often, there is a lack of consensus about basic concepts, such as what constitutes a security-sensitive operation.

In this paper, we propose an automated hook placement approach that is motivated by a novel observation — that the **deliberate choices made by clients** for objects from server collections and for processing those objects must all be authorized. We have built a tool that uses this observation to statically analyze the server source. Using real-world examples (the X server and postgresql), we show that the hooks placed by our method are just as effective as hooks that were manually placed over the course of years while greatly reducing the burden on programmers.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*access controls*

General Terms

Security

Keywords

Authorization hooks, Static taint analysis

1. INTRODUCTION

In this paper, we consider the problem of retrofitting legacy software with mechanisms for authorization policy enforcement. This is an important problem for operating systems, middleware and server applications (jointly, *servers*), which manage resources for and provide services to multiple, mutually-distrusting clients. Such servers must ensure that when a client requests to perform a security-sensitive operation on an object, the operation is properly authorized. This goal is typically achieved by placing calls (termed *authorization hooks*) to a reference monitor [2] at suitable locations in the code of the server. At runtime, the invocation of a hook results in an authorization query that specifies the subject (client), object, and operation. The placement of authorization hooks must provide *complete mediation* of security-sensitive operations performed by the server. If this property is violated, clients may be able to access objects even if they are not authorized to do so.

In the past decade, several efforts have attempted to place authorization hooks in a variety of servers. For example, discretionary access control mechanisms [14, 23] deployed in the Linux kernel were found to be insufficient to protect the security of hosts in a networked world [21]. The Linux Security Modules (LSM) framework [41] remedies this shortcoming by placing authorization hooks to enforce more powerful security policies. Even user-space servers can benefit from similar protection. For example, the X server manages windows and other objects for multiple clients. Accesses to such objects must be mediated, void of which several attacks are possible [8, 18, 39]. The X server has also therefore been retrofitted with LSM-style authorization hooks [18]. Similar efforts now abound for other server applications (e.g., Apache, Postgres, Dbus, Gconf) [6, 19, 22, 3], operating systems [38, 35, 10], and virtual machine monitors [27, 5]

Unfortunately, these efforts have been beset with problems. This is because the identification of security-sensitive operations and the placement of hooks is a manual procedure, largely driven by informal discussions on mailing lists in the developer community. There is no consensus on a formal definition of what constitutes a security-sensitive operation, leading to complaints about the difficulty of the placement task [1, 29]. Not surprisingly, this *ad hoc* process has resulted in security holes [7, 42], in some cases many years after hooks were deployed [34]. The discussion of which hooks to deploy can often last years, e.g., the original hook placement for the X server was proposed in 2003 [18], deployed in 2007 [36], and subsequent revisions have added additional hooks. What we therefore need is a principled way to identify security-sensitive operations and their occurrence in code, so that legacy servers can be automatically retrofitted with authorization hooks.

Prior work to address this problem has focused both on verification of authorization hook placement to ensure complete mediation [15, 34, 30] and on mining security-sensitive operations in legacy code [12, 13]. The work on mining security-sensitive operations is the most closely related to this paper, and uses static [13] and dynamic program analysis [12] to identify possible hook place-

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

ment locations. However, these mining techniques rely on domain-specific knowledge (e.g., a specification of the data types that denote security-sensitive objects [13]), providing of which still requires significant manual effort and a detailed understanding of the server's code base.

The main contribution of this paper is a novel automated method for placing authorization hooks in server code that significantly reduces the burden on developers. The technique produces hook placements that provide complete mediation, while optimizing away any redundant hook placements automatically, reducing the number of statements that must be considered by approximately 90% or more. To develop the technique, we rely on a key observation that we gleaned by studying server code. In a server, clients make requests, which identifies the objects manipulated and the security-sensitive operations performed on them. *We observe that the* **deliberate choices made by clients** *for objects from server collections and for processing those objects must all be authorized.* These choices determine the security-sensitive operations implemented by the server, where all must be mediated to prevent unauthorized client accesses.

Based upon this observation, we design a static program analysis that tracks user choice to identify both security-sensitive objects and the operations that the server performs on them. Our analysis only requires a specification of the statements from which client input may be obtained (e.g., socket reads), and a language-specific definition of object containers (e.g., arrays, lists), to generate a complete authorization hook placement. It uses context-sensitive, flow-insensitive data flow analysis to track how client input influences the selection of objects from containers: these are marked security-sensitive objects. The analysis also tracks how control flow decisions in code influence how the objects are manipulated: these manipulations are security-sensitive operations. The output of this analysis is a set of program locations where mediation is necessary. However, placing hooks at all these locations may be sub-optimal, both in terms of the number of hooks placed (e.g., a large number of hooks complicates code maintenance and understanding) and the number of authorization queries generated at runtime. We therefore use the control structure of the program to further optimize the placement of authorization hooks.

We have implemented a prototype tool that applies this method to C programs using analyses built on the CIL tool chain [24]. We have evaluated the tool on programs that have manual mediation for comparison, such as the X server and *postgresql*. We also evaluated the tool in terms of the reduction of programmer effort it entails and the accuracy of identification of objects and security sensitive operations in programs.

To summarize, our main contributions are:

- An approach to identifying security-sensitive objects and operations by leveraging a novel observation — that a deliberate choice by the client of an object from a collection managed by the server signals the need for mediation.
- The design and implementation of a static analysis tool that leverages the above observation to automate authorization hook placement in legacy server applications. This tool also identifies optimization opportunities, i.e., cases where hooks can be hoisted, thereby reducing the number of hooks in the source code, and by eliding redundant hook placements that would otherwise result in extra authorization queries at runtime.
- Evaluation with four significant server applications, namely, the X server, postgresql, PennMush, and memcached, demonstrating that our approach can significantly reduce the manual burden on developers in placing authorization hooks. In case of two of these servers, the X server and postgresql, there have been efforts spanning multiple years to place authorization hooks. We show that our approach can automatically infer hook placements for these servers are moreover the hooks placed are comparable to the manual hooks placed.

2. PROBLEM DEFINITION

In this section, we define the authorization hook placement problem. Authorization is the process of permitting a *subject* (e.g., user) to perform an *operation* (e.g., read or write) on an *object* (e.g., program variable or channel), which is necessary to control which subjects may perform security-sensitive operations. An authorization hook is a program statement that submits a query for the request *(subject, object, operation)* to an authorization mechanism, which evaluates whether this request is permitted (e.g., by a policy). The program statements guarded by the authorization hook may only be executed following an authorized request. Otherwise, the authorization hook will cause some remedial action to take place that is customized to the program.

The Anderson Report [2] identifies the requirements for developing a secure reference validation mechanism, which includes both the authorization hooks and authorization mechanism.

- The reference validation mechanism must always be invoked (*complete mediation*)
- The reference validation mechanism must be tamperproof (*tamperproof*).
- The reference validation mechanism must be small enough to be subject to analysis and tests (*verifiable*).

The placement of authorization hooks mainly addresses complete mediation, which requires that all security-sensitive operations be mediated by an authorization hook to ensure that the security policy permits all such operations. However, we find that the placement of authorization hooks also impacts tamperproofing, by authorizing requests that may affect the authorization process (e.g., change the policy), and verifiability, by determining where authorization queries are deemed necessary. Verifiability, in particular, may be aided by an automated method for authorization hook placement, as it often takes years for developers to arrive at a consensus regarding an acceptable placement.

