Producing Hook Placements to Enforce Expected Access Control Policies

Divya Muthukumaran[1], Nirupama Talele[2], Trent Jaeger[2], and Gang Tan[3]

[1] Imperial College, London, United Kingdom
[2] The Pennsylvania State University, University Park, PA, United States
[3] Lehigh University, Bethlehem, PA, United States

Abstract. Many security-sensitive programs manage resources on behalf of mutually distrusting clients. To control access to resources, *authorization hooks* are placed before operations on those resources. Manual hook placements by programmers are often incomplete or incorrect, leading to insecure programs. We advocate an approach that automatically identifies the set of locations to place authorization hooks that mediates all security-sensitive operations in order to enforce expected access control policies at deployment. However, one challenge is that programmers often want to minimize the effort of writing such policies. As a result, they may remove authorization hooks that they believe are unnecessary, but they may remove too many hooks, preventing the enforcement of some desirable access control policies.

In this paper, we propose algorithms that automatically compute a minimal authorization hook placement that satisfies constraints that describe desirable access control policies. These *authorization constraints* reduce the space of enforceable access control policies; i.e., those policies that can be enforced given a hook placement that satisfies the constraints. We have built a tool that implements this authorization hook placement method, demonstrating how programmers can produce authorization hooks for real-world programs and leverage policy goal-specific *constraint selectors* to automatically identify many authorization constraints. Our experiments show that our technique reduces manual programmer effort by as much as 58% and produces placements that reduce the amount of policy specification by as much as 30%.

1 Introduction

Programs that manage resources on behalf of mutually distrusting clients such as databases, web servers, middleware, and browsers must have the ability to control access to operations performed when processing client requests. Programmers place *authorization hooks*[4] in their programs to mediate access to such operations[5]. Each authorization hook guards one or more operations and enables the program to decide at runtime whether to perform the operations or not, typically by consulting an access control policy. The access control policy is deployment-specific and restricts the set of operations that each subject is allowed to perform as shown in Figure 1.

[4] Authorization Hooks are also known as Policy Enforcement Points.

[5] There are several projects specifically aimed at adding authorization hooks to legacy programs of these kinds [4, 5, 10, 11, 16, 20].

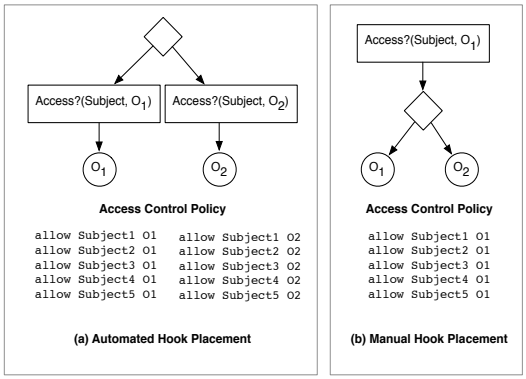

Fig. 1: Figure showing the effect of hook placement granularity on access control policy

There are two main steps programmers perform when placing authorization hooks in programs. First, they must locate the program operations that are security-sensitive. Researchers [7–9, 13, 17] have examined different techniques to infer this using a combination of programmer domain knowledge and automated program analysis. Second, programmers must decide where to place authorization hooks to mediate these security-sensitive operations. Finding location to place authorization hooks is akin to finding *joinpoints* in aspect-oriented programming. Prior automated approaches have suggested hook placements that are very fine grained when compared to placements generated manually by programmers. Understanding this difference is crucial to developing tools for automated hook placement and is the focus of this paper.

The granularity of authorization hook placement affects the granularity of access control policies that must be written for deploying programs. Placing hooks at a very fine granularity means that the access control policy that needs to be written to deploy the program becomes very fine-grained and complex, hindering program deployment. Consider the example (a) in Figure 1 where an automated approach places hooks at two distinct security-sensitive operations O_1 and O_2. If there are five subjects in a deployment, then the policy may need up to 10 rules to specify that the five subjects should have access to both operations.

To facilitate widespread adoption of their programs, programmers want to avoid complex access control policies at deployment. Consequently, they use their domain knowledge to trade off flexibility of the access control policies in favor of simplified policies. For example, an expert may know that when policies are specified for the program, every subject who is authorized to perform O_1 must also be authorized for O_2. With respect to authorization, O_1 and O_2 are equivalent and therefore it is sufficient to insert one hook to mediate both operations as shown in part (b) of Figure 1, resulting in a smaller policy with only five rules.

Minimality in hook placement must be balanced by the two main security requirements of enforcement mechanisms - *complete mediation* [2], which states that every security-sensitive operation must be preceded by an appropriate authorization hook, and the *principle of least privilege*, which states that access control policies only au-

thorize permissions needed for a legitimate prurpose. At present, programmers make these trade-offs between minimality, complete mediation and least privilege manually, which delays the introduction of needed security measures [16] yet still requires many refinements after release.

Authorization Constraints. We believe that making the programmers' domain knowledge explicit is the key to solving the problem of authorization hook placement. We propose an approach that is based on programmers specifying constraints on program operations (e.g., structure member accesses), which we call *authorization constraints*. Authorization constraints restrict the access control policies that may be enforced over a set of operations. We use two kinds of constraints in our system. First, *operation-equivalence* means that two distinct security-sensitive operations on parallel code paths must be authorized for the same set of subjects, which enables a single authorization hook to mediate both. The second kind is *operation-subsumption*, meaning that a subject who is permitted to perform one operation must be allowed to perform the second operation. Therefore, if the second operation follows the first in a code path, then an authorization hook for the second operation is unnecessary.

