Automatic Placement of Authorization Hooks in the Linux Security Modules Framework

Vinod Ganapathy
University of Wisconsin
vg@cs.wisc.edu

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

Somesh Jha
University of Wisconsin
jha@cs.wisc.edu

ABSTRACT

We present a technique for automatic placement of authorization hooks, and apply it to the Linux security modules (LSM) framework. LSM is a generic framework which allows diverse authorization policies to be enforced by the Linux kernel. It consists of a kernel module which encapsulates an authorization policy, and *hooks* into the kernel module placed at appropriate locations in the Linux kernel. The kernel enforces the authorization policy using hook calls. In current practice, hooks are placed manually in the kernel. This approach is tedious, and as prior work has shown, is prone to security holes.

Our technique uses static analysis of the Linux kernel and the kernel module to automate hook placement. Given a non-hook-placed version of the Linux kernel, and a kernel module that implements an authorization policy, our technique infers the set of operations authorized by each hook, and the set of operations performed by each function in the kernel. It uses this information to infer the set of hooks that must guard each kernel function. We describe the design and implementation of a prototype tool called TAHOE (Tool for Authorization Hook Placement) that uses this technique. We demonstrate the effectiveness of TAHOE by using it with the LSM implementation of security-enhanced Linux (SELinux). While our exposition in this paper focuses on hook placement for LSM, our technique can be used to place hooks in other LSM-like architectures as well.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*

General Terms: Algorithms, Security

Keywords: Hook placement, static analysis, LSM, SELinux

1. INTRODUCTION

The Linux security modules (LSM) framework [22] is a generic framework which allows diverse authorization policies to be enforced by the Linux kernel. Its goal is to ensure that security-sensitive operations are only performed by users who are authorized to do so. It consists of a reference monitor [1], which encapsulates an authorization policy to be enforced, and *hooks*, which define the interface that the reference monitor presents. Calls to these hooks are placed at several locations within the Linux kernel, so that they mediate security-sensitive operations performed by the kernel. Each hook call results in an authorization query to the reference monitor. The kernel performs the security-sensitive operation only if the authorization query succeeds.

Figure 1 shows a hook placed in the Linux kernel in the LSM implementation of security-enhanced Linux (SELinux) [15], a popular mandatory access control (MAC) based [16] authorization policy. The security-sensitive operation, directory removal, is accomplished by the function call `dir->i_op->rmdir` on line (V5). The hook placed on line (V3), `selinux_inode_rmdir`, checks that the process that requests the directory removal is authorized to do so by the SELinux policy; directory removal succeeds only if the hook call succeeds, in this case, by returning 0. Observe that it is crucial that the hook be placed at line (V3); in the absence of this hook, directory removal will succeed even for processes that are not authorized to do so by the SELinux policy.[1]

```
(V1)int vfs_rmdir(struct inode *dir,
                  struct dentry *dentry) {
(V2)    ...
(V3)    err=selinux_inode_rmdir(dir,dentry);
(V4)    if (!err) { ...
(V5)       dir->i_op->rmdir(dir,dentry);
(V6)    }...}
```

Figure 1: A hook placed in the Linux kernel in the LSM implementation of SELinux.

The architecture of LSM ensures a clean separation between the kernel and the policy-specific reference monitor code, which is implemented as a loadable kernel module. It also offers the advantage of being modular and extensible: to enforce a new security policy, a developer writes a new reference monitor and ensures that hooks are placed properly within the kernel. Finally, it allows reference monitors implementing different policies to co-exist within the kernel: the kernel enforces a security policy by loading the kernel module that implements the corresponding reference monitor. These features have lead LSM to become the vehicle-of-choice for the implementation of several popular MAC-based authorization policies, such as SELinux and Domain and Type enforcement [2]. It has also been incorporated into the mainstream Linux kernel (version 2.6 onwards).

There is emerging interest to enable LSM-like reference monitoring in user-level applications as well. The reason is that several applications, such as X Windows, web-servers and database-servers, support multiple users at the same time. For example,

Supported by ONR contracts N00014-01-1-0796 and N00014-01-1-0708. This work was done when Trent Jaeger was at IBM Research.

[1]While the code fragment shown in Figure 1 itself is not atomic, the code that invokes `vfs_rmdir` obtains a semaphore that prevents other processes from modifying the resources pointed to by `dir` and `dentry`. This ensures that the security-sensitive operation `dir->i_op->rmdir` is performed on the same resources which were authorized by `selinux_inode_rmdir`.

an X server allows multiple users, possibly with different security-levels, to display clients simultaneously. Hooks placed in the operating system are often insufficient to enforce authorization policies at the application-level. For example, the policy that "a cut-and-paste operation from a high-security client to a low-security client is disallowed" is better enforced by the X server than the operating system, because the operating system is unaware of the cut-and-paste operation, which is specific to X Windows. In fact, efforts are underway [12] to secure X Windows by placing hooks to an LSM-like reference monitor within the X server. The recent release of a policy management infrastructure for user-level processes [21] is intended to enable the development of reference monitors for any application that would benefit. We also note that Java's security architecture is akin to LSM, where calls are placed to check access permissions to security-sensitive objects [7]. Thus, enforcement of authorization policies by placing reference monitor hooks in user-level applications is becoming common practice.

In current practice, the decision on where to place hooks is often made informally, and hooks are placed manually at locations deemed appropriate in the Linux kernel or user-level application. This process suffers from several drawbacks:

1. *Inadvertent bugs, leading to security holes.* Prior research has shown security holes due to improper hook placement in the Linux kernel. In particular, Zhang *et al.* [23] demonstrate that inadequate placement of hooks results in security-sensitive operations being performed without the appropriate authorization query being posed to the reference monitor. Jaeger *et al.* [10] also demonstrate similar bugs by comparing the consistency of hook placements along different paths in the kernel. These bugs lead to potentially exploitable security holes.
2. *Inextensibility to new security policies.* Manual reasoning is needed to place hooks for each new security policy developed. The Linux-2.6 kernel somewhat ameliorates the effort needed by placing hooks to a dummy reference monitor at pre-defined locations in the kernel. The idea is that developers can tailor the code for hooks to suit specific security policies. However, this approach is still problematic. First, care is required to ensure that each hook call indeed authorizes the security-sensitive operations that its pre-defined placement intends to. Second, it is fairly common practice to add new hooks to implement security policies for which pre-defined hook placement does not suffice. Manual reasoning is required to determine placement points for each new hook.
3. *Inextensibility to emerging applications.* As mentioned earlier, recent proposals [12, 21] for developing MAC authorization policies for user-level applications are also based upon LSM-like architectures. As with LSM, these proposals require manual effort to determine hook placement.

While static and runtime verification techniques [10, 23] have been proposed to solve the first problem mentioned above, they do not solve the second and third problems.

In this paper, we demonstrate *a technique for automatic placement of authorization hooks* (i.e., hooks to a reference monitor that encapsulates an authorization policy). While our exposition and evaluation in this paper is restricted to placement of LSM authorization hooks in the kernel, the concepts we present extend naturally to any system/reference monitor pair that conforms to an LSM-like architecture. Our technique requires two inputs: the Linux kernel, and the reference monitor (i.e., the kernel module that implements it) which contains the source code for authorization hooks. It analyzes them and identifies locations in the kernel where hooks must be placed so that security-sensitive operations are authorized correctly.

The key idea behind our technique is to leverage semantic information embedded in the source code of the hooks and the Linux kernel. It uses static analysis to determine the set of operations authorized by each hook. A similar analysis on the kernel-side determines the set of operations performed by each kernel function. The results of hook analysis and kernel analysis are then merged to construct an *authorization graph*. An authorization graph relates each kernel function to the set of hooks that must protect it. With the authorization graph in hand, hook placement is straightforward: at each location in the kernel where a kernel function is called, insert hooks (as determined by the authorization graph) that must protect the function. This technique addresses all the problems discussed above. First, because we use the set of operations performed by a kernel function to obtain the set of hooks that must guard it, we ensure correctness by construction. Second, because our analysis is general-purpose and analyzes both hook and kernel code, it extends easily to new security policies and emerging applications alike.

