TALISMAN: Tamper Analysis for Reference Monitors

Frank Capobianco*[†], Quan Zhou*[†], Aditya Basu*, Trent Jaeger*[§][‡] and Danfeng Zhang*[¶][‡]
*The Pennsylvania State University, [§]University of California, Riverside, [¶]Duke University
{fnc110, quan.zhou, aditya.basu}@psu.edu, trentj@ucr.edu, danfeng.zhang@duke.edu

***Abstract*—Correct access control enforcement is a critical foundation for data security. The *reference monitor* is the key component for enforcing access control, which is supposed to provide tamperproof mediation of all security-sensitive operations. Since reference monitors are often deployed in complex software handling a wide variety of operation requests, such as operating systems and server programs, a question is whether reference monitor implementations may have flaws that prevent them from achieving these requirements. In the past, automated analyses detected flaws in complete mediation. However, researchers have not yet developed methods to detect flaws that may tamper with the reference monitor, despite the many vulnerabilities found in such programs. In this paper, we develop TALISMAN, an automated analysis for detecting flaws that may tamper the execution of reference monitor implementations. At its core, TALISMAN implements a precise information flow integrity analysis to detect violations that may tamper the construction of authorization queries. TALISMAN applies a new, relaxed variant of noninterference that eliminates several spurious implicit flow violations. TALISMAN also provides a means to vet expected uses of untrusted data in authorization using endorsement. We apply TALISMAN on three reference monitor implementations used in the Linux Security Modules framework, SELinux, AppArmor, and Tomoyo, verifying 80% of the arguments in authorization queries generated by these LSMs. Using TALISMAN, we also found vulnerabilities in how pathnames are used in authorization by Tomoyo and AppArmor allowing adversaries to circumvent authorization. TALISMAN shows that tamper analysis of reference monitor implementations can automatically verify many cases and also enable the detection of critical flaws.**

I. INTRODUCTION

Correct enforcement of access control is critical for protecting data security. For example, in the early 2000s, a flurry of Internet-scale worms exposed weaknesses in access control enforcement in operating systems [1]. Security researchers proposed that security-sensitive software, such as operating systems and server programs, must enforce access control in a manner that complies with the *reference monitor concept* [2], [3], which requires *complete mediation* of all security-sensitive operations by *tamperproof* reference monitors *verified to be correct* through complete testing. As a result, reference monitor implementations were created for several software systems, including Linux [4], the X Server [5], Postgres [6], and Apache [7].

[†]The authors contributed equally to this work.
[‡]Most of this work is done while the authors were working at Penn State.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.241031
www.ndss-symposium.org

Access control authorizes requests from clients to perform security-sensitive operations, so malicious clients may want to exploit incorrect reference monitor implementations to gain unauthorized access. For example, researchers found that clients could bypass authorization in Linux in some cases because its reference monitor implementation did not ensure complete mediation [8], [9]. However, concerns about tampering the reference monitor implementations was not examined. The assumption at the time was that since the reference monitor implementations were integrated with the host programs (i.e., the operating systems and server programs), a flaw in the host program would be no different from a flaw in the reference monitor. However, in the 20-plus years since the introduction of these reference monitor implementations, the mindset has changed. Most host programs include defenses, such as ASLR, to hide critical data from adversaries even when a vulnerability is exploited. In addition, interest in *privilege separation* [10], [11], [12] within programs and even operating systems has only increased, aiming to prevent access to critical data even if a component of the program is compromised. As a result, detecting whether reference monitor implementations may be tampered is becoming more important.

In the past, researchers developed several automated analyses to detect failures in complete mediation [13], [14], [15], [16], [17], [18], [19], [8], [9], [20]. Such analyses detect the absence of authorization checks [13], [20], [9], detect inconsistent use of authorization checks [8], [16], and propose placements and/or repairs for missing checks [14], [15], [17], [18], [19]. These methods use control flows to determine what authorization checks are expected at particular points in the program. More recent work detects missing security checks more broadly (i.e., for mediating other kinds of accesses) by detecting missing checks [21], inconsistent checks [22], as well as generating check placements [23], [24]. Similarly, data flow analysis [24], [25], [21] has been used to detect program points that may leak data. For example, King *et al.* [24] use information flow analysis to detect where data flows may violate a security policy to generate placements for security checks to mediate those data flows. While there are many analyses that have been proposed to detect failures in mediation and leakage, we are not aware of analyses targeting the tamperproof requirement of reference monitors[1].

Prior work on tamper-resistant software [26], [27], [28], [29] has generally involved digital rights management, protecting the integrity of the code and how it executes through forms of obfuscation. For the tamperproof reference monitor requirement, the problem is somewhat different. In this case, we want to ensure that any data used by a reference monitor to construct the *authorization queries* used to check for

[1]The *verified to be correct* requirement of reference monitors has not been examined either, but we focus on the tamperproof requirement in this paper.

authorized access satisfies a set of integrity requirements for the reference monitor. This problem is complicated by the fact that the construction of authorization queries depends on the operation requests that are made by untrusted clients of the host program. However, we observe that reference monitors only need to convert the operation requests into monitor-specific subjects, objects, and operations to construct authorization queries. This is typically done in a common manner for each reference monitor implementation, enabling untrusted inputs to be endorsed in many cases using a few common methods.

In this paper, we develop TALISMAN, a system for tamper analysis of reference monitor implementations. At its core, TALISMAN is a field-sensitive, information flow integrity analysis that detects tampering between the data used by the reference monitor (i.e., sources) and authorization queries (i.e., sinks). However, we find that standard noninterference is too restrictive (i.e., reports many safe code) because it is common for a reference monitor to perform multiple sequences of authorization queries depending on the client inputs. According to strict noninterference, such code violates the integrity policy since low-integrity inputs interfere with the existence/absence of high-integrity authorization queries. To address this limitation, we define *relaxed noninterference*, which more precisely captures the intended integrity policy: the sequence of authorization queries checked can be a choice of the reference monitor based on the operation request. TALISMAN automates the information flow analysis by constructing information flow problems from reference monitor code (i.e., identifying and labeling sources and sinks), detecting information flow integrity violations, and applying integrity endorsement. The few remaining violations indicate issues that require manual assessment, which may be caused by ad hoc or complex implementations or true tamper issues in need of vetting.