To use authorization constraints to place authorization hooks automatically, we have to overcome two challenges. First, given a set of authorization constraints, a method needs to be developed to use these constraints to minimize the number of authorization hooks placed to enforce any policy that satisfies the constraints. Second, there should be a method to help programmers discover authorization constraints. In this paper, we provide solutions to both of these challenges.

Minimizing Hook Placement. We address the first challenge by developing an algorithm that uses authorization constraints to eliminate hooks that are unnecessary. Specifically, we show that: (1) when operations on parallel code paths are *equivalent*, the hook placement can be *hoisted*, and (2) when operations that *subsume* each other also occur on the same control-flow path, we can *remove* the redundant hook. This method is shown to produce a minimal-sized[6] hook placement automatically can enforce any access control policy that satisfies the authorization constraints.

Discovering Authorization Constraints. In order to help programmers avoid the cumbersome task of writing complete authorization constraints, we introduce the notion of *constraint selectors*. These selectors are able to make a set of constraint choices on behalf of the programmer based on higher-level goals. For example, suppose the programmer's goal is for her program to enforce multi-level security (MLS) policies, such as those expressed using the Bell-La Padula model [3]. The Bell-La Padula model only reasons about read-like and write-like operations, so any two security-sensitive operations that only perform reads (or writes) of the same object are equivalent. Thus, a constraint selector for MLS guides the method to create equivalence constraints automatically for such operations.

We have designed and implemented a source code analysis tool for producing authorization hook placements that reduce manual decision making while producing placements that are minimial for a given set of authorization constraints. The tool requires

[6] Minimality in hook placement is conditional on having precise alias analysis and path sensitive analysis

only the program source code and a set of security-sensitive operations associated with that code, which can be supplied by one of a variety of prior methods [7–9, 13, 17]. Using the tool, the programmer can choose a combination of constraint selectors, proposed hook placements, and/or hoisting/removal choices to build a set of authorization constraints. Once constraints are established, the tool computes a complete authorization hook placement that has minimum number of hooks with respect to the constraints. We find that using our tool reduces programmer effort by reducing the number of possible placement decisions they must consider, by as much as 67%. Importantly, this method removes many unnecessary hooks from fine-grained, default placements. For example, simply using the MLS constraint selector removes 124 hooks from the default fine-grained placement for X Server.

In this paper, we demonstrate the following contributions:

- We introduce an automated method to produce a minimal authorization hook placement that can enforce any access control policy that satisfies a set of authorization constraints.
- We simplify the task of eliciting authorization constraints from programmers, by allowing them to specify simple high level security goals that are translated into authorization constraints automatically.
- We evaluate a static source code analysis that implements the above methods on multiple programs, demonstrating that this reduces the search space for programmers by as much as 58% and produces placements that reduce the number of hooks by as much as 30%

We believe this is the first work that utilizes authorization constraints to reduce programmer effort to produce effective authorization hook placements in legacy code.

2 Motivation

2.1 Background on Hook Placement

Authorization is the process of determining whether a *subject* (e.g., user) is allowed to perform an *operation* (e.g., read or write an object)[7] by checking whether the subject has a *permission* (e.g., subject and operation) in the *access control policy* that grants the subject with access to the operation. Authorization is necessary because some operations are *security-sensitive operations*, i.e., operations that cannot be performed by all subjects, so access to such operations must be controlled. An *authorization hook* is a program statement that submits a query to check for an authorizing permission in the access control policy. The program statements guarded by the authorization hook may only be executed following an authorized request (control-flow dominance). Otherwise, the authorization hook will cause some remedial action to take place.

Related Work: The two main steps in the placement of authorization hooks are the identification of security-sensitive operations and the identification of locations in

[7] We acknowledge that many access control policies distinguish objects (e.g., files) from the accesses (e.g., read or write), which are often called operations. We define operations to include the object and access type in this paper.

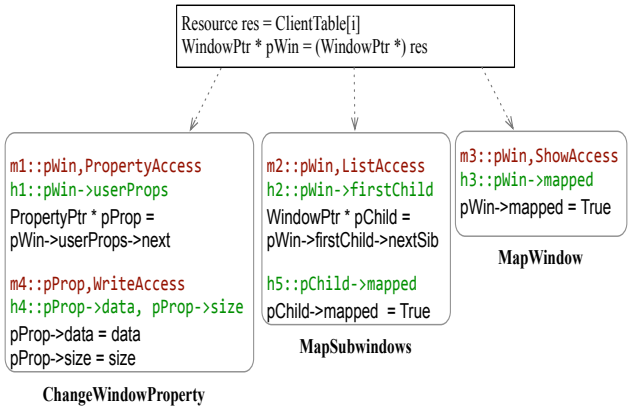

Fig. 2: Hook Placement for functions MAPWINDOW, MAPSUBWINDOWS, and CHANGEWINDOWPROPERTY

the code where authorization hooks must be placed in order to mediate such operations. Past efforts focused mainly on the first problem, defining techniques to identify security-sensitive operations [6–9, 13, 15, 17–19, 23]. Initially, such techniques required programmers to specify code patterns and/or security-sensitive data structures manually, which are complex and error-prone tasks. However, over time the amount of information that programmers must specify has been reduced. In our prior work, we infer security-sensitive operations only using the sources of untrusted inputs and language-specific lookup functions [13].