Contributions: In summary, the main contribution of this paper is a technique for automatic placement of authorization hooks in the Linux kernel. We present the design and implementation of a prototype tool called TAHOE that uses this technique. We demonstrate the efficacy of our technique by using TAHOE with the implementation of hooks from the LSM implementation of SELinux. In particular, we show how TAHOE precisely recovers the set of operations authorized by each hook from the above implementation, and the set of operations authorized by the Linux kernel. It uses this information to place hooks in the Linux kernel by constructing the authorization graph. We evaluate the hook placement that TAHOE generates by comparing it against the existing hook placement in the LSM implementation of SELinux.

Paper Organization: In the following section, we introduce some concepts used throughout the paper. We then present the algorithms used by TAHOE in Section 3, and discuss our experience with TAHOE in Section 4. We review related research in Section 5 and conclude in Section 6.

2. CONCEPTUAL OPERATIONS

The goal of the LSM framework is to ensure that security-sensitive operations on resources are only performed by entities who are authorized to do so. It achieves this by placing hooks, which pose authorization queries to a reference monitor, before kernel functions that perform such security-sensitive operations. For instance, in Figure 1, the security sensitive operation being authorized is directory removal, and the resources affected by this operation are the inodes of the directory being removed, and the directory from which it is being removed.

Because TAHOE seeks to place hooks, it works with the source code of a kernel module containing the source code of authorization hooks, such as the kernel module implementing SELinux hooks, and a non-hook-placed version of the kernel. As discussed earlier, TAHOE analyzes each of these inputs independently, and correlates the results to determine hook placement. Observe that security-sensitive operations are a unifying theme of both inputs—a hook authorizes security-sensitive operations, and the kernel performs them. Thus, to combine the results of hook analysis and kernel analysis, it suffices to determine the security-sensitive operations authorized by each hook, and the security-sensitive operations performed by each kernel function. We use the term *conceptual operations* to refer to such security-sensitive operations.

The analyses described in the rest of this paper assume that the set of conceptual operations is known. For the analyses described here, we used the set of conceptual operations used by the LSM implementations of popular MAC policies, including SELinux and Domain and Type Enforcement [2]. This set (of size 504) is fairly comprehensive, and includes generic operations on resources, such as reading from, writing to, or executing a file. We expect that this set will find use in the analysis of other LSM-like architectures as well. Conceptual operations are depicted in the rest of this paper using suggestive names, such as FILE__WRITE, FILE__READ and FILE__EXECUTE, corresponding, respectively to writing to, reading from and executing a file. We note that the analyses used by TAHOE are parameterized by the set of conceptual operations, and more conceptual operations can be added as the need arises. Changes to the set of conceptual operations does not change any of the algorithms that we present in the paper.

3. AUTHORIZATION HOOK PLACEMENT USING STATIC ANALYSIS

Figure 2 shows the architecture of TAHOE. It analyzes the source code of the Linux kernel (with no hooks placed), and the kernel module containing source code of hooks, and outputs a hook-placed kernel. To do so, it combines the results of hook analysis and kernel analysis to produce the authorization graph, which relates each kernel function to the set of hooks that must guard it—each kernel function must be guarded by a set of hooks that authorize the conceptual operations it performs.

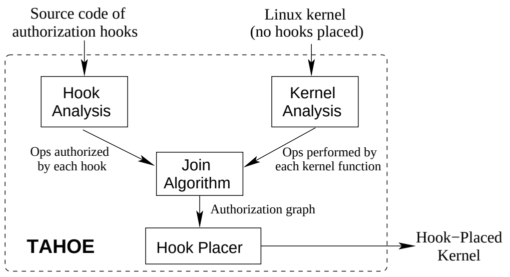

Source code of authorization hooks
Linux kernel (no hooks placed)
Hook Analysis
Kernel Analysis
Ops authorized by each hook
Ops performed by each kernel function
Join Algorithm
Authorization graph
TAHOE
Hook Placer
Hook−Placed Kernel

Figure 2: Overall architecture of TAHOE.

An example best demonstrates TAHOE's analysis. Figure 3(A) shows a snippet of kernel code: `vfs_rmdir`, the virtual file system function to remove a directory. This function accepts two arguments, corresponding to data structures of the directory to be deleted (`dentry`), and the parent of this directory (`dir`). It transitively calls `permission`, which performs a discretionary access control (DAC) [16] check on line (P3) to determine whether the current process has "write" and "execute" permissions on the parent directory. If so, control reaches line (V5), which resolves to the appropriate physical file system call (such as `ext2_rmdir`) to remove the directory.

Figure 3(B) shows a portion of the output for `vfs_rmdir` when TAHOE is used with the kernel module from the LSM implementation of SELinux (henceforth abbreviated to LSM-SELinux). It determines that the call to `dir->i_op->rmdir` (on line (V5)) must be protected with two LSM-SELinux hooks: `selinux_inode_rmdir` and `selinux_inode_permission`, called with MAY_WRITE, while the DAC check on line (P3) must be supplemented with the hook `selinux_inode_permission`, which checks for the corresponding MAC permissions. We now provide a high-level description of the analysis that TAHOE employs.

TAHOE's kernel analysis (Section 3.2) analyzes the file system code and infers that directory removal (`dir->i_op->rmdir`, line (V5)) involves performing the conceptual operations DIR__RMDIR, DIR__WRITE and DIR__SEARCH. Intuitively, this is because a typical file system, such as `ext2` does the following to remove a directory `bar` from a directory `foo`: (i) it finds the entry of `bar` in an appropriate kernel data structure of `foo` (DIR__SEARCH), and (ii) removes the entry of `bar` from this data structure (DIR__RMDIR), which involves writing to the data structure (DIR__WRITE). Note that directory removal is a specialized write: removal of `bar` from `foo` requires removal of the entry of `bar` from `foo`, as opposed to other directory manipulations, such as directory creation, which adds a new entry. Thus, DIR__RMDIR denotes a special kind of directory write, as opposed to DIR__WRITE, which denotes the *generic* write operation.

```
(P1)int permission(struct inode *inode,
                   int mask) {
(P2)  ...
(P3)  inode->i_op->permission(inode,mask);
(P4)  ...}

(M1)int may_delete(struct inode *dir,
      struct dentry *vic, int isdir) {
(M2)  ...
(M3)  // DAC check for WRITE and EXEC
(M4)  permission(dir,MAY_WRITE|MAY_EXEC);
(M5)  ...}

(V1)int vfs_rmdir(struct inode *dir,
                  struct dentry *dentry) {
(V2)  may_delete(dir, dentry, 1);
(V3)  ...
(V4)  // Remove the directory
(V5)  dir->i_op->rmdir(dir, dentry);
(V6)  ...}
```

(A) VFS code from `linux-2.4.21/fs/namei.c` for directory removal. Error checking code has been omitted for brevity.

```
Hooks for inode->i_op->permission on Line (P3):
(H1) selinux_inode_permission(dir,mask)

Hooks for dir->i_op->rmdir on Line (V5):
(H2) selinux_inode_rmdir(dir,dentry)
(H3) selinux_inode_permission(dir, MAY_WRITE)
```

(B) Analysis results of TAHOE (with SELinux hooks) for code fragment shown in (A).

Figure 3: Example to illustrate analysis performed by TAHOE.

TAHOE's hook analysis (Section 3.1) analyzes the source code of SELinux hooks (not shown in Figure 3) and infers that the hook `selinux_inode_permission`, when invoked with MAY_EXEC and MAY_WRITE, checks that the conceptual operations DIR__SEARCH and DIR__WRITE, respectively, are authorized. It also infers that `selinux_inode_rmdir` checks that both the conceptual operations DIR__SEARCH and DIR__RMDIR are authorized.