We applied TALISMAN to evaluate three reference monitor implementations of the Linux Security Modules (LSM) framework, Linux's access control enforcement system. The AppArmor [30], SELinux [31], and Tomoyo [32] LSMs are three distinct reference monitor implementations from three separate organizations that perform Linux access control using their own approaches[2]. Using TALISMAN, we analyzed 145 hook functions with 351 arguments to authorization queries, finding that the integrity of 80% of the arguments can be verified including those that can be endorsed. By applying relaxed noninterference, TALISMAN reduces the integrity violations due to implicit flows by approximately 60% for SELinux and 90% for Tomoyo and AppArmor, greatly reducing the subsequent effort in removing false positives. Of the remaining cases, TALISMAN found information flow violations where the ad hoc use of pathnames for authorization presents a vulnerability that can allow unauthorized access, due to differences in canonicalization between the Tomoyo and AppArmor LSMs and the Linux file systems. The addition of runtime endorsement to prevent tampering incurs less than a 0.14 μs of overhead for each of the LSMs for LMBench 3.0, except when `inode_permission` is endorsed, which causes some redundant overhead as discussed in Section VIII-F.

[2]Several reference monitor implementations for server programs, such as Apache [33], Postgres [34], and X Server [5] were all developed in coordination with the SELinux community. So their implementations are similar to SELinux as discussed in section II-A.

```
static int inode_has_perm(const struct cred *cred,
    struct inode *inode, u32 perms, struct
    common_audit_data *adp) {
  ...
  validate_creds(cred);
  if (unlikely(IS_PRIVATE(inode)))
    return 0;

  sid = cred_sid(cred);
  isec = inode->i_security;
  return avc_has_perm(sid, isec->sid, isec->sclass,
      perms, adp);
}
```

Fig. 1. Function `inode_has_perm` from SELinux

In this paper, we make the following contributions:

- We develop an information flow analysis based on the novel principle of *relaxed noninterference*, which allows the sequence of authorization queries run by a reference monitor to depend on the untrusted operation request, as intended by reference monitor implementations.
- We develop a tamper analysis tool, called TALISMAN, that provides field-sensitive, information-flow analysis for a three-dimensional integrity lattice that detects violations of relaxed noninterference[3].
- We apply TALISMAN to three reference monitor implementations of the Linux Security Modules framework, namely AppArmor, SELinux, and Tomoyo, finding that the integrity of 80% of the arguments to authorization queries can be verified. In addition, using TALISMAN enables us to find a security vulnerability caused by the difference in how file pathnames are canonicalized between AppArmor/Tomoyo and Linux file systems, which could allow unauthorized access.

We have reported the file pathname vulnerability to a Linux VFS maintainer, who has confirmed the problem, and the AppArmor and Tomoyo communities. We are exploring patch options.

II. PROBLEM DEFINITION

In this section, we examine the impact of tampering on reference monitors and define the problem of tamper analysis for reference monitors.

A. *Tampering of Authorizations*

Reference monitors, such as Linux Security Modules [4] (LSMs) like AppArmor [30], SELinux [31] and Tomoyo [32], authorize operation requests from their clients. Authorization determines whether a subject (e.g., client user, process, etc.) is allowed to perform an operation (e.g., read, write, etc.) upon an object (e.g., file, socket, etc.). We find that authorization may be prone to tampering in at least three ways. We use the LSMs as example although the problems are common in other systems as well.

Case 1 (Store Security Identifiers in Host Program Structures). We find that reference monitors often store the security

[3]The source code is accessible at https://github.com/isolachine/talisman.

```
1  static int apparmor_file_open(struct file *file,
2          const struct cred *cred) {
3    profile = aa_cred_profile(cred);
4    if (!unconfined(profile)) {
5      struct inode *inode = file_inode(file);
6      struct path_cond cond =
7          { inode->i_uid, inode->i_mode };
8
9      error = aa_path_perm(OP_OPEN, profile,
10         &file->f_path, 0,
11         aa_map_file_to_perms(file), &cond);
12     fcxt->allow = aa_map_file_to_perms(file);
13   }
14   return error;
15 }
```

Fig. 2. An AppArmor hook function `apparmor_file_open`

identifiers for authorization in the host program data structures. For example, SELinux stores a security identifier in the field `i_security` for each `inode`, as shown in Figure 1. The security identifier `isec` is extracted (line 8) for authorization by the function `avc_has_perm` at line 9. This security data is set by SELinux when the inode object is created. But since the security identifier is stored in the inode object, which is used in many system call operations, it is prone to illicit modification, even when using defenses like KASLR. Should this happen, the authorization process can be tampered by using the modified security identifiers. In this case, we expect that the inode's security identifier should be unmodified after its creation by SELinux; a simple check by the reference monitor can validate that fact.

Case 2 (Incorrect Use of Operation Request Input). Some uses of the operation requests submitted by clients for authorization can lead to vulnerable behaviors when done incorrectly. Figure 2 shows that the AppArmor function `apparmor_file_open`, which authorizes file access, uses the value `file->f_path` as the security identifier for the file object in the call to `aa_path_perm` on line 9, which makes the authorization decision. However, the canonicalization of the pathname by AppArmor differs from that performed by Linux file systems (and may vary among Linux file systems as well), causing AppArmor to check its policy incorrectly. Researchers recently found that this problem manifests itself when using file systems supporting case-preserving naming [35]. For example, a user may provide the name "FOO" for a pathname element, but the recent support for case-preserving directories in the Linux Ext4 file system may resolve that name to a file named "foo". If AppArmor checks whether access is permitted to "FOO" rather than "foo", incorrect authorization may result, as we detail in Section VIII-C. We note that Tomoyo also uses pathnames for security identifiers for files, leading to the same problem.

Case 3 (Authorization Bypass). LSMs often use data from the object itself in conditional statements that may result in bypassing authorization entirely. Consider the snippet of code from the function `inode_has_perm` in Figure 1. If the function `IS_PRIVATE` at line 4 returns true, the function returns 0 (i.e., the request is authorized) without invoking the authorization function `avc_has_perm`. Conditions that use inputs to the hook function (i.e., the `inode` argument) to allow authorization to be bypassed are common, but raise the risk of incorrect authorization if users can modify the value used in the conditional. We aim to detect such risky cases automatically

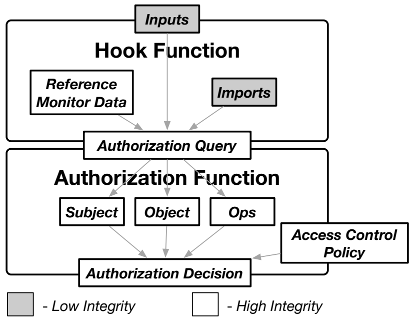


Fig. 3. Authorization in modern reference monitor implementations, showing low-integrity and high-integrity data. Authorization uses two functions: (1) *hook functions* to construct authorization queries and (2) *authorization functions* to compare the query to the monitor's access control policy.

to enable vetting.