The main challenge in producing an authorization hook placement is identifying what a security-sensitive operation is. To date there is no formal definition of this concept nor even a decent working definition, as discussed in the Introduction. As an operation must be applied to an object, the definition for security-sensitive operations must identify both security-sensitive objects and operations on these objects that may impact the program's security enforcement. In prior approaches, these definitions are program-specific, require significant manual input, and/or lack information necessary to choose placements unambiguously.

First, identifying security-sensitive objects is difficult because any program variable could be a security-sensitive. At present, there is no principled approach to determine which are security-sensitive, so the prior methods proposed to assist in authorization hook placement [42, 7, 11, 12, 13, 34, 30, 26, 33] expect programmers to specify the data types whose variables require authorization, which requires extensive domain knowledge. The identification of these data structures is not a trivial task and often takes multiple iterations to get right. For example, the X server version 1.4 did not have hooks for accessing certain classes of objects (such as "selection" objects), which were added in version 1.5. The set of security sensitive-objects for *postgresql* is still under discussion by the community.

Second, identifying security-sensitive operations upon these objects is difficult because any program statement that accesses a security-sensitive object could be security-sensitive. Traditionally, researchers use the structure member accesses on the security-sensitive user-defined types as security-sensitive operations. However, there are many such operations (9133 in X server), and clearly there are many fewer authorization hooks. In some cases, researchers identify specific types of operations as security-sensitive, such as interfaces to database functionality in PHP programs [30] or method calls in object-oriented languages [9]. However, such interfaces may not be present in the program or may only cover a subset of the security-sensitive operations. Prior methods have identified security-sensitive operations at the level of APIs [13], but this may be too coarse a granularity. For example, hook changes between version 1.4 and 1.5 of the X server mediate finer-grained operations to reduce the privilege given to some subjects.

Finally, once the security-sensitive operations are identified, authorization hooks must be placed. However, finding the "best" placement that mediates all security-sensitive operations is more difficult than it may appear. Complete mediation may be achieved by many placements, and at present, there are no principles for what constitutes an *ideal* authorization hook placement beyond complete mediation. We identify two competing goals. First, we need to place the minimal number of hooks that provide complete mediation. The minimal number of hooks also aids in verifiability while keeping the code clean which is a point of discussion in the SE-PostgreSQL work [28]. The second goal is to enforce the desired authorization policy. However, the minimal placement may not always be capable of enforcing the desired authorization policy. Suppose authorization hooks are placed at the point where a security-sensitive object is retrieved. This would necessitate the authorization of a subject for all possible operations performed along all code paths in the program that follow this retrieval, even if each code path performs an entirely different operation from the others. This in turn violates the principle of least privilege since the placement of the hook would require each subject to have all possible permissions for an object irrespective of the requested operation. We therefore need to place hooks at a granularity that distinguishes the distinct operations that may follow object retrieval.

As a result, we state that the *authorization hook placement problem is to find a non-redundant set of authorization hooks necessary to mediate all the security-sensitive operations*. This definition means that solutions depend heavily on this imprecise notion of security-sensitive operation. Ganapathy *et al.* explored techniques to group statement-level operations into sets that represent security-sensitive operations using dynamic analysis and concept analysis from manually highlighted interfaces [12, 13]. This definition also means that solutions will generally be finer-grained than manual placements, since manual placements aggregate hooks based on known or anticipated patterns in permission assignments. It is not possible for an automated placement method based on source code alone to account for authorization policy necessary to minimize hook placements. In this paper, we propose a novel method for identifying security-sensitive objects and operations that more closely approximates the programmers' intuition about hook placement. Moreover, in lieu of such an automated hook placement being finer-grained than manual placements, we find that it is possible to extract the relationship between the two placements to show programmers the security/policy implications of any given manual hook placement in terms of the finer-grained hooks that were optimized away.

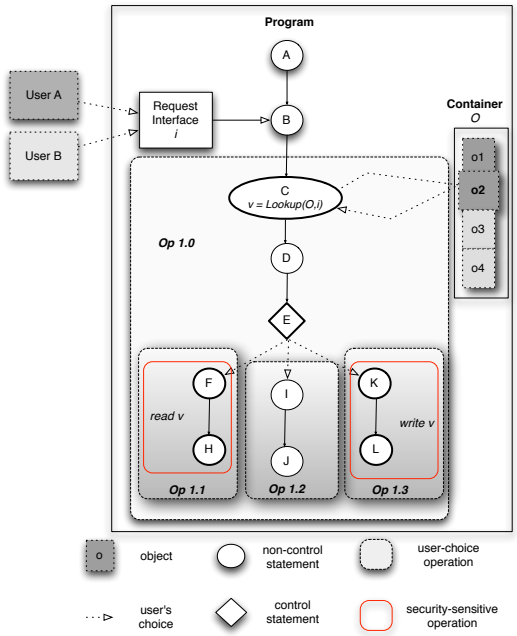

Figure 1: The authorization hook placement problem

3. APPROACH OVERVIEW

Figure 1 shows the insights used to motivate our solution approach. Authorization is needed when only a subset of subjects should be allowed to access particular program objects or perform particular accesses on those objects. Figure 1 shows that objects $o1$ and $o2$ are accessible to a subject User A, but $o3$ and $o4$ are not. Further, the subject may not be allowed to perform all operations on her accessible objects. For example, User A may only be allowed to perform a read operation on object $o1$, while she can both read and write object $o2$. The choice of authorization hook placement must ensure that the program can only perform an operation after it is mediated for that operation, while ensuring that a subject is authorized only for the operation she has requested to perform. Thus, the statements F and H should only be mediated for read operations, whereas the statements K and L should only be mediated for write operations.The statements I and J do not require mediation as they perform no security-sensitive operations.

The program's behaviors on behalf of subjects are determined by the subject's *user requests*. We find that by tracking user requests we can identify the set of objects that require mediation because such inputs guide the selection of objects for processing. Also, by tracking user requests, we can identify operations the program performs because such inputs choose the program statements that manipulate objects. We detail the method to track user request input in Section 4.1.

Programs that manage objects on behalf of multiple users typically store them in *containers*. When a subject makes a request, the program may use the request input to choose the objects to retrieve from some containers, potentially resulting in access to a data used by another subject. As the retrieved values are assigned to program variables, these variables represent the program's security-sensitive objects. In Figure 1, variable v in statement C is security-sensitive because the user request input i is used to retrieve an object from the container. By tracking the dataflow from user requests to the selection of objects in containers, we can identify the variables that hold these objects in the program. We detail the method to identify security-sensitive objects in Section 4.2.

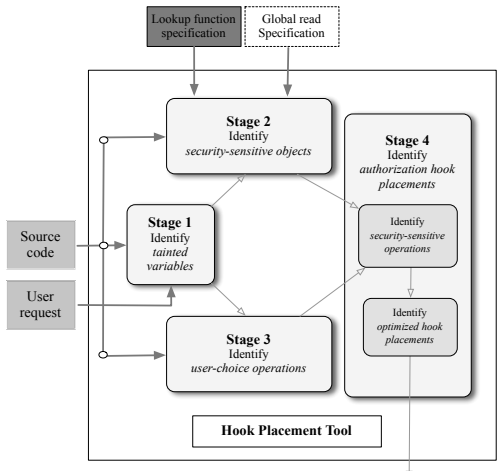

Figure 2: Shows the sequence of stages in automated authorization hook placement. The light shaded boxes shows the required programmer specification for every program, the dark shaded box shows language-specific specification, and the dashed box shows optional specification.