When it comes to the placement of authorization hooks, prior efforts typically suggest placing a hook before every security-sensitive operation in order to ensure complete mediation. There are two problems with this simple approach. First, automated techniques often use low-level representations of security-sensitive operations, such as individual structure member accesses, which might result in many hooks scattered throughout the program. More authorization hooks mean more work for programmers in maintaining authorization hooks and updating them when security requirements change. Second, such placements might lead to redundant authorization, as one hook may already perform the same authorization as another hook that it dominates. In our prior work, we have suggested techniques to remove hooks that authorize structure member accesses redundantly [13]. However, this approach still does not result in a placement that has a one-to-one correspondence with hooks placed manually by domain experts. In X server, it was found that while the experts had placed approximately 200 hooks, the automated technique suggested approximately 500 hooks. In the following subsections we discuss some reasons for this discrepancy.

2.2 How Manual Placements Differ

We find that there are typically two kinds of optimizations that domain experts perform during hook placement. We follow with examples of both cases from the X Server.

Listing 1 Example of manual hoisting in the COPYGC function in the X server.

```
1   /*** gc.c ***/
2   int CopyGC(GC *pgcSrc, GC *pgcDst, BITS32 mask){
3    switch (index2)
4     {
5         result = dixLookupGC(&pGC, stuff->srcGC,
6                              client,DixGetAttrAccess);
7         if (result != Success)
8            return BadMatch;
9         case GCFunction:
10           /* Hook(pgcSrc, [read(GC->alu)]) */
11           pgcDst->alu = pgcSrc->alu;
12           break;
13        case GCPlaneMask:
14           /* Hook(pgcSrc, [read(GC->planemask)]) */
15           pgcDst->planemask = pgcSrc->planemask;
16           break;
17        case GCForeground:
18           /*  Hook(pgcSrc, [read(GC->fgPixel)]) */
19           pgcDst->fgPixel = pgcSrc->fgPixel;
20           break;
21        case GCBackground:
22           /* Hook(pgcSrc, [read(GC->bgPixel)]) */
23           pgcDst->bgPixel = pgcSrc->bgPixel;
24           break;
25          /* .... More similar cases */
26       }
27  }
```

– First, assume there are two automatically placed hooks H_1 and H_2 such that the former dominates the latter in the program's control flow. The placement by the domain expert has a matching hook for H_1 but not for H_2. We can interpret this as the expert having *removed* (or otherwise omitted) a finer-grained hook because the access check performed by H_1 makes H_2 redundant.
– The automated tools place hooks $(H_1, ..., H_n)$ at the branches of a control statement. The domain expert has not placed hooks that map to any of these hooks, but instead, has placed a hook M_1 that dominates all of these hooks[8]. The expert has *hoisted* the mediation of operations at the branches to a common mediation point above the control statement as shown in Figure 1.

First, examine the code snippet in Figure 2. In the figure, hooks placed by the programmer have prefixes such as `m1::` and hooks placed by an automated tool [13] have prefixes such as `h1::`. The function MAPWINDOW performs the operation `write(pWin→mapped)` on the window, which makes the window viewable. We see that the programmer has placed a hook `m3::(pWin,ShowAccess)` that specifically authorizes a subject to perform this operation on object represented by `pWin`. Access mode `ShowAccess` identifies the operation. The requirement of consistency in hook placement dictates that an instance of hook `ShowAccess` should precede any instance of writing to the `mapped` field of a Window object that is security-sensitive. MAPSUBWINDOWS performs the same operation on the child windows `pChild` of a window `pWin`. While the automated tool prescribes a hook at MAPSUBWINDOWS, we find that the domain expert has chosen not to place a corresponding hook. MAPSUBWINDOWS is preceded by the manual hook `m2::(pWin,ListAccess)` for the subject to be able to list the child windows of window `pWin`, but there is no hook to authorize the operation `write(pChild->mapped)`[9].

Second, look at the example shown in Listing 1. The function COPYGC in the X server accepts a source and target object of type `GC` and a `mask` that is determined by

[8] There is no hook in the automated placement that matches M_1.
[9] We discuss the relevance of CHANGEWINDOWPROPERTY below.

the user request and, depending on the mask, one of 23 different fields of the source are copied to the target via a `switch-case` control statement. Since each branch results in a different set of structure member accesses, the automated tool infers that each branch performs a different security-sensitive operation. Therefore, it suggests placing a different hook at each of the branches. On the contrary, there is a single manually placed hook that dominates the control statement, which checks if the client has the permission `DixGetAttrAccess` on the source object. Therefore a single manually placed hook replaces 23 automated hooks in this example.

2.3 Balancing Minimality with Least Privilege

We have in the X Server a mature codebase, which has been examined over several years by programmers in order to reach a consensus on hook placement. We are convinced that a deliberate choice being made by the experts about where to place hooks and which hooks to avoid on a case-by-case basis. For example, in CHANGEWINDOWPROPERTY in Figure 2, a property `pProp` of `pWin` is retrieved and accessed. Programmers have placed a finer-grained hook `m4::(pProp,WriteAccess)` in addition to the hook `m1::(pWin,SetPropertyAccess)`. Contrast this with the MAPSUBWINDOWS example where they decided not to mediate the access of a child object.