B. The Tamper Problem for Reference Monitors

Figure 3 shows the tampering problem abstractly. First, we find that modern reference monitor implementations divide the task of authorization into two parts, which we designate as the *hook function* and the *authorization function*. First, hook functions transform client inputs specifying each operation request into a monitor-specific *authorization query*, which specifies the subject, object, and operations to authorize using the reference monitor's own security identifiers. Then, authorization functions compare the security identifiers for the subject, object, and operations in the authorization query to the reference monitor's access control policy to determine whether to authorize the request (i.e., determine the *authorization decision*).

Figure 3 shows that tampering is possible because the reference monitors use client-provided *inputs* and *imports*, in addition to high-integrity *reference monitor data*, in creating authorization queries. Inputs are the arguments passed directly to the hook function for the operation request. The arguments include the program resources used to compute security identifiers for subjects, objects, and operations. Imports are ad hoc program data (e.g., global variables) used by the reference monitor. Both of them may be modified in a manner that could cause incorrect authorization decisions. In the previous section, clients provide file pathnames as input that cause an LSM to compute an incorrect object security identifier (Case 2). In addition, LSMs may compare client inputs (`inode`) to kernel-defined values (`IS_PRIVATE`) to determine whether to perform authorization at all (Case 3).

The *reference monitor concept* [2] was proposed in 1972 to define a set of design requirements for secure authorization in operating systems. The reference monitor concept consists of three requirements of any reference monitor: (1) it must provide *complete mediation* of all security-sensitive operations; (2) it must be *tamperproof*; and (3) its operation must be *verified to be correct* via complete testing. Researchers have explored a variety of techniques to assess the correctness of authorization, particularly complete mediation, by assessing control flow [13], [20], [9], [8], [17], [18], [14], [15], [16] (i.e., all control flows to security-sensitive operations are mediated) and data flow [36], [37], [24], [25] (i.e., all accesses to data obey an information flow policy). However, those techniques do not detect tampering.

In this paper, we propose a tamper analysis to detect when and how hook functions violate a tamperproof requirement in constructing authorization queries. More specifically, we check that the arguments to authorization queries (i.e., the subject, object, and operation identifiers) are constructed in a manner free from tampering through the use of low-integrity inputs and imports (Cases 1 and 2 above). When low-integrity inputs and imports are used to construct authorization queries in an expected manner, we want to endorse these cases with minimal effort, when possible. In addition, our analysis detects cases where authorization may bypass evaluation of one or more authorization queries, and then reports the impact on authorization correctly with respect to the reference monitor semantics (Case 3).

C. Information Flow Analysis

Information flow control provides a fine-grained control on information propagation. Unlike traditional access control policies, information flow control offers an end-to-end security guarantee, known as *noninterference* [38]. In the context of information integrity of reference monitors, noninterference means that low-integrity (i.e., untrusted) inputs will never tamper high-integrity (i.e., trusted) authorization queries. To define integrity, we follow the widely-used lattice-based model [39] where information integrity is modeled as *security levels* drawn from an integrity lattice. More specifically, let data d_1 and d_2's integrity be levels ℓ_1 and ℓ_2 (drawn from a security lattice $\mathcal{L}$) respectively. Then, d_1 may flow to d_2 if and only if $\ell_2 \sqsubseteq \ell_1$ (i.e., d_1's integrity level is at least as high as d_2's). Note that an expressive lattice-based model is needed for analyzing the integrity of modern reference monitor implementations, as it cannot be simply categorized as trusted or untrusted; the data used in authorization has multiple possible sources, purposes, etc., as we elaborate in Section VI-A.

A variety of methods have been proposed to validate that an information flow policy is successfully enforced in a system, including dynamic and static analyses [40]. In this paper, we employ static information flow analysis due to its soundness guarantee: validated code obeys integrity policy in all possible executions. Although static information flow analysis is well-studied in the literature, reference monitors present unique challenges that we tackle in this paper. First, noninterference is known to be too restrictive when analyzing non-trivial systems [40]. In this paper, we propose a variant of noninterference that is more suitable for tamper analysis of reference monitors: it reduces spurious implicit flow violations in the reference monitors we evaluate by 58% to 91% relative to the strict noninterference. Second, endorsement, like its counterpart for secrecy, declassification, is often application-specific. In this paper, we develop a systematic approach of endorsing information flow integrity violations for common cases. Third, analysis precision is crucial due to the non-trivial codebase of reference monitors. We identify the bottlenecks of existing static information flow analysis and apply several improvements that remove false positives that account for 65% to 91% of the positives in the evaluated reference monitors found by a more naive implementation. We elaborate on the above technical innovations in Sections V and VI.

III. Solution Overview

We propose Talisman, which performs tamper analysis on reference monitors. The Talisman architecture is shown in Figure 4. Talisman operates in three steps to determine how authorization queries may be tampered and endorse common cases. First, Talisman generates an information-flow problem for each of the hook functions by identifying and labeling its data sources using an integrity lattice. Second, Talisman employs a relaxed variant of noninterference, we call *relaxed noninterference*, to detect information-flow violations from the hook function's information-flow problem, accounting for information flows that only differ in the sequences of sinks they may visit to reduce spurious implicit flow violations. Third, Talisman employs endorsement to resolve information-flow violations for expected transformations. The result is a set of tamperproof hook functions and/or remaining information-flow violations to be resolved manually.