The program executes statements chosen by its *control statements*. If a user request affects the values of the variables used in a control statement's predicate, then the subject can choose the program statements that may access security-sensitive objects. We call the sets of program statements that may be chosen by subjects *user-choice operations*. As shown in Figure 1, statement E is a control statement. It is shown that three user-choice operations result if E's predicate is dependent on the user request. In addition, the choice of an object from a container is also a user-choice operation. Only the user-choice operations that contain accesses to variables that hold security-sensitive objects represent security-sensitive operations, which are just operations `read` v and `write` v. These are the operations that require mediation via authorization hooks. We detail the method to identify user-choice operations in Section 4.3.

Using a naive placement of an authorization hook per security-sensitive operation may lead to sub-optimal hook placement. For instance, if all three user-choice operations at E perform the same security-sensitive operation, then we could place a single hook at E to the same effect as placing a separate hook at each branch. Also, if the same security-sensitive operation is performed twice as part of a single request, we only need to authorize it only once. Our solution to the authorization hook placement problem optimizes hook placement by removing redundant mediation in two ways. First, we remove hooks from sibling operations (i.e., user-choice operations that result from the same control statement) if they mediate the same operations, which we call hoisting common operations. Second, we remove any mediation that is already performed by existing hooks that dominate the operation, which we call removing redundant mediation. We detail these optimizations in Section 4.4.

4. DESIGN

Figure 2 shows the design steps in our approach. The primary input is the source code of the program along with a manual specification identifying the program variables where user request enters the program. We discuss the other inputs in the subsequent sections.

In Stage 1, we identify the set of program variables tainted by the user request inputs, called *tainted variables*, using an interprocedural static taint analysis [4]. The results of this analysis are used in the subsequent stages to identify objects and operations chosen by the user.

In Stage 2, we identify the set of program variables that represent *security-sensitive objects*. We use the tainted variables identified in Stage 1 to find variables whose assigned values are retrieved from container data structures using tainted variables.

In Stage 3, we identify the set of *user-choice operations*. We use the tainted variables from Stage 1 to identify control statements whose predicates include tainted variables. A user-choice operation is created for each conditional branch that is dependent on the control statement. Each user-choice operation is represented by a subgraph of the program's control dependence graph [32] for that conditional branch.

In Stage 4, we combine the security-sensitive objects identified in Stage 2 and the user-choice operations identified in Stage 3 to identify *security-sensitive operations*. A user-choice operation is security-sensitive if its statements access variables that represent security-sensitive objects. The actual accesses that take place in a security-sensitive operation determine the authorization requirements that must be approved using the authorization hook. We then propose two techniques to remove unnecessary and redundant authorization hook placements.

4.1 Stage 1: Tracking User Requests

The goal of this stage is to identify the set of all variables in the program that are dependent on the user request variables; we call this the set of *tainted variables*. To identify which program variables are dependent on the user request input, we first need to identify the user request variables in the program. We obtain the user request variables from the programmer as a manual specification of the variables that signify where user requests are read by the program. For example, in X server the READREQUESTFROMCLIENT function performs a socket read to obtain the request from the client and initializes a variable representing the request. We provide a specification that this variable represents the user request. We discuss the identification of user requests in Section 5.

We compute the set of tainted variables as follows. Let P be the program we wish to analyze, $S(P)$ be the set of all statements of P, and $V(P)$ be the set of all variables in P. We want to identify the set $\mathbf{V_T(P)}$ of all *tainted variables* in the program. Let $V_I(P) \subseteq V(P)$ be the set of variables that represent the manual specification of the user request variables. Let $\mathcal{D}(v, v')$ be a relation which is true if variable v is data-flow dependent on variable v'.

DEFINITION 1. *The set of tainted variables* $\mathbf{V_T(P)}$ *is the transitive closure of the relation* $\mathcal{D}$ *from the user request variables* $V_I(P)$.

Taint analysis is used extensively in program analysis, frequently to detect security vulnerabilities. It can identify the set of program variables that are data-dependent on any input variables. Static taint analyses can approximate this information although factors such as aliasing and polymorphic types lead to imprecision. Nevertheless, they have been successfully used in various projects to identify security vulnerabilities [20, 16, 37].

Chang *et al.* [4] have used static interprocedural context sensitive taint tracking to compute the set of program values data-dependent on network input. We adopt the same approach, but the tainted-ness that we track at each variable indicates whether it is data-dependent on user request. This technique uses procedure summarization to avoid the exponential blow-up that can occur during context-sensitive analyses. A procedure summary is a succinct representation of some procedure behavior of interest as a function parameterized by the input variables of the procedure. Through the procedure summary we want to capture if the procedure enables

the taint to be propagated at its output variables given information about whether its input variables are tainted. Therefore, we the summary of a procedure is the taintedness of the output variables of the procedure (those that escape the procedure scope such as returns, pointers, globals, etc.) represented as a function of the taintedness of its input variables.

The taint-propagation involves two passes through the call graph. The bottom up summarization phase visits each node of the call-graph in the reverse topological sort order and creates procedure summaries. Recursive procedures are treated context-insensitively. The top-down propagation phase visits the call graph in topological order, computing the actual taint at each variable, using procedure summaries to resolve procedure calls. The result of this analysis is the set of tainted variables $V_T(P)$ which can be queried during the next two stages to check if the variables are dependent on user request input.

This stage yielded 2795 variables in $V_T(P)$ for X server which is 38% of all variables in the portion of the X server program that we analyzed. Section 6 shows the size of $V_T(P)$ for the different programs analyzed.

4.2 Stage 2: Finding Sensitive Objects

The goal of this stage is to identify the set of program variables that reference security-sensitive objects (i.e., the set of objects that need authorization). As we described in Section 3, security-sensitive objects are stored in containers. A *container* is an instance of a container data type or variable that holds multiple instances of the same type. Examples of containers include arrays, lists, queues, stacks, hash tables, etc. We claim that whenever the program stores a collection of objects in a container and the user has the ability to choose specific object(s) from this container, the chosen object becomes security-sensitive. Without additional manual input, it is impossible to determine whether all objects in a container are uniformly accessible to all users. We therefore make the conservative assumption that whenever user request input determines the choice of an object from a container, this object needs authorization by default. The variables to which such objects are assigned after retrieval thus represent the *security-sensitive objects* in the program.

The user can request an object from the container by specifying an identifier for the object. For example, X server has a global array (called `clientTable`) to store the set of resources (windows, cursors, fonts, etc.) belonging to all subjects. The user requests supply an identifier which is used to index into this array to retrieve the corresponding object.

We find containers may be arranged hierarchically, such that an object retrieved from a container may also have a field with another container. For example, in the following Listing 1 for the function DIXLOOKUPPROPERTY, the `Window` object `win` (which was previously retrieved from the global container `ClientTable`) has a field `win->opt->userProps` which is a list that stores properties associated with that window. The property `prop` retrieved from this container is needs to be authorized since user request provides the input parameter `pName` that specifies the property to be looked up.

In the identification of objects retrieved from containers, we focus on instances of programmer-defined data structures. Programmers typically define custom data structures to manage resources that are uniquely tied to the functionality of the application. For example, the X server has data structures for windows, devices, screens, fonts, cursors, etc. Past efforts [12, 13, 34] have also focused on custom data structures.