The fundamental difference between a manually written hook placement and an automatically generated one is in the granularity at which security-sensitive operations are defined. When the automated tool chooses to place a hook at each of the branches of a control statement, it implicitly identifies security-sensitive operations at a finer granularity than experts. The choice of granularity of security-sensitive operations is an exercise in balancing the number of hooks placed and least privilege. A fine-grained placement allows more precision in controlling what a subject can do, but this granularity may be overkill if programmers decide that subjects must be authorized to access operations in an all-or-nothing fashion. For the switch statement with 23 branches, having 23 separate hooks will lead to a cumbersome policy because the policy will have 23 separate entries for each subject. Since the programmers decided that all subjects either can perform all 23 operations for an object or none, it is preferable to have a single hook to mediate the 23 branches.

We have also seen that even with manual hook placement multiple iterations may be necessary to settle on the granularity that balances least privilege and minimality in hook placement. For example, the X server version of 2007 had only four operation modes, namely, `read`, `write`, `create` and `destroy`. But during the subsequent release, the programmers replaced these with 28 access modes that were necessary to specify and enforce policies with finer granularity. Since the first release of X server with the XACE hooks in 2007, the hooks have undergone several changes. Over 30 hooks were added to the X server code base, and some existing hooks were also removed, moved or combined with other hooks [22]. Some of these changes are documented in the XACE specification [21]. We believe that observing typical policy specifications at runtime enabled the programmers to add and remove hooks in subsequent versions of the application. We want to understand iterative refinement and build methods to automate some tasks in the process, making programmer decisions explicit.

3 Authorization Hook Placement Problem

Authorization hook placement involves two main steps: a) finding security-sensitive operations (SSOs) in the program and, b) placement of hooks to satisfy a set of requirements. In this section we give a brief background into approaches that tackle the former, followed by our intuition for how to approach the latter problem.

3.1 Identifying Security-Sensitive Operations

We provide some background on our prior research [13] in automatically identifying the set O of security-sensitive operations (SSOs) in programs using static analysis. Each SSO is represented using a variable v and a set of read and write structure member accesses on the variable. For example, in the CHANGEWINDOWPROPERTY function shown in Figure 2, the last two statements produce a security-sensitive operation `pProp:[write(data), write(size)]`. There may be multiple instances of an SSO in a program. Each instance is represented using the tuple (o, l) where o is the SSO and l is the location (statement) in the code where the instance occurs. Let O_L be the set of all instances of all the SSOs in the program. Our goal is to place authorization hooks to mediate all the elements of O_L.

Definition 1 *An authorization hook is a tuple (O_h, l_h) where l_h is a statement that contains the hook and $O_h \subseteq O_L$ is a set of security-sensitive operation instances mediated by the hook.*

A set of authorization hooks is called an *Authorization Hook Placement*. The approach in [13] produces a Control Dependence Graph (CDG) of the program to represent program statements and hooks. A CDG of a program $CDG = (L, E)$ consists of a set of program statements L and edges E, which represent the control-dependence relationships between statements. Since this exposes the statements that a given statement depends upon for execution, it enables computation of authorization hook placements that mediate all control flows in the program by ensuring that every operating instance in O_L is included in at least one authorization hooks O_h. We will continue to use the CDG representation in this work for refining hook placements.

3.2 Consolidating Hook Placements

As mentioned in Section 2, the inital placement of hooks by automated techniques is fine-grained, i.e., typically at every security-sensitive operation instance that is identified. Our intuition is that constraints on the access control policies to be enforced in the program can be leveraged to consolidate authorization hooks in order to achieve the right granularity for hook placements. Let U be the set of all subjects for a hypothetical access control policy $\mathcal{M}$ for the given program. Let $Allowed$ be a function that maps each security-sensitive operation in O to subjects U that are allowed to perform the operation according to policy $\mathcal{M}$. We identify two cases that are relevant to the placement of authorization hooks:

- **Invariant I**: First, given any two operations o_1 and o_2, if access control policy $\mathcal{M}$ permits $Allowed(o_1) = Allowed(o_2)$ then o_1 and o_2 are equivalent for the purpose of authorization. This means that any hook that mediates o_1 and dominates o_2 in the code automatically authorizes o_2 and vice versa.
- **Invariant II**: Second, given two operations o_1 and o_2, if access control policy $\mathcal{M}$ permits $Allowed(o_1) \subset Allowed(o_2)$ then operation o_1 'subsumes' o_2 for the purpose of authorization. This means that a hook that mediates o_1 and dominates o_2 can also mediate o_2 but not vice-versa.

As described above, we have observed that programmers often assume that their programs will enforce access control policies that satisfy Invariant I (e.g., see Listing 1) and Invariant II for *every* access control policy $\mathcal{M}$ that may be enforced by their programs (see Figure 2). This observation leads us to believe that in order to consolidate hooks we need to impose equivalence and partial-order relationships between the elements in O. Therefore, we define a *set of authorization constraints* as follows:

Definition 2 *A* set of authorization constraints $\mathcal{P}$ *is a pair* (S, Q) *of relationships between SSOs in the program, where* Q *stands for* **equivalence** *and* S *stands for* **subsumption**.

We can see that the equivalence relationship Q results in a partitioning of the set O of security-sensitive operations. Let O_Q be the set of partitions produced by Q. The subsumption relationship S imposes a partial order between the elements in O_Q. We can use these two relationships to consolidate hooks on parallel code paths to a dominating program location (hook *hoisting* operation) or eliminate a redundant hook on a post-dominating program location (hook *removal* operation). We describe the technique for this in the Section 4.