When TAHOE combines the results of these analyses (Section 3.3) it produces an authorization graph, a portion of which is shown in Figure 3(B). Because `dir->i_op->rmdir` performs DIR__SEARCH and DIR__RMDIR, it is protected by `selinux_inode_rmdir`, which authorizes these operations. In addition, `dir->i_op->rmdir` performs DIR__WRITE, which `selinux_inode_permission` authorizes when invoked with MAY_WRITE. TAHOE also supplements existing DAC checks, such as the one on line (P3), with hooks that perform the corresponding MAC checks, as shown in line (H1) of the output. It is important to note that TAHOE does *not* use existing DAC checks to determine hook placement. Its analysis is based upon conceptual operations performed by each kernel function.

Indeed, hooks are also placed in LSM-SELinux as shown in Figure 3(B), though in the case of LSM-SELinux this placement was determined manually. This validates the results of TAHOE's analysis because the hook placement in LSM-SELinux has been tested thoroughly, and the errors found by verification tools [10, 23] have been fixed. However, in the case of `vfs_rmdir`, LSM-SELinux optimizes hook placement: A closer look at the source code of `vfs_rmdir` reveals that all code paths to line (V5) pass through line (M4) and line (P3) (formally, line (M4) and line (P3) *dominate* line (V5) [17]). Because the hook call on line (H3) is subsumed by the hook call on line (H1), LSM-SELinux only places the hooks shown in line (H1) and line (H2). While TAHOE infers the authorization graph which relates hooks and kernel functions correctly, it currently does not optimize hook placement; we leave optimization for future work. We also note that the security of the LSM framework is determined by the correctness of the authorization graph. The rest of this section describes each of TAHOE's components in greater detail.

3.1 Analysis of Authorization Hooks

To determine the conceptual operations authorized by each hook, TAHOE analyzes the kernel module that contains source code of hooks. In addition to determining the conceptual operations authorized, it also determines the conditions under which these operations are authorized. Consider Figure 4(A), which shows a snippet of the implementation of the hook `selinux_inode_permission` in the LSM-SELinux kernel module. This snippet authorizes searching, writing to, or reading from an inode representing a directory, based upon the value of `mask`. The authorization is performed by the call to `inode_has_perm`, which authorizes a conceptual operation on an inode based upon the *access vector*[2] it is invoked with. In Figure 4(A), the access vector is obtained by a call to `file_mask_to_av`.

```
(S1)int selinux_inode_permission(struct *inode, int mask)
(S2){ if (mask == 0) return 0;
(S3)  return inode_has_perm
           (file_mask_to_av(inode->i_mode,mask),...);
(S4)}

(F1)access_vector_t file_mask_to_av(int mode, int mask)
(F2){ access_vector_t av = 0;
(F3)  if ((mode & S_IFMT) != S_IFDIR) {
(F4)    /* File-related conceptual operations */
(F5)  } else {
(F6)    if (mask & MAY_EXEC) av |= DIR__SEARCH;
(F7)    if (mask & MAY_WRITE) av |= DIR__WRITE;
(F8)    if (mask & MAY_READ) av |= DIR__READ;
(F9)  }
(F10) return av; }
```

(A) Code for the hook `selinux_inode_permission`.

Analysis output (from Algorithm 1) for `selinux_inode_permission`:
- ⟨(mask ≠ 0) ∧ inode_isdir ∧ (mask & MAY_EXEC) ‖ DIR__SEARCH⟩
- ⟨(mask ≠ 0) ∧ inode_isdir ∧ (mask & MAY_WRITE) ‖ DIR__WRITE⟩
- ⟨(mask ≠ 0) ∧ inode_isdir ∧ (mask & MAY_READ) ‖ DIR__READ⟩

where "inode_isdir" denotes (inode->i_mode & S_IFMT == S_IFDIR).

(B) Portion of the output of TAHOE's hook analysis for `selinux_inode_permission`.

Figure 4: Example to illustrate TAHOE's hook analysis.

Figure 4(B) shows a fragment of the output of TAHOE's analysis for this hook. Each line of the output is a tuple of the form ⟨predicate ‖ OPERATION⟩, where the predicate only contains formal parameters of the hook. This tuple is interpreted as follows: if the hook is invoked in a context such that predicate holds, then it checks that the conceptual operation OPERATION is authorized. In this case, TAHOE infers that for inodes that represent directories (i.e., the inodes with (inode->i_mode & S_IFMT == S_IFDIR)) the hook `selinux_inode_permission` checks that the conceptual operations DIR__SEARCH, DIR__WRITE or DIR__READ are authorized, based upon the value of `mask`. We now describe the hook analysis algorithm used by TAHOE.

[2]Conceptual operations in LSM-SELinux are represented using bit-vectors, called access vectors. Because we derived the set of conceptual operations used by TAHOE by examining LSM implementations of popular MAC policies, including LSM-SELinux, there is a one-to-one mapping between the conceptual operations used by TAHOE and the access vectors in LSM-SELinux.

3.1.1 *The Hook Analysis Algorithm*

The algorithm to analyze the kernel module containing source code of hooks is shown in Algorithm 1. For ease of explanation, assume that there is no recursion; we explain how we deal with recursion later in the section. The analysis proceeds by first constructing the *call-graph* [17] of the kernel module. A call-graph captures caller-callee relationships. Each node of the call-graph is a function in the kernel module; an edge $f \longrightarrow g$ is drawn if function f calls function g. The call-graph is processed bottom-up, starting at the leaves, and proceeding upwards. For each node in the call-graph, it produces a *summary* [19], and outputs summaries of hooks.

```
Algorithm : ANALYZE_MODULE($M$, $H$)
Input     : (i) $M$: Module containing source code of hooks, (ii) $H$: A set
            containing the names of hooks.
Output    : For each $h \in H$, a set {⟨predicate ‖ CONCEPTUAL-OP⟩},
            denoting the conceptual operations authorized by each hook,
            and the conditions under which they are authorized.
1 Construct the call-graph $G$ of the module $M$
2 $L$ := List of vertices of $G$, reverse topologically sorted
3 foreach ($f \in L$) do
4     Summary($f$) := ANALYZE_FUNCTION($f$, Entrypoint($f$), true)
5 foreach ($h \in H$) do
6     Output Summary($h$)
```

Algorithm 1: TAHOE's algorithm for hook analysis.

Summary construction is described in Algorithm 2. The summary of a function f is a set of pairs ⟨pred ‖ OP⟩, denoting the condition (pred) under which a conceptual operation (OP) is authorized by f. The analysis in Algorithm 2 is *flow*- and *context-sensitive*. That is, it respects the control-flow of each function, and precisely models call-return semantics. Intuitively, summary construction for a function f proceeds by propagating a predicate p though the statements of f. At any statement, the predicate denotes the condition under which control-flow reaches the statement. The analysis begins at the first statement of the function f (denoted by Entrypoint(f)), with the predicate set to true.

At an `if-(q)-then-else` statement, the true branch is analyzed with the predicate p ∧ q, and the false branch is analyzed with the predicate p ∧ ¬q. For instance, the value of p at line (F3) in Figure 4(A) is true. Thus, lines (F6)-(F8) are analyzed with true ∧ (mode & S_IFMT) == S_IFDIR. At `Call` $g(a_1, a_2, \ldots, a_n)$, a call to the function g, the summary of g is "specialized" to the calling-context. Note that because of the order in which functions are processed in Algorithm 1, the summary of g is computed before f is processed. The summary of g is a set of tuples ⟨q$_i$ ‖ OP$_i$⟩. Because of the way summaries are computed, formal parameters of g appear in the predicate q$_i$. To specialize the summary of g, actual parameters a_1, a_2, ..., a_n are substituted in place of formal parameters in q$_i$. The resulting predicate r$_i$ is then combined with p, and the entry ⟨p ∧ r$_i$ ‖ OP$_i$⟩ is included in the summary of f. Intuitively, g authorizes operation OP$_i$ if the predicate q$_i$ is satisfied. By substituting actual parameters in place of formal parameters, we determine whether *this* call to g authorizes operation OP$_i$; i.e., whether the predicate q$_i$, specialized to the calling context, is satisfiable. Because the call to g is reached in f under the condition p, an operation is authorized by g only if p ∧ r$_i$ is satisfiable.