Since containers are language-specific abstractions, we require a language-specific way of identifying the retrieval of objects from containers. We use the term *lookup function* to refer to any routine that retrieves objects from containers. Our approach depends on a language-dependent specification of lookup functions as shown in Figure 2. Since our analysis deals with C programs and the C language does not have standard lookup functions, we provide specifications in the form of code patterns. For example, in the above code snippet, the DIXLOOKUPPROPERTY function uses a standard code-level idiom in the C language, `next` pointer, to iterate through the list, comparing each item in the list with the identifier to determine if it satisfies the criteria. This code pattern is an example of a lookup function specification that the programmer needs to specify. Many object-oriented programming languages provide container classes which export well-defined methods to create a new container, insert, delete and provide access to objects in the container. In such cases, it is straightforward to identify lookup functions. We discuss the lookup function identification in more detail in Section 5.

Listing 1 Procedure to look up a specific property of a specific window in X server.

```
1  /*** property.c ***/
2  int dixLookupProperty(PropertyPtr *p,
3      Window * win, Atom pName,Client * c)
4  {
5    PropertyPtr prop;
6    for (prop=win->opt->userProps;
7            prop; prop = prop->next)
8        {
9         if (prop->name == pName)
10         break;
11       }
12    *p = prop;
13 }
```

Given such lookup functions, we can now formally define the set of all security-sensitive variables $\mathbf{V_S(P)}$ in the program.

DEFINITION 2. *A variable $v \in V_S(P)$ if any following are true: a) If it is assigned a value from a container via a lookup function using a variable $v' \in V_T(P)$. b) If $\mathcal{D}(v', v)$ is true for some $v' \in V_S(P)$. c) If it is a global variable and in the set $V_T(P)$.*

First, any variable retrieved from a container using a tainted variable is security-sensitive. Second, any variable data-dependent on security-sensitive variables is also security-sensitive. Finally, any globals that can potentially be modified based on user request are also security-sensitive. This prevents trivial data flows between subjects using globals that may be modified based on user requests. If a global variable contains secret data that must be authorized before being read by a subject, these variables must be identified manually, as shown in Figure 2. We have found these variables to be rare in the programs that we have examined.

4.3 Stage 3: Finding User-Choice Operations

In this stage, our goal is to identify the set of operations that the user can choose to execute by modifying their user request input; we call these the *user-choice operations*. Control statements such as `if`, `switch` and function pointers are choice points in the program, where different program statements may be executed based on the value of the predicate evaluated in these control statements or the choice of function pointer values. If a tainted variable is used in a control statement's predicate or as a function pointer, then the subject can choose among different program functionality based on the values in the user request. Thus, the user request input provides the subject with a means to choose among sets of program statements to execute. These sets of program statements are the user-choice operations.

Listing 2 The code snippet showing the procedure to change a specific property of a window of X server.

```
1  // ''stuff'' stores a formatted version
2  //  of the client request
3  int ChangeWindowProperty(ClientPtr *c,
4      WindowPtr * w, int mode)
5  {
6    WindowPtr * win;
7    PropertyPtr * pProp;
8    err = LookupWin(&win,stuff->window, c);
9    rc =  LookupProperty(&pProp, win,
10            stuff->property, c);
11   if (rc == BadMatch)
12     {/* Op 1*/
13      pProp->name = property;
14      pProp->format = format;
15      pProp->data = data;
16      pProp->size = len;
17     }
18   else
19     { /* Op 2 */
20       if (mode == REPLACE)
21         { /* Op 2.1 */
22           pProp->data = data;
23           pProp->size = len;
24           pProp->format = format;
25         }
26       else if (mode == APPEND)
27         {/* Op 2.2 */
28           pProp->data = data;
29           pProp->size += len;
30         }
31     }
32 }
```

Consider the code snippet in Listing 2 from X server which shows the function CHANGEWINDOWPROPERTY. Here the user request inputs are `stuff`, `rc` and `mode` and statements predicated on these variables lead to user choice. First, `stuff` determines the choice of window and property being extracted. The lookup of an object from a container itself becomes the start of an operation that includes all statements in the program that can be executed following the lookup of the object. Next, the three control statements that are predicated on `rc` and `mode` also cause user-choice operations. The first, `if (rc == BadMatch)` leads to two branches, `Op 1` and `Op 2` that represent two conceptually different operations, namely, adding a new property and changing an existing one. Furthermore, the latter operation has additional control statements `if (mode == REPLACE)` and `if (mode == APPEND)` through which the subject has the option of choosing between replacing an existing property (`Op 2.1`) and appending toit (`Op 2.2`).

We use a standard program representation called the Control Dependence Graph [32] (CDG) to characterize operations. A CDG of a program $CDG(P) = (S(P), E)$ consists of a set of program statements $S(P)$ and the edges are represent the control dependence relationship between two statements. The control dependence property is typically defined in terms of control dominance relationships that exist between nodes in the Control Flow Graph (CFG) of the program. A statement Y is control-dependent on X if Y post-dominates a successor of X in the CFG but does not post dominate all successors of X. We make a minor adjustment to the traditional CDG representation by adding a dummy node for each choice (e.g., branch or possible function pointer value) of a control statement to group the program statements associated with each

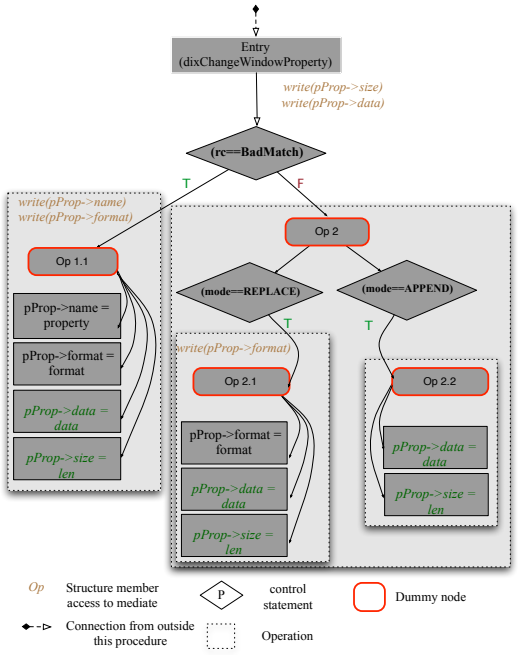

Figure 3: Shows the CDG with the user-choice operations for the DIXCHANGEWINDOWPROPERTY function

possible choice. We now define operations in terms of the CDG of the program.

DEFINITION 3. *An* operation *is a subgraph of the CDG rooted at dummy node. If a dummy node's control statement is predicated on a variable in $V_T(P)$, then the operation rooted at the dummy node is a* user-choice operation.

Figure 3 shows the CDG constructed for the procedure DIXCHANGEWINDOWPROPERTY from listing 2. It also shows the four user-choice operations `Op 1.1`, `Op 2`, `Op2.1` and `Op 2.2` identified for the procedure DIXCHANGEWINDOWPROPERTY in listing 2. The statements in the diamond-shaped figures are the control statements, whose children are dummy nodes representing the start nodes for operations.

For the X server code the tool returned a total of 4760 user-choice operations. The results for the other programs are shown in Section 6.

4.4 Stage 4: Placing Authorization Hooks

Our final stage is the placement of authorization hooks. In order to do this, we first need to determine which of the user-choice operations computed in Stage 3 are actually security-sensitive operations. We then need to find placements of authorization hooks to mediate these operations. Our placement mechanism ensures complete mediation of all security-sensitive operations while ensuring that a user will never be authorized of any operations other than the ones requested.