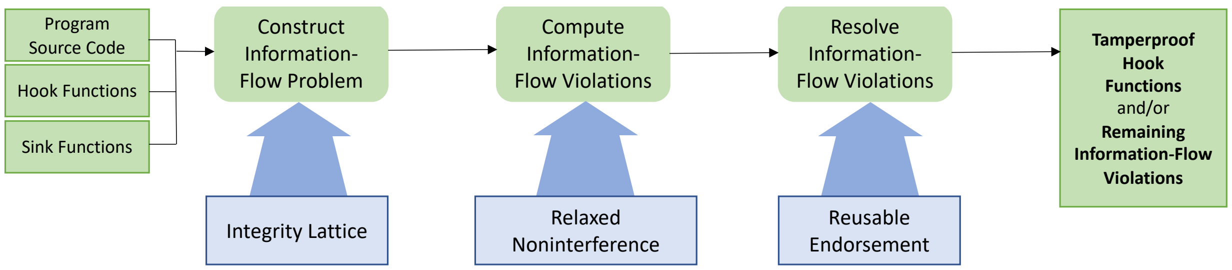
Not all information-flow violations remaining are necessarily errors or vulnerabilities. There are some cases where hook functions create complex scenarios that we discuss in Section VIII. While some cases can be endorsed (e.g., some code of Case I and categories for Case II), the remaining may be addressed by endorsements that we have not designed yet. Thus, we further classify the remaining cases to distinguish the challenges that may remain. Notably, as described in Section VIII-C, the use of pathnames in Tomoyo and AppArmor is vulnerable to exploitation as currently implemented, so the functionality must be changed to enable tamperproof operation (e.g., Case 2 in Section II-A). In addition, as described in the Section VIII-B, we find that some cases may be best addressed by code refactoring while others depend on implicit information from the kernel that needs to be made visible to the analysis and preserved as the kernel changes (e.g., Case 3 in Section II-A). These lessons may impact the future design of authorization modules, in the kernel and elsewhere, to ensure tamperproof operation.

IV. Threat Model

Figure 3 shows our threat model. We assume that the reference monitor code, including hook functions and authorization functions, is high integrity (i.e., trusted). Any reference monitor data defined by the reference monitor itself in these functions is assumed to be high integrity as well.

On the other hand, we assume that all data external to the reference monitor is low integrity (i.e., untrusted), including any inputs to the reference monitor from the program and other program imports. In this paper, tampering is possible if an authorization query is built using low integrity data that has not been endorsed using a trusted endorser of the reference monitor (e.g., using high integrity data stored by the reference monitor).

We assume that the reference monitor is type-safe. Historically, analyses to validate complete mediation [13], [20], [9], [8], [17], [18], [24], [25], [14], [15], [16] assumed type safety, yet were valuable in finding real vulnerabilities in access control enforcement. Further, we assume that memory errors that may be present in the kernel do not modify the reference monitor data or execution. That is, the reference monitor is still assumed to be high integrity.



Program Source Code
Hook Functions
Sink Functions
Construct Information-Flow Problem
Compute Information-Flow Violations
Resolve Information-Flow Violations
Tamperproof Hook Functions and/or Remaining Information-Flow Violations
Integrity Lattice
Relaxed Noninterference
Reusable Endorsement

Fig. 4. TALISMAN overview - *Rectangular boxes represent inputs, where as rounded boxes represent computation steps that transform hook functions into tamperproof hook functions and/or report remaining information-flow violations.*

While this threat model covers only a subset of the possible attack vectors from the kernel via misuse of user process input, any failure of the reference monitor to achieve tamperproofing under this threat model will also apply in more powerful threat models. We discuss strengthening the threat model and further countermeasures to prevent tampering in Section IX.

V. DESIGN PRINCIPLES

In this section, we propose a methodological approach in handling the tampering analysis with information flow control. We first show the limitation of the standard noninterference for LSM tamper analysis and propose its variant to accommodate our threat model. Then, we show the design of our static information flow analysis as well as plugging the relaxed noninterference into the analysis.

A. Notation

In the context of information integrity, the focus of this paper, we consider a deterministic program (e.g., a hook function) $\mathcal{H}$ whose small-step semantics has the form of

$$\langle c, \mu \rangle \xrightarrow{\texttt{call}(s,\vec{v})} \langle c', \mu' \rangle$$

Here, c and c' are commands, μ and μ' are the memory states (i.e., mapping from variables to their values) and $\rightarrow$ represents the transition from c to c' in one evaluation step. When c calls one of the predefined sink functions s with parameter values $\vec{v}$, we annotate the transition with $\texttt{call}(s, \vec{v})$ above the transition edge; otherwise, we leave it empty. We use $\vec{v}$ to represent the sink function call's *actual parameters* (i.e., the values being passed to the function). We intentionally leave the semantics abstract here as the noninterference definition is applicable to any concrete semantics with the format above.

In our context, $\mathcal{H}$ is the invocation of one of the predefined hook functions of interest, and $\mathcal{H}$ terminates with a return value that represents an authorization decision (i.e., pass or fail). Starting from its entry command c_0 and memory state μ_0, $\mathcal{H}$ emits a sequence of transitions in the following form:

$$\begin{aligned} &\langle c_0, \mu_0 \rangle \rightarrow \langle c_1, \mu_1 \rangle \xrightarrow{\texttt{call}_1(s_3,\vec{v}_1)} \langle c_2, \mu_2 \rangle \rightarrow \ldots \\ &\rightarrow \langle c_{n-1}, \mu_{n-1} \rangle \xrightarrow{\texttt{call}_2(s_5,\vec{v}_2)} \langle c_n, \mu_n \rangle \end{aligned}$$

The subscript of `call` is its corresponding index among all sink calls in the small-step semantics. In the execution trace above, there are two sink calls ($\texttt{call}_1$ and $\texttt{call}_2$) emitted by c_1 and c_{n-1} that invoke sink functions s_3 and s_5 respectively.

For simplicity, we use the term *trace*, written as τ, to represent a finite sequence of all sink calls in the order of their corresponding index found in the small-step semantics. More specifically, we write $\langle c_0, \mu_0 \rangle \hookrightarrow \tau$ when c_0 under μ_0 emits a trace τ. For the example above with two sink calls $\texttt{call}_1$ and $\texttt{call}_2$, we simply write

$$\langle c_0, \mu_0 \rangle \hookrightarrow (\texttt{call}_1(s_3, \vec{v}_1), \texttt{call}_2(s_5, \vec{v}_2))$$

instead of its full small-step semantics.

For a trace τ, we also denote its return value as a numerical number $\mathcal{R}(\tau) \in \mathbb{Z}$. A hook function returns an numerical code to represent a successful or failed check, thus respectively granting or denying the requested permission.

As discussed in Section II-C, we follow the widely-used lattice-based model [39] where information integrity is modeled as security levels drawn from an integrity lattice. We use $\mathcal{L}$ to denote an integrity lattice and introduce the concrete lattice that we use in tamper analysis in Section VI. To specify the integrity levels of data sources and sink functions, we assume a *source-label* map $\mathcal{S}$[4] and a *sink-label* map $\mathcal{I}$ respectively. In particular, $\mathcal{S}$ specifies the integrity levels of used data (e.g., global variables, function inputs, constants)

$$\mathcal{S}(x) \rightarrow \ell \in \mathcal{L}$$

whereas $\mathcal{I}$ specifies the integrity levels on each sink function

$$\mathcal{I}(s, i) \rightarrow \ell \in \mathcal{L}$$

where ℓ is the integrity level for the i-th parameter of s.