For other statements, the analysis determines whether the statement potentially authorizes an operation OP. Determining whether a statement authorizes an operation OP is specific to the way conceptual operations are represented in the kernel module. For instance, in LSM-SELinux, conceptual operations are denoted by bit-vectors, called access vectors (of type `access_vector_t`), and there is a one-to-one mapping between access vectors and concep-

```
Algorithm : ANALYZE_FUNCTION($f$, $s$, p)
Input     : (i) $f$: Function name, (ii) $s$: Statement in $f$ from which to start
            the analysis, (iii) p: A Boolean predicate.
Output    : A set $\{\langle$predicate $\|$ CONCEPTUAL-OP$\rangle\}$.
1  $R := \phi$
2  switch TYPE-OF($s$) do
3      case if (q) then $B_{true}$ else $B_{false}$
4          $R$ := ANALYZE_FUNCTION($f$, Entrypoint($B_{true}$), p $\wedge$ q)
5              $\cup$ ANALYZE_FUNCTION($f$, Entrypoint($B_{false}$), p $\wedge \neg$q)
6      case Call $g(a_1, a_2, \ldots, a_n)$
7          $G$ := Summary($g$)
8          foreach ($\langle$q$_i$ $\|$ OP$_i\rangle \in G$) do
9              r$_i$ := q$_i$ specialized with $a_1, a_2, \ldots, a_n$
10             $R := R \cup \{\langle$(p $\wedge$ r$_i$) $\|$ OP$_i\rangle\}$
11         $R := R \cup$ ANALYZE_FUNCTION($f$, ControlFlowSucc($f$, $s$), p)
12     otherwise
13         if ($s$ authorizes conceptual operation OP) then $R := \{\langle$p $\|$ OP$\rangle\}$
14         Update p appropriately
15         $R := R \cup$ ANALYZE_FUNCTION($f$, ControlFlowSucc($f$, $s$), p)
16 foreach ($\langle$p $\|$ OP$\rangle \in R$) do
17     Existentially quantify-out any local variables of $f$ appearing in p
18 return $R$
```

Algorithm 2: Producing the summary of a function.

tual operations. Thus, for LSM-SELinux we use the occurrence of an access vector (e.g., reading its value) in a statement to determine if the statement authorizes a conceptual operation.

Where possible, the predicate p is also updated appropriately based upon the action of statement s. For instance, if the statement in question is `j := i`, and predicate p propagated to this statement is (i == 3), then the predicate p is updated to (j == i) $\wedge$ (i == 3). In cases where the effect of s on p cannot be determined, the new value of p is set to Unknown, a special value denoting that the value of p cannot be determined precisely.

For functions with a formal parameter of type `access_vector_t`, but do not refer to any particular access vector (such as DIR__READ, DIR__WRITE, or DIR__SEARCH), the analysis returns $\{\langle$true $\| \lambda x.x\rangle\}$ (not shown in Algorithm 2 for brevity), which says that the function potentially authorizes any conceptual operation, based upon the access vector it is invoked with (the variable x in $\lambda x.x$ denotes the access vector).

After processing a statement s in f, the analysis continues by processing the control-flow-successors of s. The analysis terminates when all the statements reachable from Entrypoint(f) have been analyzed. To keep the analysis tractable, Algorithm 2 analyzes loop bodies exactly once. That is, it ignores back-edges of loops. As a result, loops are treated conceptually equivalent to `if-then-else` statements.

Finally, any local variables of f appearing in predicates p (for each $\langle$p $\|$ OP$\rangle$ in the summary of f) are quantified-out. As a result, predicates appearing in the summary of f only contain formal parameters of f.

We illustrate Algorithm 1 using Figure 4(A). For the function `file_mask_to_av`, Algorithm 2 returns the output:

$\langle$mode_isdir $\wedge$ (mask & MAY_EXEC) $\|$ DIR__SEARCH$\rangle$
$\langle$mode_isdir $\wedge$ (mask & MAY_WRITE) $\|$ DIR__WRITE$\rangle$
$\langle$mode_isdir $\wedge$ (mask & MAY_READ) $\|$ DIR__READ$\rangle$

where 'mode_isdir' denotes 'mode & S_IFMT == S_IFDIR'.

Observe that the summary only contains formal parameters of `file_mask_to_av`. When this summary is specialized to the call on line (S3), formal parameters are replaced with the actual parameters (e.g., `mode` by `inode->i_mode`), thus specializing the summary to the call-site, producing:

$\langle$inode_isdir $\wedge$ (mask & MAY_EXEC) $\|$ DIR__SEARCH$\rangle$
$\langle$inode_isdir $\wedge$ (mask & MAY_WRITE) $\|$ DIR__WRITE$\rangle$
$\langle$inode_isdir $\wedge$ (mask & MAY_READ) $\|$ DIR__READ$\rangle$

where 'inode_isdir' denotes 'inode->i_mode & S_IFMT == S_IFDIR'.

For `inode_has_perm`, Algorithm 2 returns $\{\langle$true $\| \lambda x.x\rangle\}$, which intuitively means that the function authorizes a conceptual operation based upon the access vector (x) passed to it. Thus, when this call to `inode_has_perm` is specialized to the call on line (S3), the summary obtained is the same shown above. Because line (S3) in `selinux_inode_permission` is reached when (mask $\neq$ 0), this predicate is combined with predicates in the summary of the function `inode_has_perm` to produce the result shown in Figure 4(B).

Handling Recursion: Recursion in the kernel module introduces strongly-connected components in its call-graph. Note that Algorithm 1 requires the call-graph to be a directed acyclic graph (DAG). To handle recursion, we consider the functions in a strongly-connected component together. That is, we produce a consolidated summary for each strongly-connected component. Intuitively, this summary is the set of conceptual operations (and the associated conditions) that could potentially be authorized if *any* function in the strongly-connected component is called. Observe that handling recursion also requires a small change to lines (7)-(11) of Algorithm 2. Because of recursion, the summary of a function g that is called by a function f may no longer be available in line (7), in which case we skip forward to line (11).

3.1.2 Precision of Hook Analysis

Observe that Algorithm 2 analyzes all reachable statements of each function. Thus, if a function f authorizes operation OP, then $\langle$q $\|$ OP$\rangle \in$ Summary(f), for some predicate q. However, because of the approximations employed by Algorithm 1 and Algorithm 2 to keep the analysis tractable, the predicate q may not accurately describe the condition under which OP is authorized.

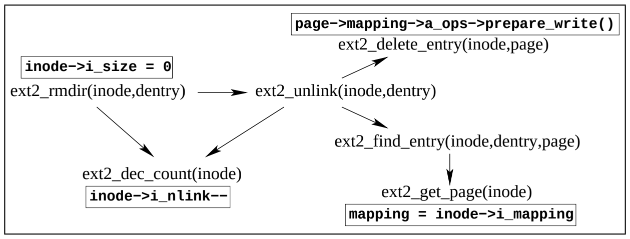
If a kernel module M is recursion-free, all functions in M are loop-free, and updates to predicates can be determined precisely (i.e., predicates are not set to Unknown), then Algorithm 2 propagates predicates precisely. That is, the predicate at statement s is p if and only if s is reached under condition p.

Because Algorithm 2 ignores back-edges on loops, loop bodies are analyzed exactly once, and the predicates retrieved will be imprecise. Similarly, because Algorithm 2 employs a heuristic to handle recursion, the predicates retrieved will be imprecise. These predicates are used during hook placement to determine the arguments that the hook must be invoked with. Thus, imprecision in the results of the analysis will mean manual intervention to determine how hooks must be invoked. Fortunately, the code of hooks, even in complex kernel modules such as LSM-SELinux is relatively simple, and we were able to retrieve the conditions precisely in most cases. For instance, there were no loops in any of the hooks from LSM-SELinux that we analyzed.