We now have the set of user-choice operations any of which can be requested by the user and the set of security-sensitive objects can of which can also be requested by the user. We now define security-sensitive operations as follows:

DEFINITION 4. *A operation is* security-sensitive *if it is a user-choice operation and it allows the user to access a variable in $V_S(P)$.*

A security-sensitive operation must be mediated by at least one authorization hook. In order to place only necessary authorization hooks, we need to determine the authorization requirements of each

security-sensitive operation. Such authorization requirements of an operation consist of the set of *security-sensitive accesses* performed by the operation's statements. Past efforts [13, 12, 34] have used structure member accesses of variables to represent the permissions required in programs. However, as accesses to variables directly may need to be protected, we generalize the definition slightly. In this work, each security-sensitive access consists of a variable that holds a security-sensitive object and the accesses made upon that variable (i.e., read or write).

We now have all the information we need to generate a placement of authorization hooks. A naive placement would simply associate an authorization hook with each security-sensitive operation. However, a naive placement would also be suboptimal. There are two problems. First, an authorization hook may be redundant if a dominating operation already authorizes all of the security-sensitive accesses performed in this operation. Second, an authorization hook may be hoisted if all the security-sensitive operations for the same control statement perform the same security-sensitive accesses. We must be careful to ensure that each authorization hook only authorizes security-sensitive accesses performed by the associated security-sensitive operation, which limits the amount of hoisting of authorization hook placement possible. Under this constraint, we discuss the two optimizations below. Each optimization relies heavily on the CDG of the program.

Hoisting common operations. If all security-sensitive operations associated with the same control statement perform the same security-sensitive accesses, then it implies that irrespective of the choice the user makes, the security implication is the same. So we can hoist the authorization hook to operations that dominate those operations. This will enable us to place a single hook instead of a hook at each branch. Figure 3 shows how the structure member accesses `write(pProp->data)` and `write(pProp->size)` have been hoisted above control statement `rc==BadMatch` presumably at the operation that contains a call to the function DIXCHANGEWINDOWPROPERTY. In practice though, we restrict the hoisting of the structure member accesses to the entry point of the procedure if the procedure has more than one caller.

Algorithm 1 shows how we perform this optimization. For a node s_i, let $AS[s_i]$ represent the set of all security-sensitive accesses occurring in all operations dominated by s_i. For a control statement predicated on the user request, $AS[s_i]$ is the intersection of security-sensitive accesses occurring in all its successors. For all other statements, it is the union of accesses on their successors. We perform this optimization by processing the CDG bottom-up in reverse topological sort order, computing $AS[s_i]$ at each node s_i.

Algorithm 1 Bottom Up Operation Accumulation

$top' = TopoSortRev(G_d(P))$
while $top' \neq \emptyset$ **do**
  $s_i = top'.pop()$
  if $isControl(s_i, V_T(P))$ **then**
    $AS[s_i] = \bigcap_j AS[s_j] | (s_i, s_j) \in CDG(P)$
  else
    $AS[s_i] = \bigcup_j AS[s_j] \cup acc(s_j) | (s_i, s_j) \in CDG(P)$
  end if
end while

Avoiding redundant mediation. Our second optimization is aimed at avoiding redundant authorizations. We take advantage of the fact that it is sufficient to perform a specific authorization check only once along any control flow path. For a particular path $s_1 \rightarrow s_2 \rightarrow^* s_k$ in the CDG, the set of authorizations at s_k is the union of all authorizations on that path. However, there may be multiple distinct paths that reach s_k in the program. For example, a function may be called by two callers, one which has authorization hooks and one which does not. Let $AP[s_i]$ store the set of all security-sensitive accesses authorized along all paths leading to s_i in the CDG. Let $AT[s_i]$ be the set of security-sensitive accesses to be authorized at a node s_i. Algorithm 2 shows how we compute $AT[s_i]$. For this step, we process the CDG in the topological sort order.

In Figure 3 we can see that the security-sensitive accesses `write(pProp->data)` and `write(pProp->size)` that have already been authorized first control before the first control statement and therefore do not have to be mediated again at any operation that this statement dominates.

Algorithm 2 Control Flow Dominance

$top = TopoSort(G_d(P))$
while $top \neq \emptyset$ **do**
  $s_i = top.pop()$
  $AP[s_i] = AS[s_i] \cup \bigcup_j AP[s_j], (s_j, s_i) \in CDG(P)$
  $AT[s_i] = AS[s_i] - \bigcup_j AP[s_j], (s_j, s_i) \in CDG(P)$
end while

The set of hook placement locations is now defined as

$$A(P) = \{s_i | AT[s_i] \neq \emptyset\}$$

For X server we found that the there were 1382 placement locations before optimization and 532 after optimization. The total number of hook placement locations for the other test programs is shown in Section 6.

4.5 Proving Complete Mediation

Proving complete mediation. We present an *oracle-based argument* that our technique ensures complete mediation of all security-sensitive operations. That is, we can reason about the correctness of the output of our approach assuming the correctness of certain oracles upon which it depends. Complete mediation stipulates that *every security-sensitive operation must be control flow dominated by an authorization hook*. This involves showing that (a) the identification security-sensitive operations is complete, i.e., we have identified all security-sensitive operations in the program, and b) the completeness of hook placement to authorize the security-sensitive operations.

First, let us consider the identification of security-sensitive operations. This task depends on the identification of *security-sensitive objects* and *user-choice operations*. The completeness of these steps depends on the output of static taint analysis and manual specification of user-request variables. Proving that static taint analysis is complete depends on knowing which variables can alias each other. Since alias analysis is intractable in a language like C, we cannot show that static taint tracking it is complete. Additionally, the identification of security-sensitive objects is also dependent on the complete specification of all container lookup functions. Therefore, the completeness of identification of security-sensitive operations is contingent on the completeness of two oracles: (1) static taint analysis, which although intractable, can be made conservative, and (2) the specification of user-request inputs and lookup functions, which requires domain-specific expert input.

Second, we need to show the completeness of hook placement. Our approach starts by placing a hook at every security-sensitive operation identified. Note that this placement provides complete mediation of the operations that were identified in the first step. We now have to show that subsequent optimization phases do not violate complete mediation. The first hoisting stage shown in Algorithm 1 propagates the hooks pertaining to operations that are

common to all branches of a control statement in a bottom-up fashion in the CDG. This stage does not remove any hooks. The second redundancy removal stage in Algorithm 3 propagates information about hooks placed in a top-down fashion in the CDG. When each node n of the CDG is processed, the set of propagated hooks that reach n represent the hooks that control dominate n. Therefore, if a hook placed at node n is in the set of propagated hooks then that hook can be safely removed without violating the completeness guarantee. Therefore, the two hook optimization stages of hook maintain completeness.

5. IMPLEMENTATION

We have implemented our tool using the CIL framework [25] and all our code is written in OCaml. Our implementation consists of the following modules, all written in OCaml:

- A call graph constructor: It consists of 237 lines of code and uses a simple function pointer analysis. Any function whose signature matches that of a function pointer and whose address is taken is considered a potential callee. This analysis is conservative in the absence of typecasts.

- Static Taint tracker: It consists of 438 line of code. We currently do not use an alias analysis. We found that the alias analyses provided with the CIL distribution did not terminate within reasonable time period for some of our larger code bases such as the X server and `postgres`. Not having precise alias analysis may currently lead to false negatives in the results of static taint tracking. However, we found that our tool gives fairly accurate results even in the absence of alias analysis, presumably since alias analysis does not greatly affect user-request propagation. We hope to improve the precision of our tool using alias analysis as part of the future work. We also make conservative assumptions about library functions which are not defined inside the code we analyze. We assume that when we encounter such function calls, all actual call parameters affect the result. We also need to consider the case where sensitive-objects are passed as parameters to such function calls.