Finally, two memory states μ and μ' are ℓ*-equivalent*, written as $\mu \approx_\ell \mu'$, if they share the same values on all data at or above a security level $\ell \in \mathcal{L}$:

$$\mu \approx_\ell \mu' \iff \forall x.\ \ell \sqsubseteq \mathcal{S}(x) \Rightarrow \mu(x) = \mu'(x)$$

B. Strict Noninterference and Its Limitations

Consider a hook function $\mathcal{H}$ with inputs of various integrity levels. Informally, noninterference requires that for any two different executions of $\mathcal{H}$ where only low-integrity inputs change, the high-integrity parameters of any called sink function must stay the same. Such restrictions imply that an attacker who controls low-integrity inputs may never tamper the values of high-integrity parameters passed into sink functions. More formally,

[4]The computation that constructs the mapping is explained in Sec. VI-B.

```
1  static int apparmor_ptrace_traceme(
2          struct task_struct *parent) {
3    int error = cap_ptrace_traceme(parent);
4    if (error)
5      return error;
6
7    return aa_ptrace(parent, current, PTRACE_MODE_ATTACH);
8  }
```

Fig. 5. Source code for `apparmor_ptrace_traceme`.

Definition V.1 (Strict noninterference). *A hook function $\mathcal{H}$ satisfies* strict noninterference *if for any integrity level $\ell \in \mathcal{L}$ and any two arbitrary ℓ-equivalent memory states $\mu_0 \approx_\ell \mu_0'$, we have:*

$$\begin{gathered}\langle \mathcal{H}, \mu_0 \rangle \hookrightarrow \tau \wedge \langle \mathcal{H}, \mu_0' \rangle \hookrightarrow \tau' \wedge \\ \forall i, s, \vec{v}.\ \mathtt{call}_i(s, \vec{v}) \in \tau \implies (\mathtt{call}_i(s, \vec{v'}) \in \tau' \wedge \\ (\forall 1 \leq j \leq s_a.\, \ell \sqsubseteq \mathcal{I}(s, j) \Rightarrow \vec{v}[j] = \vec{v'}[j]))\end{gathered}$$

However, the strict noninterference property is too conservative for analyzing security hooks: it might reject many secure ones. We use the `apparmor_ptrace_traceme` function in Figure 5 as a representative example. This function checks whether the subject `parent` task (with integrity label ℓ_1) has the permission to trace the object, the `current` process (with integrity label ℓ_2). In this example, the designated sink function is `aa_ptrace`. Consider two execution traces. The first trace returns a non-zero error code at line 5 due to a failed check on the relationship between `parent` and `current` at line 3, resulting in no sink calls. The second trace passes the check at line 3 and eventually reaches the sink function. According to Definition V.1, this hook function violates strict noninterference because a sink call found in one trace cannot be found in another trace. In other words, the sequence of sink call is influenced by low integrity data. But this is clearly a false alarm, given that the early return at line 5 is intended and correct because an error is detected and there is no need for a further access control check. It is also very common for a security hook to call different sink functions based on its operation parameter (e.g, file read vs. file write). Such influences on the selection of sink functions are also part of the intended functionality of the authorization process.

C. Relaxed Noninterference

Based on the observation above, we argue that the threat to tamperproof reference monitors is that an attacker might control untrusted inputs to either tamper the input values passed to sink functions or bypass all sink functions and trick a hook function to return success (e.g., value 0 in LSMs) when it should not. Hence, we present a *relaxed noninterference* that allows untrusted inputs to change the sequence of sink function calls, or bypass them completely when it is safe to do so. Note that although confidentiality and integrity are usually considered as duals in terms of noninterference, relaxed noninterference is uniquely designed for tamper analysis, as from the confidentiality perspective, any absence of a sink function call also leaks information via implicit flows and/or side channels.

Consequently, to relax strict noninterference, we only need to (1) make sure untrusted input cannot bypass all sink calls and return success, and (2) compare the sink calls that can be matched between two different traces, which is captured by the relaxed noninterference we introduce next.

Definition V.2 (Relaxed noninterference). *Let* SUCCESS *be a success code. A hook function $\mathcal{H}$ satisfies relaxed noninterference if for any integrity level $\ell \in \mathcal{L}$, any two ℓ-equivalent memory states $\mu_0 \approx_\ell \mu_0'$, we have:*

$$\langle \mathcal{H}, \mu_0 \rangle \hookrightarrow \tau \wedge \langle \mathcal{H}, \mu_0' \rangle \hookrightarrow \tau' \wedge$$

Case I:

$$(|\tau| = 0 \wedge |\tau'| > 0) \implies \mathcal{R}(\tau) \neq \mathtt{SUCCESS}$$

And Case II:

$$\begin{gathered}\forall i, s, \vec{v}.\ (\mathtt{call}_i(s, \vec{v}) \in \tau \wedge \mathtt{call}_i(s, \vec{v'}) \in \tau') \\ \implies (\forall 1 \leq j \leq s_a.\, \ell \sqsubseteq \mathcal{I}(s, j) \implies \vec{v}[j] = \vec{v'}[j])\end{gathered}$$

Here, Case I allows trace τ to bypass all sink calls when the authorization fails (i.e., an error code is returned). In other words, it rules out the *unsafe* scenario of bypassing sink calls: when untrusted input leads to a trace that bypasses all sink calls that appear in any other trace ($|\tau| = 0 \wedge |\tau'| > 0$) and the hook function $\mathcal{H}$ returns success ($\mathcal{R}(\tau) = \mathtt{SUCCESS}$). Note that the symmetric case (where $|\tau| > 0$ and $|\tau'| = 0$) is also ruled out when we flip μ_0 and μ_0' (and hence τ and τ') in the for-all quantifier in the definition.

Case II captures the relaxation on noninterference: if a sink call (e.g., the one at Line 7 in Figure 5) is found exclusively in one of the two traces, no noninterference check is needed, allowing the reference monitor to choose the authorization queries to make based on the operation requests. Therefore, the example shown in Figure 5 that violates Definition V.1 in fact satisfies Definition V.2. Note that the relaxed noninterference still checks the matching sink calls on both traces to ensure that none of their parameters are tampered (the last line in Case II). For example, a low-integrity value cannot be used as the first parameter of sink function `aa_ptrace` since for the traces that do reach line 7, such a low-integrity value might vary and hence, fail the check at the last line in Case II.