3.2 Analysis of the Linux kernel

TAHOE's kernel analysis complements its hook analysis by determining the set of conceptual operations performed by each function in the kernel. For instance, TAHOE's kernel analysis infers that `vfs_rmdir`, the virtual file system function for directory removal (Figure 3(A)), performs the conceptual operations DIR__RMDIR, DIR__SEARCH, and DIR__WRITE, corresponding, respectively, to removing, searching within, and writing to a directory. Observe that line (V5) of `vfs_rmdir` is a call through a function pointer. Its targets are physical file system-specific functions for directory removal, such as `ext2_rmdir` in the `ext2` file system.

Figure 5(A) shows a portion of the call-graph of `ext2_rmdir`. Note that the functions shown in the call-graph (`ext2_unlink`, `ext2_dec_count`, etc.) can also be called by other functions in the kernel; these edges are not shown in Figure 5(A). When a request is received to remove directory `bar` from directory `foo`, `ext2_rmdir` checks to see that `bar` is empty via a call to `ext2_rmdir_empty` (not shown in Figure 5(A)). It then calls `ext2_unlink`, which modifies `ext2`-specific data structures and removes the entry of `bar` from the inode of `foo`. Finally, it calls `ext2_dec_count` to decrement the field `i_nlink` on the inodes of both `foo` and `bar`. Figure 5(B) shows the relevant snippet of TAHOE's analysis on the Linux kernel. It infers that `ext2_rmdir` performs the conceptual operations DIR__RMDIR, DIR__SEARCH and DIR__WRITE. Because `ext2_rmdir` is pointed to by `dir->i_op->rmdir` (as determined by pointer analysis algorithms employed by CIL [18], the tool that TAHOE is built upon), it is called indirectly from `vfs_rmdir`, and TAHOE infers that `vfs_rmdir` performs these conceptual operations as well. We now examine the analysis in detail.


page->mapping->a_ops->prepare_write()
ext2_delete_entry(inode,page)
inode->i_size = 0
ext2_rmdir(inode,dentry)
ext2_unlink(inode,dentry)
ext2_find_entry(inode,dentry,page)
ext2_dec_count(inode)
inode->i_nlink--
ext2_get_page(inode)
mapping = inode->i_mapping

(A) Portion of the call-graph of the Linux kernel, showing `ext2_rmdir`. Relevant code snippets from each function are shown in boxes.

`ext2_delete_entry`:	DIR__WRITE
`ext2_get_page`:	DIR__SEARCH
`ext2_find_entry`:	DIR__SEARCH
`ext2_dec_count`:	FILE__UNLINK
`ext2_unlink`:	FILE__UNLINK, DIR__WRITE, DIR__SEARCH
`ext2_rmdir`:	DIR__RMDIR, DIR__WRITE, DIR__SEARCH

(B) Portion of kernel analysis results relevant to (A).

Figure 5: Example to illustrate TAHOE's kernel analysis.

3.2.1 The Kernel Analysis Algorithm

Like hook analysis, TAHOE's kernel analysis recovers the set of conceptual operations performed by each kernel function. However, unlike hook analysis, it does not recover the conditions under which a conceptual operation is performed. Several hooks, including `selinux_inode_permission` (Figure 4(A)), authorize different conceptual operations based upon the arguments they are invoked with. Consequently, the conditions recovered by hook analysis can be used to infer arguments during hook placement. On the other hand, for kernel functions, we only need to infer if *there exist* arguments such that the kernel function performs a conceptual operation. For instance, suppose `ext2_rmdir` is invoked in response to a request to remove directory `bar` from directory `foo`. As mentioned earlier, `ext2_rmdir` first checks to see that `bar` is empty; if not, directory removal fails, and `bar` is not removed. However, it is important to note that if `bar` was empty (and certain other conditions satisfied), then it would have been removed. That is, there exists *some* argument such that `ext2_rmdir` performs the conceptual operations shown in Figure 5(B). Thus, it suffices to recover the conceptual operations performed by a kernel function irrespective of the conditions under which they are performed.

The kernel analysis algorithm is shown in Algorithm 3. It processes the call-graph of the kernel in a bottom-up fashion, analyzing a function after all its callees have been analyzed. Recursion, which leads to strongly-connected components in the call-graph, is dealt with by consolidating the results for each strongly-connected component, as described in Section 3.1.1 for hook analysis.

```
Algorithm : ANALYZE_KERNEL
Input     : Linux kernel.
Output    : For each function in the kernel, a set {OP} of operations that it
            may perform.
1 Construct the call-graph $G$ of the kernel
2 $L$ := List of vertices of $G$, reverse topologically sorted
3 foreach $f \in L$ do
4     $R$ := ANALYZE_KERNEL_FUNCTION($f$)
5     foreach $g$ such that $f$ calls $g$ do
6         $R := R \cup$ CodePatterns($g$)
7     CodePatterns($f$) := $R$
8     OPS := SEARCH_IDIOMS($R$)   /* Described in Section 3.2.2 */
9     KernelSummary($f$) := OPS
```

Algorithm 3: TAHOE's algorithm for kernel analysis.

Informally, Algorithm 3 searches for combinations of *code patterns* in each kernel function. It then searches through a set of *idioms* (on line 8) for these code-patterns to determine if the function performs a conceptual operation. An idiom is a rule that relates a combination of code-patterns to conceptual operations. Intuitively, these code-patterns correspond to manipulations of kernel data structures that typically happen when a conceptual operation is performed by the kernel. For instance, removal of a directory `bar` from `foo` (conceptual operation DIR__RMDIR) usually involves setting the field `i_size` of the inode of `bar` to 0, and decrementing the field `i_nlinks` of the inodes corresponding to `bar` and `foo`. Similarly, reading from a directory (conceptual operation DIR__READ) usually involves modifying its access time (field `i_atime` of the inode). We describe the expressive power of, and the methodology used to write idioms in Section 3.2.2.

```
Algorithm : ANALYZE_KERNEL_FUNCTION($f$)
Input     : $f$: A kernel function.
Output    : A set of code patterns that appear in $f$.
1 $R := \phi$
2 foreach statement $s$ of $f$ do
3     if ($s$ matches an entry $\mathcal{P}$ in IdiomCodePatterns) then $R := R \cup \{\mathcal{P}\}$
4 return $R$
```

Algorithm 4: Searching for code patterns that appear in idioms.

Algorithm 3 first gathers the set of code patterns that appear in its body, as well as those that appear in its callees. Code patterns are gathered as described in Algorithm 4, which scans the code of a kernel function, and searches for code-patterns from the set IdiomCodePatterns. This set contains code-patterns that appear in the idioms used by TAHOE. Algorithm 3 then searches through the set of idioms (line 8) to determine the set of operations that are potentially performed by the kernel function.

For instance, consider Figure 5(A): the lines `inode->i_size = 0` and `inode->i_nlink--` appear in the functions `ext2_rmdir` and `ext2_dec_count`, respectively. As Figure 6(B) shows, one of the idioms TAHOE uses is "DIR__RMDIR `:-` SET `inode->i_size` TO 0 $\wedge$ DECR `inode->i_nlink`". Both these code patterns appear in the set CodePatterns(`ext2_rmdir`) after line (7) on Algorithm 3 when `ext2_rmdir` is processed. Because these patterns also appear in the idiom above, the operation DIR__RMDIR is added to the set OPS on line (8), and consequently to KernelSummary(`ext2_rmdir`), which denotes the set of conceptual operations performed by `ext2_rmdir`.

Observe that code patterns that appear in an idiom can be drawn from several functions. This is because several common tasks in the kernel are often delegated to helper functions. Consequently several idioms used by TAHOE contain code patterns drawn from different functions. For instance, while `inode->i_size = 0` appears in `ext2_rmdir`, decrementing `inode->i_nlink` is delegated to `ext2_dec_count`. Thus, it is important to search through the set of idioms *after* code patterns are gathered from all the callees of the function being analyzed.

Algorithm 4 scans the code of a function in a *flow-insensitive* fashion, i.e., it does not consider control-flow while scanning the statements of the function. This suffices for kernel analysis because, as mentioned earlier, the analysis does not track the conditions under which an operation is performed. Instead, it returns the set of conceptual operations that *may* be performed by the kernel. We use the occurrence of an idiom in the function body to determine if an operation is performed by the kernel, and this can be achieved using a simple flow-insensitive scan of the function body.