The challenge we address in this paper is how to use authorization constraints to consolidate hook placements. Our system uses an algorithm that, given a program and its set of authorization constraints, generates a minimal authorization hook placement that satisfies complete mediation and least privilege. **Complete mediation** states that every SSO instance in a program should be dominated by an authorization hook that mediates access to it. A placement that enforces **least privilege** ensures that during an execution of a program, a user of the program (subject) is only authorized to perform (or denied from performing) the SSOs requested as part of that execution. This effectively puts a constraint on how high a hook can be hoisted.

We observe that even though automated placement methods may be capable of producing placements that can enforce any policy, and thus can enforce least privilege, programmers will not accept a hook placement unless there is a justified need for that hook. Specifically, programmers only want a hook placed if that hook will actually be necessary to prevent at least one subject from accessing a security-sensitive operation. That is, while programmers agree that they should give subjects the minimal rights, they also require *that a program should have only the minimal number of hooks necessary to enforce complete mediation and least privilege*.

4 Design

There are two main inputs to our approach: a) the set O of SSOs and their instances O_L in the program, and b) the control dependence graph CDG of the program. We have defined these inputs in Section 3.1 and discussed prior work that uses static analysis to infer them automatically.

Step 1: Generating a default placement: First, using O, O_L and CDG, we generate a default placement. Since the elements in O_L are in terms of code locations, and the nodes in the CDG have location information, we can create a map $C2O$ from each node s_i in the CDG to the set of SSO instances in O_L that occur in the same location as s_i. The default placement $\mathcal{D}$ has a hook at each node s_i where $C2O[s_i] \neq \emptyset$.

Step 2: Generating constrained hook placement: If a set of authorization constraints $\mathcal{P}_i$ is already available, step 2 of our tool is able to automatically generate a minimal placement[10] $\mathcal{E}_i$ that can enforce any access control policy that satisfies the constraints. Our approach uses the *equivalence* constraints to perform *hoisting* and *subsumption* constraints to perform *removal*, thereby minimizing the number of hooks. This procedure is described in Section 4.1. We discuss how we may assist programmers in selecting authorization constraints for their program using high-level goals encoded as constraint selectors in Section 4.2.

4.1 Deriving a constrained placement

Given default placement $\mathcal{D}$ and set of authorization constraints $\mathcal{P}$, our system can derive a candidate constrained placement $\mathcal{E}$ that satisfies complete mediation and least privilege enforcement with the minimum number of hooks (used in step 3 of our design). The subsumption S and equivalence Q relationships in $\mathcal{P}$ enable us to perform two different hook refinements on $\mathcal{D}$ to derive the $\mathcal{E}$. We present the algorithm next and the proof why it has desired properties is in Appendix A.

Hoisting The first refinement is called *hoisting* and it aims to consolidate the hooks for mediation of equivalent operation instances that are siblings (appear on all branches of control statements). This lifts hook placements higher up in the CDG based on Q. Given a node s_i in the CDG, if each path originating from that node s_i contains SSOs that are in the same equivalence class in O_Q, then we can replace the hooks at each of these paths with a single hook at s_i. This relates to the example in Listing 1, where if the operations along all the 23 branches of the $Switch$ statement in line 3 are equivalent, then we can replace the 23 automatically generated hooks at those branches with a single hook that dominates all of them.

Algorithm 1 shows how hoisting is done. It uses the CDG and the $C2O$ map as inputs. Accumulator α gathers the set of SSOs at each node s_i by combining the $C2O$ of s_i with the α mapping from the child nodes. The algorithm traverses the CDG in reverse topological sort order and makes hoisting decisions at each node in the CDG. We partition the set of nodes in the CDG into two types - control and non-control nodes. Control nodes represent control statements (such as if, switch etc) where hoisting can

[10] Henceforth referred to as a *constrained placement*.

Algorithm 1 Algorithm for hoisting

$top' = TopoSortRev(CDG)$
while $top' \neq \emptyset$ **do**
  $s_i = top'.pop()$
  if $isControl(s_i)$ **then**
    $\alpha[s_i] = C2O[s_i] \cup \bigcap_j^Q \{\alpha[s_j] \mid (s_i, s_j) \in CDG\}$
  else
    $\alpha[s_i] = C2O[s_i] \cup \bigcup_j \{\alpha[s_j] \mid (s_i, s_j) \in CDG\}$
  end if
end while

Algorithm 2 Algorithm for removal

$top = TopoSort(CDG)$
while $top \neq \emptyset$ **do**
  $s_i = top.pop()$
  $O_D = \bigcap_j^{QS} \{\phi[s_j] \mid (s_j, s_i) \in CDG\}$
  $\phi[s_i] = \alpha[s_i] \cup O_D$
  $O_R = \emptyset$
  for all $o_m \in \alpha[s_i]$ **do**
    if $\exists o_n \in O_D, (o_n\ S\ o_m)$ or $(o_n\ Q\ o_m)$ **then**
      $O_R = O_R \cup \{o_m\}$
    end if
  end for
  $\beta[s_i] = \alpha[s_i] - O_R$
end while

be performed. At control nodes[11], we perform the intersection operation $\bigcap^Q$ which uses the equivalence relation Q to perform set intersection in order to consolidate equivalent SSOs. Note that this intersection operation limits how high hooks may be hoisted in the program. At non-control nodes, we accumulate SSOs from children using a union operation.