- Automatic hook placement: It consists of a control dependence graph generator and the code to identify security-sensitive operations and the bottom up and top down algorithms for placing authorization hooks, written in 1196 lines of code.

Control Dependence Graph Construction We first build the intra-procedural control dependence graph for each procedure in the program using the built-in control flow graph and the visiting engine in CIL to compute postdominators. We then connect each procedure call site to the entry of the procedure. Note we do not do a context-sensitive inlined version of the control dependence graph since an operation is considered security-sensitive in all possible contexts. There can be multiple incoming edges to each procedure entry. But during the *bottom up operation accumulation* step of the hook placement, we do not propagate operations beyond the function entry point if it has multiple callers and in the *top down control flow dominance* step we do not propagate operations to an entry node unless it is performed at all the callers.

Specifications of Lookup Functions Identifying lookup functions for most object-oriented languages would be a straightforward language-level specification of standard container abstractions. Since the C language does not provide such standard abstractions, we provided specification at the level of common code patterns in C. We specify patterns for retrieval of objects from static and dynamic containers using indexing and pointer arithmetic and retrieval from recursive data structures by identifying instances where the next pointer is used inside loops to iterate through the container. In larger programs such as `postgres`, programmers typically implement their own containers. Therefore for postgres we created a program-level specification of lookup functions.

6. EVALUATION

We tested the tool on two types of user-space server programs: (1) two programs with manually-placed authorization hooks, namely, X server and *postgres*, and (2) two programs without hooks, viz. *memcached* and *pennmush*. We evaluated our approach using two metrics: reduction in programmer effort and accuracy relative to manual placements.

The key results of the analysis are: (1) programmer effort was reduced by 80-90% for the tasks of identifying security-sensitive variables (objects) and data structures and operations requiring mediation when compared to manual placement and (2) the method placed fine-grained authorization hooks for X server and *postgres* which are comparable to the manual hooks placed.

6.1 Reduction in Programmer Effort

We show how this tool aids the programmer by showing the reduction in the problem space that the different stages provide in Figure 4. We show how the tool helps in highlighting variables, data structures, statements, and operations that are of significance in placing authorization hooks using the following metrics.

1. *Lines of code:* The total number of line of code *LOC* the programmer would have to examine to analyze the program.

2. *Variables:* Total number of variables in the program (*All*), the number of tainted variables(*Tainted*), identified in Stage 1, the number of security-sensitive variables (*Sensitive*) identified in Stage 2.

3. *Data structures:* The total number of data structures defined in the program (*All*), the number of data structures that correspond to security-sensitive objects identified (*Sensitive*).

4. *Control Statements:* The total number of control statements in the program (*All*), the number of control statements predicated on user-choice (*User-choice*).

5. *Operations:* The number of operations incident on all user-choice control statements (*User-choice*), the number of security-sensitive operations (*Sensitive*) and the number of operations where hooks are placed after optimization (*Hook*).

Table 4 shows the reduction in problem space for the programs that we evaluated. We can see that we reduce the effort the programmer needs to put in from examining thousands of lines of code to a few hundred authorization hooks.

We can see that less than 50% of all variables are tainted by user-request, and less than 10% are security-sensitive, which correspond to less than 20% of all data structures in the program. The number of security-sensitive operations is less than 30% of all user-choice operations and the number of operations where hook placement is suggested is typically around 11% of user-choice operations. Since our hook placement is conservative and fine-grained the programmer would only need to examine the final hook placement operations to determine which ones to incorporate into the program. We therefore significantly reduce the amount of programmer effort needed to place authorization hooks.

Below, we examine the reduction in program effort for two programs that have no authorization hook placement.

Source		Variables			Data Structures		Control Statements		Operations			Performance
Program	LOC	All	Tainted	Sensitive	All	Sensitive	All	User-choice	User-choice	Sensitive	Hook	Time(m)
X server	28658	7795	2975 (38%)	823 (10%)	404	61 (15%)	4297	3170 (73%)	4760	1382 (29%)	532 (11%)	10.11
postgres	49042	12350	5100 (41%)	402 (3%)	278	30 (10%)	5821	3289 (56%)	5063	1378 (27%)	579 (11%)	53.7
memcached	8947	2350	490 (20%)	82 (3%)	41	7 (17%)	982	647 (65%)	996	203 (20%)	56 (5%)	0.48
pennmush	78738	24372	4168 (17%)	1573 (6%)	311	38 (12%)	20202	4135 (20%)	6485	1382 (21%)	714 (11%)	170.6

Figure 4: Table showing the reduction in programmer effort that the automated approach provides.

Memcached. This is a distributed memory object caching system for speeding up dynamic web applications. It is an in-memory key-value store of arbitrary chunks of data (strings, objects) from results of database calls, API calls, or page rendering. Our automated tool identifies seven security-sensitive data structures out of 41, each of which appeared to be critical to the correctness of the program. Two data structures covered the key-value pairs and statistics collected over their use. Another data structure was associated with a comment stating that modifying any variable of this type can result in undefined behavior. Also, security-sensitive global variables were identified that store program settings. By mediating access to variables of these data structures only, only 5% of the user-choice operations need to be examined by programmers.

PennMush. *PennMush* is a server of textual virtual reality used for social and role-playing activities which maintains maintains a world database containing players, objects, rooms, exits, and programs. Clients can connect to the server and take on characters in the virtual world and interact with other players. Our tool found 38 security-sensitive data structures of the 311 data structures in the program. We were able to confirm that at least 9 data structures are security-sensitive. Among them are data structure representing the object stored in the database and its attributes, communication channels and locks in addition to the cache, mail messages exchanged between players, and the player's descriptor data structures. These 714 hooks reduce the programmer effort by nearly 90% relative to the user-choice operations in the program.

6.2 Accuracy Relative to Manual Placement

For programs with hooks manually-placed, we verify the accuracy of our approach by comparing manual and automated hook placements.

X Server. We analyzed the `dix` module of the X server, and our tool automatically chose 532 placements in X server, which largely correspond to the 201 already placed manually, albeit at a finer granularity. While finer granularity placements provide flexibility in controlling objects and operations, programmers typically employ domain-knowledge that arises from writing policy specification to further optimize hooks. We discuss how our tool can help with such optimizations in Section 6.3.

Out of the 201 existing hooks, there were 7 cases where our tool could not map manually-placed hooks to automated hooks. We identify three causes. First, in three cases, there were no structure member accesses performed on the object mediated by the manual hooks. We found that sometimes manual hooks mediate objects that are not used. For example, in the function DIXLOOKUPCLIENT there is lookup of a resource stored in `pRes` which is never used again but is mediated by a manual hook. We will investigate whether these are bugs in the manual placement, but such anomalous placements can be identified by our tool. One case was due to a read of a global variable (screen saver) which according to our model will have to be specified as an input. Finally, three more cases was due to the fact that a client can request an operation on all elements of a container at the same time without naming a specific one. We propose the following simple extension to deal with

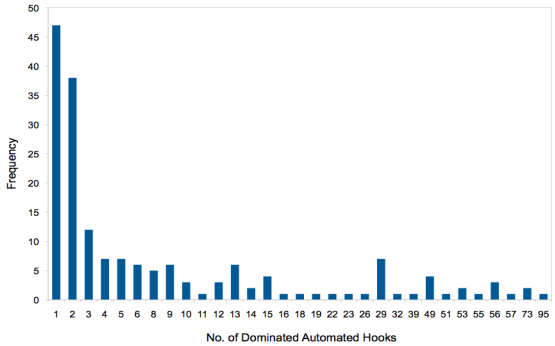

Figure 5: The figure shows the frequency of the number of automated hooks mapped to manual hooks in the X server.

this. If a container holds any security-sensitive objects (i.e., any container in which a lookup is used to retrieve objects), then all objects in the container are security-sensitive. Thus, a variable upon which a security-sensitive container object is data-dependent is also security-sensitive.