3.2.2 Idioms

Idioms are rules with conceptual operations on the left-hand-side, and conjunctions of code-patterns on the right-hand-side. Each conceptual operation OP can appear on the left-hand-side of several rules. Figure 6(A) shows the grammar used to express idioms for TAHOE; there are currently six kinds of code-patterns, which we have found sufficient to express idioms for most conceptual operations. Code-patterns are expressed in terms of the abstract-syntax-tree (AST) of a variable, rather than variable names. Figure 6(B) shows a few idioms, relevant to Figure 5.

Idiom :=	OP :- $\bigwedge_{i=1}^{n}$ (CodePat$_i$ | ¬CodePat$_i$)
CodePat :=	SET AST | SET AST TO value
	| READ AST | CALL AST
	| INCR AST | DECR AST
AST :=	(type->)* fieldname

(A) Idiom Grammar.

DIR__WRITE :-	`SET inode->i_ctime` ⋀
	`CALL address_space_ops->prepare_write()`
DIR__SEARCH :-	`READ inode->i_mapping`
FILE__UNLINK :-	`DECR inode->i_nlink` ⋀
	¬`SET inode->i_size TO 0`
DIR__RMDIR :-	`SET inode->i_size TO 0` ⋀
	`DECR inode->i_nlink`

(B) Examples of idioms used in the analysis of `ext2_rmdir`.

Figure 6: Idiom grammar and examples of idioms.

Using Idioms: Idioms are used to determine which conceptual operations are performed by each kernel function. After the set of code-patterns that appears in a function f and its callees is gathered in CodePatterns(f) (line (7) of Algorithm 3), SEARCH_IDIOMS searches through the set of idioms. If the code-patterns that appear on the right-hand-side of an idiom also appear in CodePatterns(f), then SEARCH_IDIOMS adds the left-hand-side of the idiom to the set of conceptual operations performed by f. For instance, because the value of CodePatterns(`ext2_unlink`) is {`DECR inode->unlink`}, and this matches the third idiom in Figure 6(B), FILE__UNLINK is added to KernelSummary(`ext2_unlink`). Note that because CodePatterns(`ext2_rmdir`) contains the pattern "`SET inode->i_size TO 0`", it does *not* match the third idiom, and FILE__UNLINK ∉ KernelSummary(`ext2_rmdir`).

Methodology used to write idioms: We explain the methodology to write idioms by considering two examples from Figure 6(B); More examples of idioms can be found elsewhere [9].

1. DIR__WRITE `:- SET inode->i_ctime` ∧ `CALL address_space_ops->prepare_write()`: Writing to a directory usually involves a statement that adds new content to the data structures that store directory content (achieved via the call to `prepare_write()`), followed by setting the field `i_ctime` of the directory's inode, indicating the change time.
2. DIR__RMDIR `:- SET inode->i_size TO 0` ∧ `DECR inode->i_nlink`: Removing a directory `bar` from a directory `foo` involves decrementing the field `i_nlink`, the link count, of the inodes of both `foo` and `bar`, followed by setting `i_size`, the size of the inode of `bar` to 0.

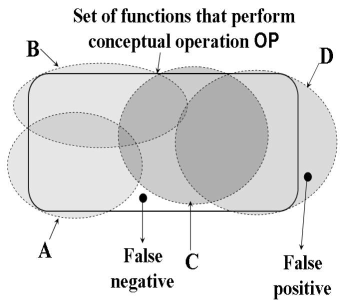


Figure 7: Writing idioms.

Currently, writing idioms is an iterative, manual procedure. For each conceptual operation OP, we used our knowledge of the kernel to reason about the sequence of steps the kernel must take to perform OP. Using this information, we extracted code-patterns, as shown in the examples above. When we were unfamiliar with the sequence of steps the kernel would take, we considered examples of functions in the kernel which perform the conceptual operation, and used these examples to formulate idioms.

Figure 7 illustrates the general methodology we used to write idioms. The goal is to find code-patterns that exactly cover the set of functions that perform OP (depicted as a box). To do so, we consider several code-patterns which approximate this set: Figure 7 shows four sets of functions, A, B, C and D, which contain four distinct code patterns. We first guessed code-patterns using our knowledge of the kernel, following which we manually refined these patterns by combining more code patterns using conjunction, or removing existing code patterns to reduce *false positives* and *false negatives*. False positives denote functions which contain the code-patterns guessed, but do not perform OP; we identified these by first obtaining the set of functions containing the code-patterns (automatically, using Algorithm 4), and manually inspecting the operations performed by each function in the set. False negatives denote functions which perform OP, but do not contain any of the code-patterns guessed. These are harder to identify; we used our knowledge of the kernel to identify missing entries in the set of functions covered by the code-patterns, and added more patterns, as required.

TAHOE currently uses about 100 idioms, representing rules for conceptual operations related to file systems and networking—it took us about a week to write these idioms. While these idioms work reasonably well (Section 4.1), we believe that it will take a shorter amount of time for experienced kernel developers to formulate idioms. Moreover, as we argue in Section 3.2.4, idioms are reusable, and writing them is a one-time activity.

Expressive Power: As mentioned earlier, code-patterns in idioms can be drawn from different functions. Thus, idioms can be used in an *interprocedural* analysis, such as ANALYZE_KERNEL, to determine if a kernel function performs an operation. However, they cannot express temporal properties, which can be used to enforce order between code patterns. For instance, "`ext2_rmdir` checks that a directory is empty before removing it" is a temporal property that cannot be expressed using idioms that follow the grammar in Figure 6(A). While temporal properties are strictly richer than what we can express using the grammar in Figure 6(A), we have been able to express idioms for most conceptual operations using the above grammar. We also note that an interprocedural analysis that checks for temporal properties is computationally expensive, and significant engineering tricks, such as those employed by MOPS [3] and MC [4], will have to be employed for the analysis to scale. On the other hand, Algorithm 3 works in time O($m + f \times n$), where m is proportional to the size of the Linux kernel, f is the number of functions in the kernel, and n denotes the number of idioms to be checked, and is thus linear in the number of idioms to be checked.

3.2.3 *Precision of Kernel Analysis*

Algorithm 3 and Algorithm 4 search each statement in the kernel for code-patterns. Thus, the analysis is precise in the following sense: if the code-patterns on the right-hand-side of an idiom $\mathcal{I}$ appear in a function f, then the conceptual operation on the left-hand-side of $\mathcal{I}$ is added to KernelSummary(f). Consequently, the precision of kernel analysis depends on the quality of idioms used. As mentioned earlier, idioms can be refined iteratively to remove false positives and false negatives; each such refinement also improves the precision of kernel analysis.

3.2.4 *Discussion*

While at first glance it may seem that TAHOE simply shifts the burden of placing hooks to that of writing idioms for conceptual operations, it is not the case, as we argue below:

1. *Idiom writing only requires knowledge of the kernel.* As we demonstrated in Section 3.2.2, writing idioms only requires an understanding of how the kernel performs the conceptual operation. In particular, kernel analysis is independent of hook analysis, and writing idioms does *not* require understanding any policy-specific code written in kernel modules such as LSM-SELinux. This is in stark contrast with current practice, where manual hook placement requires an understanding of both the conceptual operations performed by each kernel function, as well as the operations authorized by each hook.
2. *Idiom writing is a one-time activity.* Idioms need to be written only once for each version of the kernel. In addition, kernel analysis itself is a one-time activity. The results of kernel analysis can be combined with the results of hook analysis for different MAC policies. We also conjecture that idioms will not change much across different versions of the kernel, and hence will only require incremental updates as the kernel evolves. The reason for our belief is because the kernel usually performs conceptual operations in a few "standard" ways. For instance, unlinking an inode typically involves decrementing its link count (`n_link`), and this is standard across most versions of the kernel. Unless the kernel is radically restructured, the set of idioms will remain relatively stable.
3. *Idiom refinement can improve analysis quality.* Finally, we believe that iteratively refining idioms by identifying false positives and false negatives is a formal and systematic way to improve the quality of kernel analysis. We are unaware of any systematic techniques for refinement in manual hook placement.