Note that this algorithm does not remove any hooks. It places new hooks that dominate the control statements where hoisting occurs. For example, given Listing 1, the algorithm would place a new hook before the Switch statement. The removal operation which we discuss next will eliminate the 23 hooks along the different branches because of the new hook that was placed by this algorithm.

Removal The second refinement is called redundancy removal and aims to eliminate superfluous hooks from CDG using S. Whenever a node s_1 that performs SSO o_1 dominates node s_2 that performs SSO o_2 and o_1 either subsumes or is equivalent to o_2 according to $\mathcal{P}$, then a hook at s_1 for o_1 automatically checks permissions necessary to permit o_2 at s_2. Therefore, we may safely remove the hook at s_2 without violating complete mediation.

In the example in Figure 2, if we had authorization constraints specify that operation $read(pWin \rightarrow firstChild)$ subsumes (or is equivalent to) operation $write(pChild \rightarrow mapped)$, then we do not need the suggested hook `h5`.

Algorithm 2 shows how the removal operation is performed. The algorithm takes as input the CDG and the map α computed by the hoisting phase. It traverses the CDG in topological sort order (top-down) and at each node s_i makes a removal decision based on the set of operations checked by all hooks that dominate s_i. The accumulator ϕ

[11] Each control node has dummy nodes as children each representing a branch of the control node

stores for each node s_i the set of operations checked at s_i and all nodes that dominate s_i. While processing each node, the algorithm computes O_D, which is the set of operations checked at dominators to node s_i. Note that the CDG is constructed interprocedurally (refer to Section 3.1) and a node can have multiple parents at function calls[12]. The $\bigcap^{QS}$ that combines authorized operations in case of multiple parents is shown in Algorithm 3.

Algorithm 3 Compute $\bigcap^{QS}$ on two sets O_i and O_j. Returns result in O_T.

```
O_T = ∅
for all o_i ∈ O_i do
    for all o_j ∈ O_j do
        if o_i Q o_j then
            O_T = O_T ∪ {o_i}
        end if
        if o_i S o_j then
            O_T = O_T ∪ {o_j}
        end if
        if o_j S o_i then
            O_T = O_T ∪ {o_i}
        end if
    end for
end for
```

Next, Algorithm 2 creates the set O_R which is the set of operations that do not have to be mediated at s_i since they are either subsumed by or equivalent to operations that have been mediated at dominators to s_i. The resulting map β from nodes to the set of operations that need to be mediated at the node gives the final placement. The constrained placement $\mathcal{E}$ suggests a hook at each node s_i such that $\beta[s_i] \neq \emptyset$.

Note that both the bottom-up hoisting and top-down removal must be performed in sequence to get the final mapping from nodes to the set of SSOs that need mediation.

4.2 Helping Programmers Produce Placements

In this section we discuss how a programmer might produce a hook placement using the approach presented above. More specifically, would the programmer have to manually generate a fine-grained set of authorization constraints in order to use our approach? We envision an approach where programmers only need to specify high-level security goals as opposed to fine-grained set of authorization constraints.

Suppose the programmer wishes to enforce a well-known security policy, such as Multi-Level Security [12]. In MLS, subjects are assigned permissions at the granularity of read and write accesses to individual objects. In our method, program objects are referenced by variables in program operations, so any MLS policy that permits a subject to read a field of a variable also permits that subject to read any other fields of that variable; a similar case holds for writes. This means that all read-like (write-like) accesses of a variable can be treated as equivalent. The programmer can produce a small

[12] We avoid cycles in the CDG by eliminating back edges. This is for the purpose of being able to sort nodes topologically in the CDG. When we perform hook hoisting and removal, it is the control dominance information in a CDG that gets used in our algorithms; so eliminating back edges would not affect our analysis.

Program	LOC	MANUAL	DEFAULT	MLS		
				Total	*%-Reduction*	*%-Difference*
X Server 1.13	28K	207	420	296	30	58
Postgres 9.1.9	49K	243	360	326	9	29
Linux VFS 2.6.38.8	40K	55	139	135	2	5
Memcached	8K	0	32	30	6	n/a

Table 1: Table showing the lines of code (LOC), number of manual hooks (MANUAL), default automated hooks (DEFAULT), and the impact of using the MLS constraint selectors for hook placement, including the resultant number of hooks (Total), percent reduction in total hooks (%-Reduction), and the percent reduction in the difference between the manual and automated placements (%-Difference).

set of such rules to encode the security policy (we call this a constraint selector) which serves as an input to the placement approach in lieu of a complete set of authorization constraints.

Whenever a hoisting or a removal decision needs to be made by the approach, the constraint selector will serve as an oracle that aids in this decision. The MLS constraint selectors will stipulate that whenever all the branches perform either only reads or only writes of the same variable (irrespective of the fields being read) a hoisting operation should succeed. Similarly, if a hook mediated the read operation of a variable and is dominated by a hook that mediates the read of the same variable (irrespective of the field being accessed) the removal operation should succeed. The programmers can create any such constraint selector that encodes relationships between data types that may be application specific, or even encode the results or auxiliary static and dynamic analysis.

5 Evaluation

We implemented our approach using the CIL [14] program analysis framework and all our code is written in OCaml. The CDG construction and default hook placement is similar to the approach mentioned in prior work [13]. Our prototype implementation does not employ precise alias analysis or a path sensitive analysis which may produce some redundant hooks (e.g. in code paths that are not feasible).