We note that while relaxed noninterference reduces spurious implicit flow violations of standard noninterference, the remaining violations are not necessarily errors or vulnerabilities. For example, a Case I violation that bypasses all sink calls and returns success might be intentional and correct from a functionality perspective. For example, the hook function in Figure 6 checks if `profile` is confined or not at line 7. When it is unconfined, it directly returns 0 without calling the sink function `aa_task_setrlimit`. This code violates Case I but its behavior is functionally correct. However, we choose to use Case I since (1) "functional correctness" is hard to formalize, and (2) each violation of Case I represents a potential risk that worth being reported for further analysis. In Section VI-D, we introduce simple *bypass endorsers* to further reduce the number of cases that violates relaxed noninterference.

In theory, relaxed noninterference might remove errors or vulnerabilities that does impose integrity risk. However, we believe that this is very unlikely since only certain implicit flows that only affect the sequence of sink function calls are removed; all explicit flows and other forms of implicit flows are still being checked. Moreover, we are not aware of any known errors or vulnerabilities being removed.

```
static int apparmor_task_setrlimit(
    struct task_struct *task, unsigned int resource,
    struct rlimit *new_rlim) {
  struct aa_profile *profile = __aa_current_profile();
  int error = 0;

  if (!unconfined(profile))
    error = aa_task_setrlimit(profile, task, resource,
        new_rlim);
  return error;
}
```

Fig. 6. Source code for `apparmor_task_setrlimit`.

D. Information Flow Analysis

To analyze whether a hook function satisfies relaxed noninterference (Definition V.2) or not, we follow an existing static information flow analysis [41] that collects information flow *constraints* of the form $\mathcal{E}_1 \sqsubseteq \mathcal{E}_2$ where $\mathcal{E}_1$ and $\mathcal{E}_2$ (called *constraint elements*) can be either an integrity level $\ell \in \mathcal{L}$ or a constraint variable to be solved. Intuitively, constraints state the information flow restrictions w.r.t. the partial ordering relation of lattice $\mathcal{L}$.

Given source and sink labeling functions $\mathcal{S}$ and $\mathcal{I}$, we assign a hook function's input level to the one as specified in $\mathcal{S}$ and assign distinguished constraint constant to each parameter in each sink function call as specified by $\mathcal{I}$. By the soundness result of [41], the program being analyzed satisfies *strict noninterference* (Definition V.1) if the corresponding set of constraints is solvable (e.g., by using a Rehof-Mogensen solver [42]).

To close the gap between strict and relaxed noninterference, we first note that Case I is relatively easy to identify on a control-flow graph with a constant-propagation analysis. For Case II, we note that no check is needed when different sequences of sink function calls occur on two traces. Hence, our tamper analysis utilizes the control-flow graph of source code, and avoids tainting a program-counter (*pc*) label of a branching instruction when all of its branches lead to different sequences of sink function calls. For instance, the branch at line 4 of Figure 5 leads to two control flows: one without any call to `aa_ptrace`, and one with one call to `aa_ptrace`. Hence, *pc* label after line 4 remains *untainted* by the label of `error`, which is tampered by `parent` at line 3. Consequently, the tamper analysis successfully reasons that there is no illegal information flow from `parent` to the parameters of `aa_ptrace`. So it accepts the program.

E. Constructing Information Flow Problems

TALISMAN constructs information flow problems to perform tamper analysis for hook functions. An information flow problem consists of a set of sources with their respective labels and a set of sinks with their respective labels, where the labels are drawn from the integrity lattice defined in Section VI-A.

To construct information flow problems for the above analysis, we need to identify and label the sources that may affect the value of the authorization query arguments at the sinks, which are known for each reference monitor, as described in Section VII. To do that, we employ a backward static taint analysis from each of the sink function arguments on a Program Dependence Graph (PDG) representation [43] of the hook functions and its supporting functions.

TALISMAN's backwards taint analysis traverses PDGs starting from each sink argument until the analysis reaches either an input (i.e., a hook function argument) or an import that has no incoming data dependencies (e.g., a global variable or constant value). Inputs are assigned to labels based on their position among the arguments in the hook function. Reference monitors typically order arguments to hook functions based on their purpose. For imports, their labels depend on where they are defined (i.e., whether they are defined in a program or monitor source file). Globals and program-defined root objects that are not constants are labeled as *external* to the reference monitor. Constants are assumed to be immutable by adversaries.

F. Endorsing Information Flows

Since the task of hook functions is to transform operation requests, consisting of low-integrity inputs, into authorization queries, whose arguments must be high integrity, most hook functions cause integrity violations. However, most reference monitors use a small number of common methods for transforming inputs, so our claim is that many violations can be resolved by a small number of simple endorsers. Violations occur due to explicit flows and implicit flows, so we propose general approaches to endorse these two types of violations. We describe examples of specific information flow violations and their endorsements in LSMs in Section VI-D.

In an explicit flow violation, low-integrity data (inputs and/or imports) is used to assign values to high-integrity authorization query arguments. In some cases, explicit flows may be endorsed because reference monitors vet sensitive kernel resources at creation time and assign security identifiers (i.e., high-integrity data generated by the reference monitor for use in authorization) to these resources that should be immutable (or only modified by the reference monitor). Thus, given knowledge of how the security identifiers are assigned to resources (e.g., in one or more fields in the resource's structure), endorsers can be added to store this security data on creation and validate the integrity of this security data upon its use in authorization by identifying their accesses (i.e., reads and writes). To identify where the security identifiers used by the reference monitors in authorization is stored, we perform a parameter access analysis of automated privilege separation [44] to collect the fields accessed by the hook function and the authorization function for the input resources (i.e., including resources reachable from the input references indirectly). We then add code to compare this set of fields to the set of fields whose values are set by the reference monitor at creation time. When inputs are provided to the hook function, they are then endorsed if they match the expected values assigned on creation.