3.3 Combining the Results of Hook and Kernel Analysis

With the results of hook analysis and kernel analysis in hand, TAHOE obtains the set of hooks that must guard each kernel function. Recall that the output of hook analysis is a set $\{\langle \mathsf{p}_i^h \parallel \textsc{op}_i^h \rangle\}$ for each hook h, and the output of kernel analysis is a set $S = \{\textsc{op}_i^k\}$ for each kernel function k. Finding the set of hooks to guard k then reduces to finding a cover for set S using the output of hook analysis. The predicates in the output of hook analysis help determine the arguments that must be passed to the hook.

Instead of giving a formal description, we illustrate the algorithm on our running example. Kernel analysis infers that KernelSummary(`ext2_rmdir`) is {DIR__RMDIR, DIR__WRITE, DIR__SEARCH}. Analysis of hooks infers that Summary(`selinux_inode_rmdir`) is {⟨true ∥ DIR__RMDIR⟩, ⟨true ∥ DIR__SEARCH⟩}, and that Summary(`selinux_inode_permission`) is as shown in Figure 4(B) (only the relevant portions of the summaries are shown). Because the operations authorized by these two hooks cover the set KernelSummary(`ext2_rmdir`), these hooks are chosen to authorize `ext2_rmdir`. The hook `selinux_inode_rmdir` unconditionally checks that the operations DIR__RMDIR and DIR__SEARCH are authorized, and can hence be called with the relevant variables in scope at locations where `ext2_rmdir` is called, for instance within `vfs_rmdir`. Because the hook `selinux_inode_permission` checks that the operation DIR__WRITE is authorized when it is invoked such that (mask $\neq$ 0) $\wedge$ (inode->i_mode & S_IFMT == S_IFDIR) $\wedge$ (mask & MAY_WRITE) is true, it is invoked with `mask` = MAY_WRITE.

As shown above, the problem of finding the set of hooks to guard each kernel function reduces to find a set cover for the set of operations performed by the kernel function. This is a well-known NP-complete problem [6]. We currently employ a simple greedy heuristic to find a set cover, based upon the number of operations in common to each hook and the kernel function. However, the number of hooks applicable for each kernel function is fortunately quite small, and if necessary brute-force search can be employed to find all possible set covers. The example above also demonstrates how predicates obtained via hook analysis determine how each hook must be called. Formally, a satisfying assignment to the predicate determines the arguments that the hook must be called with. While we manually obtain satisfying assignments in the current implementation of TAHOE, this process can easily be automated by querying a simple theorem-prover, such as Simplify [20], for satisfying assignments to predicates.

We use the term *authorization graph* to refer to the relationship obtained using the analysis discussed above, because it has the structure of an undirected, bipartite graph. The authorization relationship discovered forms the edges of this bipartite graph, whose nodes correspond to hooks and kernel functions.

```
Algorithm : FIND_CONTROLLED_KERNEL_FUNCTIONS
Input     : (i) $CG$: Call-graph of Linux kernel (ii) $AG$: Authorization
            Graph
Output    : The set of controlled kernel functions
1 foreach ($f \in CG$) do
2     foreach ($r \in$ roots of $CG$) do
3         // hooksAG($f$) is set of hooks (in $AG$) that protect $f$.
4         if there is a path from $r$ to $f$ in $CG$, and hooksAG($f$) = hooksAG($r$)
          then
5             if hooksAG($f$) $\neq$ hooksAG($c$) for at least one child $c$ of $f$ then
6                 $CKF := CKF \cup \{f\}$;
7 return $CKF$
```
Algorithm 5: Finding controlled kernel functions.

While the authorization graph relates *each* kernel function to the set of hooks that must protect it, in practice, hooks are placed only to protect a small set of kernel functions, which we call *controlled kernel functions*. The idea is that protecting these functions protects all security sensitive operations performed by the kernel. TAHOE uses the call-graph of the kernel and the authorization graph to find controlled kernel functions. Algorithm 5 describes the heuristic currently employed to find controlled kernel functions.

The basic intuition behind Algorithm 5 is to place hooks as close as possible to the functions that actually perform security sensitive operations. For instance, while our analysis infers that `sys_rmdir` does directory removal (formally, KernelSummary(`sys_rmdir`) = {DIR__RMDIR, DIR__WRITE, DIR__SEARCH}), the directory removal is actually performed by `ext2_rmdir`, which is transitively called by `sys_rmdir`. Formally, KernelSummary(`sys_rmdir`) = KernelSummary(`ext2_rmdir`), and `ext2_rmdir` is the deepest function in the call-graph reachable from `sys_rmdir` with this property. Thus, our analysis infers that `ext2_rmdir` is a controlled kernel function. Similarly, because `ext2_unlink` is the deepest function in the call graph with KernelSummary(`ext2_unlink`) = KernelSummary(`sys_unlink`), it is a controlled kernel function.

3.4 Hook Placement

Hook placement using the authorization graph is straightforward. At each location in the kernel where a controlled kernel function is called, TAHOE places the hooks determined by the authorization graph. Currently, TAHOE does not optimize hook placement (as was shown in Section 3). In the future, we plan to extend our implementation to optimize hook placement.

4. IMPLEMENTATION & EXPERIENCE

TAHOE is implemented as a plugin to the CIL toolkit [18], and consists of about 3000 lines of Objective Caml [14] code. In this section, we discuss the precision of TAHOE's analysis, its performance, and our experience with TAHOE.

4.1 Precision of Hook Placement

Methodology: To evaluate the effectiveness of TAHOE's hook placement, we used it with the Linux-2.4.21 kernel, and the hooks from LSM-SELinux, which has 149 hooks placed at 248 locations in the kernel. This version of the kernel is available both with hooks placed for LSM-SELinux and without, thus allowing us to objectively evaluate the results of TAHOE.

We have currently written idioms for conceptual operations representing file and socket operations (numbering about 100 idioms), and we evaluated the precision of TAHOE in placing these hooks. For each hook, we manually compare its placement in LSM-SELinux to the placement suggested by TAHOE. Because the hook placement of LSM-SELinux has been extensively verified, we believe that it is bug-free, and hence provides a good benchmark to compare the effectiveness of TAHOE. We report two metrics, false negatives and false positives, as discussed below.

False negatives: A hook placed in LSM-SELinux, but not placed by TAHOE classifies as a false negative. Because a false negative in the output of TAHOE corresponds to a missing hook, it results in insufficient authorization, thus leading to a potential security hole.

TAHOE currently analyzes a subset of file hooks (26 hooks) from the LSM-SELinux kernel module (Section 4.3 has details on the hooks currently not analyzed) which authorize a variety of conceptual operations on files and inodes. LSM-SELinux places these hooks at 40 different locations in the kernel. When we used TAHOE for obtaining hook placement, the output was missing 5 hooks, which fell into 3 categories, as discussed below:

1. The hook `selinux_file_receive`, placed in a kernel function `scm_detach_fds` in LSM-SELinux, was missing from the output of TAHOE. We found that the reason was because kernel analysis was missing an idiom each for conceptual operations FILE__READ, FILE__WRITE and FILE__APPEND. This false negative is eliminated by adding idioms for these operations.
2. The hook `selinux_file_set_fowner`, placed in 3 kernel functions, was missing from the output. We found that this hook was not analyzed properly by TAHOE. In particular, this hook updates a data structure internal to LSM-SELinux, and does not contain any access vectors. As a result, the analysis described in Section 3.1 determined that this hook does not analyze any operations, leading to the false negatives. These false negatives are easily eliminated by considering the update to the data structure as a new conceptual operation, and adding corresponding idioms for kernel analysis.
3. The hook `selinux_inode_revalidate`, placed in a kernel function `do_revalidate`, was missing from the output. However, upon closer investigation we found that this was *not* a false negative. In particular, in LSM-SELinux, the authorization query posed by this hook always succeeds. As a result, TAHOE infers that no operations are authorized by this hook. This example shows that semantic information contained in hooks is valuable in determining hook placement.