Our goal with the evaluation was to answer two questions:
a) Does the approach produce placements that are closer to manually placed hooks?
b) Does the approach reduce programmer effort necessary to place authorization hooks?

In order to evaluate our approach, we compare hook placement produced by using constraint selectors against the default hook placement produced using the technique presented in our prior work [13]. We perform this comparison along two dimensions:

- First, we determine the number of hooks produced using both techniques and show that using constraint selectors reduces the number of hooks by as much as 30%. We also show that using the constraint selectors reduces the gap between manual and automated placements by as much as 58% compared to the default approach [13].
- Second, we show that using constraint selectors reduces the programmer effort, measured in terms of the number of authorization constraint options to manually consider by as much as 58%.

Program	DEFAULT		MLS	
	REMOVE	*HOIST*	*REMOVE*	*HOIST*
X Server 1.13	237	55	113	10
Postgres 9.1.9	208	42	146	21
Linux VFS 2.6.38.8	53	4	49	3
Memcached	8	1	6	0

Table 2: Table showing the hoisting (HOIST) and removal (REMOVE) suggestions in the default placement (DEFAULT) and placements generated using the constraint selectors (MLS).

Evaluating Hook Reduction: Table 1 shows the total number of hooks placed for each experiment. 'LOC' shows the number of lines of code that were analyzed, 'MANUAL' shows the number of hooks placed manually by domain experts, 'DEFAULT' shows the number of hooks placement in the default case by the automated technique and 'MLS' shows how using the constraint selectors affects the total number of hooks and how this compares with the number of hooks placed manually. Within the 'MLS' column, 'Total' refers to the total number of hooks placed when using the constraint selectors '%-Reduction' refers to the percentage reduction in number of hooks compared to the default placement and '%-Difference' refers to percentage reduction in the gap between automated and manual placement when compared against the default placement. For example, in the case of X Server we see that in the experiment we considered 28K lines of code, where programmers had placed 207 hooks manually, whereas the default placement suggested 420 hooks. When the 'MLS' constraint selectors was used, the number of hooks suggested by the automated technique went down to 296 which is a 30% reduction in the number of hooks compared to manual placement and reduces the gap between automated and manual hook placements by 58%.

Evaluating Programmer Effort: We define programmer effort as the search space of authorization constraints that the programmer has to examine manually. Therefore, we measure the reduction in programmer effort by counting the number of *hoisting* and *removal* choices that the tool automatically makes using the constraint selectors after the default placement. The results for this experiment are shown in Table 2 with the number of removal choices (REMOVE) and hoisting choices (HOIST) for each experiment. For example, there were 237 removal choices and 55 hoisting choices available to the programmer after the default placement for X Server. After applying the proposed placement approach using the MLS constraint selectors, there are 113 removal and 10 hoisting choices remaining from which the programmer has to select from. This implies that using the MLS constraint selectors has reduced the number of choices that the programmer has to make to produce their desired placement. Making some set of hoisting and removal choices may expose additional choices due to newly introduced dominance and branch relationships. Therefore the number of choices shown in this table is not a measure of the total remaining programmer effort but only of the next set of choices available to the programmer.

We ran our experiments on four programs:

X Server 1.13. Our results show that we are able to reduce the amount of programmer effort by 58%. The number of hooks generated is reduced by 30% (reducing the gap between manual and automated placements by 58% as well).

Postgres 9.1.9. This version has mandatory access control hooks [1] hooks, but these are incomplete according to the documentation of the module [1]. Therefore, we only consider discretionary access control hooks. Our experiments show that we are able to

reduce the amount of programmer effort by 32%. The number of hooks generated is reduced by 15% (reducing the gap between manual and automated placements by 46%).
Linux kernel 2.6.38.8 virtual file system (VFS). The VFS allows clients to access different file systems in a uniform manner. The Linux VFS has been retrofitted with mandatory access control hooks in addition to the discretionary hooks. Our results show that there is an 17% reduction in programmer effort and four fewer hooks then default placement.
Memcached. This general-purpose distributed memory caching system does not currently have any hooks. Our experiments show that constraint selectors are able to reduce the amount of programmer effort by 33% and the number of hooks by 22%.

In the case of the Linux VFS, 'MLS' does not make a significant dent on the removal choices. All the removal choices in Linux VFS fell into one of three categories. First, 24 of the 49 remaining choices are interprocedural hook dominance relationships where the security-sensitive objects being guarded by the hooks were of different data types. For example, the hook in function `do_rmdir` dominates the hook in `vfs_rmdir`, therefore there is a removal opportunity. But the hook in the former mediates an object of type `nameidata` and the latter mediates an object of type `struct dentry *`. Our approach currently only performs removal when the hooks mediate the same variable. Second, 19 choices are interprocedural hook dominance relationships where the object mediated is of the same type but because it is across procedure boundaries and our approach does not employ alias analysis, it conservatively assumes that they are different objects. Finally, six choices were due to intraprocedural hooks on the same object but one mediates reads and the other mediates writes. The 'MLS' constraint selector forbids removal in these cases.

6 Conclusion

In this paper we have successfully demonstrated that our automated system can generate minimal authorization hook placements that satisfy complete mediation and least privilege guided by authorization constraints. We show that using static and dynamic analysis techniques to help programmers select these authorization constraints. We show how our technique can be practically used by programmers to reduce the manual effort in weighing different authorization hook placement options.