PostgreSQL. We analyzed the `tcop` module of Postgres that performs command dispatching from user requests. Our labeling of lookup operations is described in the Section 5. Postgres has 325 authorization hooks, including those provided by the SEPostgreSQL [19] project and with existing role-based access control hooks. Our tool identified 33 data structures corresponding to the security-sensitive operations, of which 22 were data types corresponding to catalog, tuples and relations. Our automated hook placement resulted in 579 hooks being placed for similar reasons to those discussed above for the X server. We found our hooks covered every one of the manually-placed hooks.

6.3 Helping the Programmer Place Hooks

Given our results, we find that the automated approach produces a finer-grained placement of hooks than the manual placement. To help the programmer understand the difference, we use the notion of *dominance* to compare manually-placed hooks to automated hooks. Automated hooks may be dominated by manual hooks in two ways.

Operation dominance: First, we consider the case where the automated approach identifies operations at a lower granularity. In this case, both the manual hook and automated hook are authorizing the same object retrieved from the same container. However, the coarser manual hooks require that the client have more permissions than are actually required to perform some of the operations in the code. Consider Figure 3. A manual hook placed at the entry node would require that the client have permissions for all the structure member accesses (name, format, data, and size) to the same object `pProp`, whereas only a subset are necessary in some branches (e.g., data and size for `Op2.2`). Thus, a hook at entry *operation-dominates* a hook at `Op2.2`.

Object Dominance: Second, we consider the case where we identify objects at a lower granularity. In this case, one object may have a field that is itself a container from which other security-

sensitive objects are retrieved. A hook on the parent object dominates the object retrieved from the container. For example, in Listing 2 any hook to authorize access to `pWin` *object-dominates* a hook to authorize access to `pProp`.

The additional stipulation in both cases is that the manual hook must also control flow dominate the automated hook. Based on these two notions of dominance, we can compute for each manual hook, the set of automated hooks it dominates. Figure 5 shows the statistics of this dominance relationship for the manual and automated hook placement in the X server. The graph shows the frequency distribution of the number of automated hooks mapped to manual hooks. We can see that for over 50% of the cases the mapping is between 1-3 automated hooks for each manual hook. We see that 30% more lie between 4-15 hooks. Beyond that, the numbers taper off except for a few interesting cases that result in larger mappings.

Our tool can, for each manual hook classify the mapped automated hook into three categories: those that are object-dominated, those that are operation-dominated, and, those that are dominated by multiple manual hooks. We give a sampling of some of the cases.

First, we look at the cases with 56 manual hooks. Of the 56 hooks, 43 hooks stem from two functions COPYGC and CHANGEGC both of which assign one of over 20 fields of the same object using a switch-case branching construct that is controlled by a mask provided by user-request. So our tool places 20 hooks as opposed to the one corresponding manual hook in both the functions. We see that for the programmer to reason about these 43 hooks, she only needs to reason about the two user-request controlled switch statements. The remaining hooks correspond to an additional seven control statements. Therefore, the programmer only has to deal with nine control statements to reason about this difference of 55 hooks. Similarly, a case with 29 hooks actually boils down to reasoning about six control flow statements.

We found several cases with object-dominance predominantly in the use of `child` windows of a particular window which were not explicitly authorized by manual hooks. In the case with 95 hooks, we found that it boiled down to 51 control flow statements and 5 objects.

We also saw cases where our tool placed a more optimized hook than the current manual hooks. For example, our tool placed a hook at the function FREERESOURCE which frees an resources from the resource table. In contrast, the manual hooks were placed at each of the callers instead.

Finally, we found that the performance of our tool is reasonable as shown in Figure 4. Majority of the time was spent in performing the topological and reverse topological sort of the control dependence graph. In comparison, manual effort to place such hooks which has been shown to take several years in the case of *X Server* and *postgres*.

7. RELATED WORK

We discuss related work in two categories.

1. *Automatic Hook Placement*: In past efforts, this typically involved using manual specification of some combination of subjects, objects operations and even hook code in order to place authorization hooks. In comparison, the only program-specific input specification we need is the few variables that represent the entry point of user requests in the program.
2. *Automatic Hook Verification*: This problem assumes that a certain number of hooks have already been placed correctly. This information is used to derive security-sensitive operations and the approach verifies if all instances of these operations are mediated.

7.1 Automated Hook Placement

Ganapathy *et al.* [11] presented a technique for automating hook placement assuming that the module implementing the hooks was already available. Their tool also required manually written code-level specifications of security sensitive operations. Using this information, they identified the set of operations that each hook protects and also the set of operations the program performs. Placing hooks is then a matter of matching operations in the code with the hooks that mediate those operations. Following this, they presented a hybrid static/dynamic analysis tool [12] that used program traces of security sensitive operations to derive the code level specifications automatically. For this technique to be effective, the user must know precisely which operations are security-sensitive and gather traces for them. In our tool, we automatically infer the set of security-sensitive operations as well as their code level specification using static analysis. The work most closely related to our approach is the one by Ganapathy et al [13] that used Concept Analysis [40] to group structure member accesses in program APIs in order to identify sets of structure member accesses that were frequently performed together in the program. In contrast to this work, we use a more intuitive technique for grouping structure member accesses into operations which falls out of the user's ability to make choices of objects and operations in the program. We envision that our hooks will therefore be at a granularity that is closer to a manual effort. The concept analysis approach also requires the user of the tool to specify APIs that are entry points into the program and the security-sensitive data structures. We identify the latter automatically. Our approach follows aspect-oriented programming paradigm [17]. In particular, each operation denotes a region of code before which a reference monitor hook must be placed, thereby identifying the join points.

7.2 Automated Hook Verification

There have been several tools to verify hook placement and detect missing hooks that employ both static and dynamic analyses. Zhang *et al*. [42] used simple manually specified security rules to verify the complete mediation property of reference monitors while Edwards et al. [7] used dynamic analysis to detect inconsistencies in the data structure accesses. Tan *et al.* [34] start with the assumption that production level code is already fairly mature and most of the hooks are already in place. They verify consistency of hook placement by using existing hooks to characterize security-sensitive operations in terms of structure member accesses, then check for unmediated operations. Srivastava *et al.* [31] use the notion that modern API's have multiple independent implementations. They use a flow and context sensitive analysis that takes as input multiple implementations of an API, and the definitions of security checks and security-sensitive events to see if the two implementations enforce the same security policy. In contrast to this, we attempt a first stab at solving the authorization hook problem on legacy programs and in the process define program level notions of security sensitive objects and operations which can also serve as a specification for automatically verifying that the existing authorization hooks in programs are comprehensive.