In an implicit flow violation, low-integrity data (inputs and/or imports) are used in conditions that determine the values assigned to the authorization query. While such conditions can be ad hoc, reference monitors often check low-integrity data to determine whether it matches known, constant values. In many cases, these conditions compare resource fields or operation values against constant values using bitwise or equality comparisons. When the constants are used in these simple comparisons, then we assume they endorse the implicit flows created by these conditions (i.e., by endorsing the specific input value or bit value). The specific endorsement depends

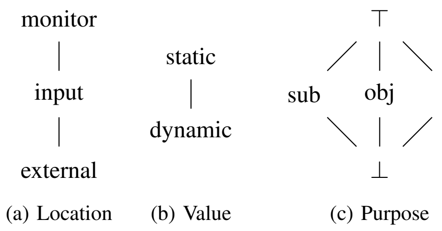


Fig. 7. Three dimensions of the integrity lattice

on whether these constants are associated with operations or resources, which we currently identify manually as described in Section VI-D.

VI. Applying Tamper Analysis

We describe the application of Talisman for tamper analysis as applied to SELinux, AppArmor, and Tomoyo LSMs. Results of this analysis are detailed in Section VIII.

A. LSM Integrity Lattice

We first define the integrity lattice that Talisman uses. One key finding is that the integrity of authorization decisions is determined from multiple perspectives, leading to a three-dimensional lattice shown in Figure 7.

Location. Based on the threat model in Section III, the LSM (*monitor*) is higher integrity than the kernel (*external*). A common pattern is for the kernel to pass authorization request inputs to LSMs through authorization APIs. To distinguish between inputs received via the LSM authorization API and other kernel data received as ad hoc imports, we define the intermediate integrity label *input*, which is higher integrity than *external*, although both are lower integrity than required for authorization queries arguments (i.e., *monitor*).

Value. LSMs are expected to predict the data used in authorization queries because they define this data or it is constant. The value dimension differentiates predictable data from other data whose values are not statically predictable, by labeling as *static* or *dynamic*, respectively, where *static* has higher integrity.

Purpose. Authorization queries enable LSMs to check whether a subject may perform an operation on an object. We use the *purpose* dimension to separate data relevant to each of these aspects of an authorization query, with corresponding levels for *subject*, *object*, and *operation*. Values derived from data with more than one purpose is labeled as $\perp$. This is the lowest integrity level since its purpose is unknown. For example, $\perp$ cannot flow to *object* since it might be derived from *subject*.

The Integrity Lattice. Combining the aforementioned three dimensions, the integrity lattice $\mathcal{L}$ follows partial ordering on each integrity dimension. That is, if the label of datum **A**, l_A meets or exceeds (i.e., has at least as high an integrity level) the label of datum **B**, l_B, then **A** is more "trusted". This is written as $l_B \sqsubseteq l_A$.

B. Constructing Information Flow Problems

To construct information flow problems using the method of Section V-E, Talisman must label sources according to the LSM lattice. Each LSM has well defined sinks, whose arguments must be $\langle \texttt{monitor}, \texttt{static}, \textit{purpose} \rangle$, where *purpose* refers to the appropriate purpose level for the argument.

Source labeling. Talisman locates source inputs and imports using the analysis described in Section V-E, and here we describe how this analysis is extended to label the sources using the integrity lattice in Section VI-A. The label for the Location dimension is determined by whether the source is a hook function parameter (*input*) or not. Globals are defined as imports (*external*). For static data (e.g., constants and macro definitions), we use the location of their corresponding definitions in the source code to identify their source. Constants or macros defined in LSM files are labeled as *monitor* and others are labeled as *external*. We describe the implementation used to find their definitions in Section VII. The label for the Value dimension is based on whether a data source is static or dynamic. This is determined based on whether the LLVM IR represents the source as a constant (*static*) or not (*dynamic*). Labeling the *purpose* dimension is done using the patterns implied by the hook APIs. Resources and operations are passed as arguments, while the process is typically retrieved via well-known functions or macros.

C. Information-Flow Analysis for LSMs

In many LSM hook functions, a single data structure holds information for multiple purposes in separate fields. To precisely analyze LSMs, we need to differentiate these fields in a data structure and properly handle union types.

Figure 8 shows a concise layout of generic data structure `tomoyo_request_info`, used for storing authorization queries in Tomoyo. This data structure is commonly created in hook functions where a subset of its fields are used in authorization query processing. To allow precise information flow analysis, it is important to enable field-sensitivity, i.e., labeling data at the granularity of data structure fields and array indices. To do so, we use the type information associated with data structures and arrays to precisely track the security label of each memory block within a data structure or array. The information flow constraints are created based on this fine-grained labeling, granting us the ability to differentiate information flows on a field granularity. When type information is unavailable (e.g., in the presence of aliasing and generic (void) pointers), we conservatively treat the target in a field-insensitive way, retaining the soundness of the analysis.

`tomoyo_request_info` in Figure 8 also uses unions to store object and operation information in its `param` field with nine variants, each with different memory layouts and sizes. A naive analysis loses field-sensitivity for unions, as a union variant's field might share memory with another variant's fields with different integrity labels. Hence, `param` in Tomoyo would have the least integrity label among all of its variants (i.e., $\perp$), introducing many false positives. To tackle the challenge, we observe that each hook function only uses one variant of union type. Hence, it is possible to gain field-sensitivity in a per-function analysis. More specifically, we leverage the type casting instruction on the union value to extract the actual variant being used. Note that the byte-wise information analysis is still sound when multiple variants of a union type are used, since each byte is constrained multiple times when multiple casts exist.

```
struct tomoyo_request_info {
  struct tomoyo_obj_info *obj;
  struct tomoyo_execve *ee;
  struct tomoyo_domain_info *domain;
  union {
    struct {
      const struct tomoyo_path_info *filename;
      const struct tomoyo_path_info *match_path;
      u8 operation;
    } path;
    ...
    struct {
      const struct tomoyo_path_info *type;
      const struct tomoyo_path_info *dir;
      const struct tomoyo_path_info *dev;
      unsigned long flags;
      int need_dev;
    } mount;
    ...
  } param;
  struct tomoyo_acl_info *matched_acl;
  u8 param_type;
  ...
};
```

Fig. 8. Layout of `struct tomoyo_request_info`

D. Endorsing Information Flows

We examine two cases based on the approaches described in Section V-F for an explicit flow and an implicit flow.