Acknowledgement. We thank the anonymous reviewers of this paper and the shepherd Dr. Christian Hammer for their valuable feedback in preparing this manuscript. This material is based upon work supported by the National Science Foundation Grant No. CNS-1408880 and in part by DARPA under agreement number N66001-13-2-4040. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. F.38. sepgsql. `http://www.postgresql.org/docs/9.1/static/sepgsql.html`, 2013.
2. J. P. Anderson. Computer security technology planning study, volume II. Technical Report ESD-TR-73-51, HQ Electronics Systems Division (AFSC), October 1972.

3. D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, HQ Electronic Systems Division (AFSC), March 1976.
4. J. Carter. Using GConf as an Example of How to Create an Userspace Object Manager. *2007 SELinux Symposium*, 2007.
5. D.Walsh. Selinux/apache. `http://fedoraproject.org/wiki/SELinux/apache`.
6. A. Edwards, T. Jaeger, and X. Zhang. Runtime verification of authorization hook placement for the Linux security modules framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 225–234, 2002.
7. V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux Security Modules framework. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 330–339, Nov. 2005.
8. V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 214–229, May 2006.
9. V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, May 2007.
10. L. Gong and R. Schemers. Implementing protection domains in the java$^{\text{tm}}$ development kit 1.2. In *NDSS*, 1998.
11. R. Love. Get on the D-BUS. `http://www.linuxjournal.com/article/7744`, Jan. 2005.
12. Multilevel security in the department of defense: The basics. `http://nsi.org/Library/Compsec/sec0.html`, 1995.
13. D. Muthukumaran, T. Jaeger, and V. Ganapathy. Leveraging "choice" to automate authorization hook placement. In *CCS'12: Proceedings of the 19th ACM Conference on Computer and Communications Security*, page TBD. ACM Press, October 2012.
14. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction, 11th International Conference, CC 2002*, pages 213–228. Springer, 2002.
15. J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. Adsafety: type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 12–12. USENIX Association, 2011.
16. SE-PostgreSQL? `http://archives.postgresql.org/message-id/20090718160600.GE5172@fetter.org`, 2009.
17. S. Son, K. S. McKinley, and V. Shmatikov. 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*, OOPSLA '11, pages 1069–1084. ACM, 2011.
18. F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 11–11. USENIX Association, 2011.
19. L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. Autoises: automatically inferring security specifications and detecting violations. In *Proceedings of the 17th conference on Security symposium*, pages 379–394. USENIX Association, 2008.
20. Implement keyboard and event security in X using XACE. `https://dev.laptop.org/ticket/260`, 2006.
21. Implement keyboard and event security in X using XACE. `https://dev.laptop.org/ticket/260`, 2006.

22. Xorg-Server Announcement. http://lists.x.org/archives/xorg-announce/2008-March/000458.html, 2008.
23. X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, August 2002.

A Hook Placement Properties

First, we want to prove that the authorization hook placement mechanism satisfies two goals: *least privilege* enforcement and *complete mediation*. We start by showing that our initial placement satisfies these properties and the subsequent hoisting and removal phases preserve this property. Least privilege is defined as follows:

Definition 3 *In a least privilege placement a hook (O_h, l_h) placed at location l_h authorizing a set of SSOs O_h implies that for each $o_i \in O_h$, on each path in the program originating from l_h, there must be an operation instance (o_j, l_j) such that $(o_i \, S \, o_j) \vee (o_i \, Q \, o_j)$.*

Complete mediation is defined as:

Definition 4 *Complete Mediation requires that for every operation instance (o_i, l_i), there exists a hook (O_h, l_h) such that l_h control flow dominates l_i and there exists $o_h \in O_h$ such that $(o_i \, Q \, o_h) \vee (o_h \, S \, o_i)$.*

Our approach depends on two inputs a) The set of all SSOs in the program b) The authorization constraints that determine all possible optimizations (hoisting and removal) in hook placement. Our proof assumes that both of these specifications are complete.

Our approach starts by placing a hook at every instance of every SSO in the program. First, it is trivial to show that this results in a placement that adheres to complete mediation since every SSO instance has a corresponding hook. Second, this approach guarantees least privilege since every hook is post-dominated by the SSO instance for which it was placed.

Hoisting. The hoisting in Algorithm 1 hoists the hooks pertaining to equivalent SSOs on all branches of a control statement in a bottom-up fashion in the CDG. The hoisting operation introduces a new hook which dominates the control statement. This new hook preserves least privilege since all the branches of the control statement (therefore all paths originating from the new hook) must contain instances of operations that are equivalent to the one mediated by the new hook. This stage does not remove any hooks so complete mediation is preserved.

Removal. The redundancy removal stage in Algorithm 2 propagates information about hooks placed in a top-down fashion in the CDG. The removal operation does nothing to violate least privilege since it does not add additional hooks. 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 h placed at node n is subsumed by or equivalent to any hook in the set of propagated hooks, then h can be safely removed without violating the complete-mediation guarantee.

Additionally given sound and complete alias analysis we can also guarantee a minimality in hook placement (constrained by complete mediation and least privilege). The

construction of our algorithm guarantees that both hoisting and removal at each node are performed transitively in the context of all successors and predecessors respectively. Therefore, using an oracle-based argument similar to the proof in [13] we can show that with respect to a given set of authorization constraints after using our technique to remove hooks, no additional hoisting or removal can be performed, resulting in a minimal placement.