Category	Num. Locs.	False Pos.	False Neg.
File hooks(26)	40	13	4
Socket hooks(12)	12	4	0

Figure 8: Comparison of TAHOE's output with LSM-SELinux. False positives count locations where TAHOE places an extra hook, while false negatives count locations with missing hooks.

TAHOE currently analyzes 12 socket hooks, which are placed at 12 locations in the kernel in LSM-SELinux. It identified all these hook placements without any false negatives.

False positives: The output of TAHOE may contain hooks which are not placed in LSM-SELinux. This may arise because of one of two reasons: (i) Imprecision in the analysis, for instance, because the kernel analysis infers that a kernel function performs more controlled operations than it actually does, or (ii) Unoptimized hook placement, for instance, as discussed in Section 3, where `selinux_inode_permission` was placed redundantly. We only classify hooks in category (i) as false positives, because they result in extra authorizations being performed. While false positives do not lead to security holes, they may result in entities with requisite permissions being denied authorization. Thus, it is desirable to have a low false positive rate.

We found that TAHOE had false positives at 13 out of the 40 locations in LSM-SELinux where file hooks are placed, and at at 4 out of the 12 locations where socket hooks are placed in LSM-SELinux. In each case, one extra hook was placed in addition to the required hook. We observed that this imprecision was because of imprecision in the idioms employed by kernel analysis. In particular, several functions were wrongly classified as performing the conceptual operations FILE__READ, DIR__READ, FILE__EXECUTE and DIR__SEARCH. We expect that further refinement of these idioms will reduce the number of false positives.

Effectiveness at finding controlled kernel functions: In the discussion so far, we evaluated TAHOE's hook placement at the controlled kernel functions as defined by LSM-SELinux hook placements. However, TAHOE also infers controlled kernel functions, using the heuristic described in Algorithm 5. We found that the controlled kernel functions identified by TAHOE for placing file and socket hooks were the same as those identified by LSM-SELinux in all but one case. TAHOE identified `open_namei` as a controlled kernel function that performed several controlled operations, including FILE__CREATE and FILE__EXECUTE. However, in LSM-SELinux, hooks to protect these operations were placed in functions that were called by `open_namei`, as opposed to locations where `open_namei` was called.

4.2 Performance

We ran timing experiments on a machine with a 1GHz AMD Athlon processor, and 1GB RAM. Hook analysis took about 11 minutes, while kernel analysis took about 8 minutes. The smaller runtime for kernel analysis can be attributed to its simpler nature.

4.3 Opportunities for Improvement

While we are encouraged by TAHOE's ability to place hooks, we have identified some shortcomings, which we plan to address in future work. First, TAHOE currently does not analyze all hooks in LSM-SELinux. In particular, LSM-SELinux has several hooks to allocate and free data structures internal to the kernel module (which implements hooks). While these do not authorize any conceptual operations, it is crucial that they be placed at appropriate locations in the kernel; improper placement of these hooks could lead to runtime exceptions. Second, while TAHOE's kernel analysis recovers the conceptual operations performed by a kernel function, it currently does not recover the specific *resource instances* on which they are performed—this is currently done manually. For instance, in Figure 3(B), the resource instances `dir` and `dentry` were recovered manually. Third, TAHOE can currently only place hooks at the granularity of function calls, i.e., it places hooks at each location where controlled kernel functions are called. There are cases in LSM-SELinux (`selinux_file_set_fowner`), where hooks are placed at a finer granularity, such as before modifications of kernel data structures. Last, while idiom writing and refinement can improve the results of kernel analysis, they are manual procedures. We plan to investigate automatic idiom writing and refinement techniques in the future.

5. RELATED WORK

Prior work on the formal analysis of hook placement in the LSM framework has focused on verifying the correctness of existing hook placement. Vali [10] is a runtime tool to determine the consistency of hook placement in LSM. It is based upon the observation that hook placement in LSM is typically consistent across different code paths in the kernel, and thus inconsistencies are indicative of bugs. Analysis is performed on execution traces obtained by running an instrumented version of the kernel. The authors also demonstrate a static version of Vali, which is built using the analysis capabilities of JaBA [13], a static analysis tool to automatically identify the least privileges needed to execute a Java program. Zhang *et al.* [23] demonstrate the use of a type-qualifier-based tool, CQUAL [5], to determine inadequacies in hook placement. In particular, their analysis determines that a resource has been authorized by a hook before a conceptual operation is performed on it. However, the analysis requires as input the set of conceptual operations performed on a resource in order to verify the adequacy of hook placement, which is used as the type-qualifier-lattice by CQUAL. They use the output of Vali to obtain the type-qualifier lattice. The above efforts however do not use the source code of hooks in their analysis.

While we have focused on the problem of automatic hook placement, we believe that the analysis employed by TAHOE can be used for verifying existing hook placement as well. For instance, authorization graphs extracted by TAHOE can be compared against the authorization graph corresponding to the existing hook placement, and anomalies can be flagged as potential errors.

The analyses employed by various stages of TAHOE are also related to prior work. Hook analysis employs a flow-sensitive, context-sensitive program analysis, which has been explored by several tools in the past, including MOPS [3], MC [4], and JaBA [13]. The use of idioms in kernel analysis is conceptually similar to the use of compiler-extensions (written in a language called Metal) by MC. Vali used runtime techniques to extract the authorization graph from a hook-placed kernel, and used consistency analysis on this graph to identify anomalies. Authorization graphs are also similar to access rights invocation graphs used by JaBA.

While TAHOE addresses the problem of enforcing a given authorization policy by placing hooks to a kernel module that encapsulates the policy, it does not ensure that the authorization policy itself meets security goals, such as integrity and confidentiality. SLAT [8] and Gokyo [11] are tools that can be used for this purpose. Both these tools construct an abstract model of the authorization policy (e.g., an SELinux policy), and analyze them to determine conflicts between the policy and the system security goals.

6. CONCLUSION

The emerging popularity of the LSM framework to implement MAC authorization policies, coupled with recent interest in LSM-like frameworks for user-level applications [12, 21] underscores the need for security of these frameworks. We believe that the techniques presented in this paper are a useful first step towards automatic enforcement of authorization policies through the use of formal reasoning and program analysis.

Acknowledgments: Many thanks to Shai Rubin and the CCS reviewers for their insightful comments. Thanks also to Vijayan Prabhakaran for patiently answering our file system questions.

7. REFERENCES

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

[2] L. Badger, D. Sterne, D. Sherman, K. Walker, and S. Haghighat. A domain and type enforcement UNIX prototype. In *5th USENIX UNIX Security*, June 1995.

[3] H. Chen. *Lightweight Model Checking for Improving Software Security*. PhD thesis, University of California, Berkeley, Fall 2004.

[4] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific programmer-written compiler extensions. In *4th ACM/USENIX OSDI*, December 2000.

[5] J. S. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN PLDI*, May 1999.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, New York, NY, 1979.

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

[8] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in security-enhanced Linux. *JCS*, 13(1):115–134, 2005.

[9] Example idioms. www.cs.wisc.edu/∼vg/papers/ccs2005a/idioms.html.

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

[11] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *12th USENIX Security*, August 2003.

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

[13] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *ACM SIGPLAN OOPSLA*, November 2002.

[14] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system (release 3.08). Technical report, INRIA Rocquencourt, July 2004.

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

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

[17] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *11th Intl. Conf. on Compiler Construction*, April 2002.

[19] M. Sharir and A. Pnueli. Two approaches to interprocedural dataflow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice Hall, 1981.

[20] Simplify. http://research.compaq.com/SRC/esc/Simplify.html.

[21] Tresys Technology. Security-enhanced Linux policy management framework. http://sepolicy-server.sourceforge.net.

[22] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *11th USENIX Security*, August 2002.

[23] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *11th USENIX Security*, August 2002.