More recently there has been some work [30, 33] on automating authorization hook placement and verification for web applications that leverage programming paradigms of the web domain as specifications. In RoleCast [30] the authors verify the consistency of access hooks placed in web applications. They assume the presence of existing hooks, and the set of security-sensitive objects and operations are the backend database and database update op-

erations which are the same for all web applications they consider. They take advantage of the programming paradigm where web applications are built around pre-defined roles (such as admin, regular user) and different roles have different mediation requirements. So they use a combination of static analysis and heuristics to identify the portions of the application that implement the functionality relating to different roles. Sun *et al.* [33] mediate web applications at a much coarser granularity of web pages while also using a specification of roles, and infer the difference between sitemaps of privileged and unprivileged roles as the definition of security-sensitive operations. In contrast, our approach proposes concepts that are applicable uniformly in all domains.

8. CONCLUSION

Many efforts over the past decade have attempted to retrofit legacy servers with authorization hooks. Unfortunately, these efforts have largely been informal, community-driven exercises, often spanning multiple years. Moreover, vulnerabilities have been discovered in retrofitted servers, often years after the hooks were placed.

In this paper, we developed an approach that leverages the insight that deliberate user choice of objects from collections managed by the server should drive authorization decisions. We built a static analysis tool that implements this approach, and demonstrated its effectiveness on four real-world servers. Our approach can infer security-sensitive objects and operations in programs and place optimized hook placements. In addition, we also show how automated placement compare to manually placed hooks.

9. REFERENCES

[1] Implement keyboard and event security in X using XACE. `https://dev.laptop.org/ticket/260`, 2006.

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

[3] CARTER, J. Using GConf as an Example of How to Create an Userspace Object Manager. *2007 SELinux Symposium* (2007).

[4] CHANG, R., JIANG, G., IVANCIC, F., SANKARANARAYANAN, S., AND SHMATIKOV, V. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 186–199.

[5] COKER, G. Xen security modules (xsm). *Xen Summit* (2006), 1–33.

[6] D.WALSH. Selinux/apache. `http://fedoraproject.org/wiki/SELinux/apache`.

[7] EDWARDS, A., JAEGER, T., AND ZHANG, X. Runtime verification of authorization hook placement for the Linux security modules framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security* (2002), pp. 225–234.

[8] EPSTEIN, J., AND PICCIOTTO, J. Trusting X: Issues in building trusted X window systems -or- what's not trusted about X? In *Proceedings of the 14th Annual National Computer Security Conference* (October 1991).

[9] ERLINGSSON, Ú., AND SCHNEIDER, F. B. Irm enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy* (2000), pp. 246–255.

[10] FADEN, G. Multilevel filesystems in solaris trusted extensions. In *Proceedings of the 12th ACM symposium on Access control models and technologies* (New York, NY, USA, 2007), SACMAT '07, ACM, pp. 121–126.

[11] GANAPATHY, V., JAEGER, T., AND JHA, S. Automatic placement of authorization hooks in the Linux Security Modules framework. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Nov. 2005), pp. 330–339.

[12] GANAPATHY, V., JAEGER, T., AND JHA, S. Retrofitting legacy code for authorization policy enforcement. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (May 2006), pp. 214–229.

[13] GANAPATHY, V., KING, D., JAEGER, T., AND JHA, S. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)* (May 2007).

[14] GRAHAM, G. S., AND DENNING, P. J. Protection — principles and practice. In *Proceedings of the AFIPS Spring Joint Computer Conference* (May 1972), vol. 40, AFIPS Press, pp. 417–429.

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

[16] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IN 2006 IEEE SYMPOSIUM ON SECURITY AND PRIVACY* (2006), pp. 258–263.

[17] KICZALES, G., AND HILSDALE, E. Aspect-oriented programming. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2001), ESEC/FSE-9, ACM, pp. 313–.

[18] KILPATRICK, D., SALAMON, W., AND VANCE, C. Securing the X Window system with SELinux. Tech. Rep. 03-006, NAI Labs, March 2003.

[19] KOHEI, K. Security enhanced postgresql. `SEPostgreSQLIntroduction`.

[20] LIVSHITS, V. B., AND LAM, M. S. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), USENIX Association, pp. 18–18.

[21] LOSCOCCO, P. A., SMALLEY, S. D., MUCKELBAUER, P. A., ANd̃ S. J. TURNER, R. C. T., AND FARRELL, J. F. The Inevitability of Failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conferenc e* (October 1998), pp. 303–314.

[22] LOVE, R. Get on the D-BUS. `http://www.linuxjournal.com/article/7744`, Jan. 2005.

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

[24] NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction, 11th International Conference, CC 2002* (2002), Springer, pp. 213–228.

[25] NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction* (London, UK, 2002), CC '02, Springer-Verlag, pp. 213–228.

[26] POLITZ, J. G., ELIOPOULOS, S. A., GUHA, A., AND KRISHNAMURTHI, S. Adsafety: type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 12–12.

[27] SAILER, R., JAEGER, T., VALDEZ, E., CACERES, R., NALD PEREZ, R., BERGER, S., GRIFFIN, J. L., AND VAN DOORN, L. Building a MAC-based security architecture for the xen open-source hyperviso r. In *Proceedings of the 2005 Annual Computer Security Applications Conference* (Dec. 2005), pp. 276–285.

[28] Re: Adding support for SE-Linux security. `http://archives.postgresql.org/pgsql-hackers/2009-12/msg00735.php`, 2009.

[29] SE-PostgreSQL? `http://archives.postgresql.org/message-id/20090718160600.GE5172@fetter.org`, 2009.

[30] SON, S., MCKINLEY, K. S., AND SHMATIKOV, V. Rolecast: finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2011), OOPSLA '11, ACM, pp. 1069–1084.

[31] SRIVASTAVA, V., BOND, M. D., MCKINLEY, K. S., AND SHMATIKOV, V. A security policy oracle: detecting security holes using multiple api implementations. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 343–354.

[32] STAFFORD, J. A. *A Formal, Language-Independent, and Compositional Approach to Interprocedural Control Dependence Analysis*. PhD thesis, University of Colorado, 2000.

[33] SUN, F., XU, L., AND SU, Z. Static detection of access control vulnerabilities in web applications. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 11–11.

[34] TAN, L., ZHANG, X., MA, X., XIONG, W., AND ZHOU, Y. Autoises: automatically inferring security specifications and detecting violations. In *Proceedings of the 17th conference on Security symposium* (Berkeley, CA, USA, 2008), USENIX Association, pp. 379–394.

[35] VANCE, C., MILLER, T., AND DEKELBAUM, R. Security-enhanced darwin: Porting selinux to mac os x. *Proceedings of the Third Annual Security Enhanced Linux* (2007).

[36] WALSH, E. Integrating x.org with security-enhanced linux. In *Proceedings of the 2007 Security-Enhanced Linux Workshop* (Mar. 2007).

[37] WASSERMANN, G., AND SU, Z. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 32–41.

[38] WATSON, R. N. M. Trustedbsd: Adding trusted operating system features to freebsd. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 15–28.

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

[40] WILLE, R. Restructuring lattice theory: An approach based on hierarchies of concepts. In *Proceedings of the 7th International Conference on Formal Concept Analysis* (Berlin, Heidelberg, 2009), ICFCA '09, Springer-Verlag, pp. 314–339.

[41] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux security modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium* (August 2002), pp. 17–31.

[42] ZHANG, X., EDWARDS, A., AND JAEGER, T. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium* (August 2002), pp. 33–48.