Endorsing Explicit Flows Dynamically: Security Identifiers. We use the function `selinux_file_open` from SELinux shown in Figure 9 as an example of endorsing subject and object security identifiers. We specify that the inode and task security structures are stored at `file→f_inode→i_security` and `cred→security`, respectively. Applying the parameter access analysis [44] (see Section V-F) identifies that only the `sid` and `sclass` fields of the inode's security structure (`isec`) are accessed within the authorization function (i.e., since they are the only fields passed in line 47), so only these fields of the inode security structure require endorsement. For the task, the `sid` and `create_sid` fields of the task's security structure (from the `cred` resource) must be endorsed. The calls to the task and inode endorsers are on lines 12 and 16, respectively, of Figure 9. Note that the endorsement code on line 24 adds the file security state for the newly opened file for later endorsement.

To endorse the expected (i.e., LSM-defined) values of the kernel resources, the task and inode, we must record the mapping between those resources and the assigned values of the fields used in authorization (e.g., `sid` and `sclass` for the inode) produced by the LSM at resource creation time. Thus, we need to uniquely identify resources to ensure that we checked the expected values for those resources. For tasks, their identity is determined by their thread ID. For inodes, they are uniquely identified by a combination of device number (`inode→i_rdev`) and inode number (`inode→i_ino`). We specify these identifying sources manually for each resource type. However, there are only ten distinct kernel resource types used in authorization for the hooks we have evaluated. Finally, endorsers retrieve the values of the security identifier fields assigned for each relevant resource based on its identity (i.e., type and values of its uniquely identifying fields); if the values of the security identifier fields assigned for this resource matches those in the security identifier being used, the endorsement passes.

```
1   static int selinux_file_open(struct file *file,
2         const struct cred *cred) {
3     struct file_security_struct *fsec;
4     struct inode_security_struct *isec;
5
6     fsec = file->f_security;
7     isec = file_inode(file)->i_security;
8
9     /* Endorse: verify task */
10    __u64 val;
11    val = EXX_VALUE_SELINUX_TASK(tsec);
12    exx_verify(&exx_se_task, EXX_KEY_TASK(get_current()),
          &val, sizeof(val));
13
14    /* Endorse: verify inode */
15    val = EXX_VALUE_SELINUX_INODE(isec);
16    exx_verify(&exx_se_inode, EXX_KEY_INODE(isec->inode),
          &val, sizeof(val));
17
18    fsec->isid = isec->sid;
19    fsec->pseqno = avc_policy_seqno();
20
21    /* Endorse: add file */
22    void *dup = exx_dup((void *) fsec, sizeof(*fsec));
23    if (dup)
24      exx_add(&exx_se_file, EXX_KEY_FILE(file), dup,
            sizeof(*fsec));
25
26    return file_path_has_perm(cred, file, open_file_to_av(
          file));
27  }
28
29  static int file_path_has_perm(const struct cred *cred,
30        struct file *file, u32 av) {
31    return inode_has_perm(cred, file_inode(file), av, &ad)
          ;
32  }
33
34  static int inode_has_perm(const struct cred *cred,
35        struct inode *inode, u32 perms,
36        struct common_audit_data *adp) {
37    struct inode_security_struct *isec;
38    u32 sid;
39
40    validate_creds(cred);
41    if (unlikely(IS_PRIVATE(inode)))
42      return 0;
43
44    sid = cred_sid(cred);
45    isec = inode->i_security;
46
47    return avc_has_perm(sid, isec->sid, isec->sclass,
          perms, adp);
48  }
```

Fig. 9. Hook function `selinux_file_open`, including the endorsement of task and inode's security state as well as recording endorsement information for file's security state (in red). The endorsement data for the task is added in the `selinux_bprm_committed_cred()` hook and for inode it is added in the `selinux_d_instantiate()` hook.

Endorsing Implicit Flows Statically: Authorization Bypass. LSMs allow bypass of an authorization function under some conditions. For common cases, we manually identify and assess the expected conditions. One example is the `unlikely(IS_PRIVATE(inode))` check in Figure 9 (line 41), which checks if the filesystem-internal bit in an inode flag is set, which results in returning early and falling back on the kernel for authorization. Since these checks all compare low-integrity values to constants, we add these endorsers to the information flow model for analysis using the static analysis approach in Section V-D, so there is no runtime overhead.

Other cases of bypass often involve checks for errors in the input (e.g., invalid operations or null pointers). We find it unusual that these cases sometimes return 0 (i.e., "authorized"). But further inspection finds that the kernel, instead of the LSMs, handles these suspicious cases and denies access.

TABLE I. Hook Verification results including the "Hooks Verified" and unverified under "Hooks Remaining" - The "Hooks with Integrity Violations" shows the number of hooks that failed for each Case I and Case II cause (e.g., for each argument). A hook may fail in multiple ways, so the sum of hooks across all four causes is greater than the number of "Hooks Remaining".

	Hook Verification				Hooks with Integrity Violations			
	Total Hooks	Hooks Analyzed	Hooks Verified	Hooks Remaining	Return 0 (Case I)	Subject (Case II)	Object (Case II)	Operation (Case II)
SELinux	178	102	63	39	29	23	31	17
Tomoyo	28	23	0	23	23	5	23	4
AppArmor	34	20	0	20	12	0	20	0

Hence, these unusual cases are also endorsed. However, we still believe that a better approach is to assign specific error codes for their corresponding scenarios. Doing so will increase the transparency of the authorization process by removing ambiguity of the error code 0, and in turn, make the whole system more robust.

VII. Implementation

We implemented Talisman in C++ using the LLVM framework, consisting of the following main components.

Information Flow Analysis. We implement the information flow analysis on top of PIDGIN [41], which uses DSA [45] as its point-to analysis. Our extension to handle field-sensitivity, union types and so on contains ~13K LoC. The interprocedural and context-sensitive analysis generates information flow constraints which has a straightforward graph representation. All paths from sources to sinks are checked for potential gaps (i.e., the source has lower or incompatible integrity label).

Backwards Taint Source Identifier. The backwards taint analysis is implemented in C++, consisting of ~6K LoC as an LLVM 9.0 optimization pass and it is built on a PDG generated by PtrSplit [46], which is a field-sensitive but context-insensitive analysis. Once the PDG has been constructed, taint analysis is performed over the PDG as a simple graph-based search algorithm to traverse data- and control-flow paths through the program. Once a data source is identified (via inspecting LLVM instruction types), that source is labeled based on metadata found during the taint analysis and data computed in the Clang front-end analysis.

Constant Value Analyzer. We implemented a front-end Clang analysis, consisting of ~350 LoC, over the Clang 9.0 abstract syntax tree (AST), which generates a list of definitions and uses for globally defined integers and enumerations. We leverage the Clang 9.0 pre-processor tool *pp-trace* to reconstruct macro metadata and create a list of definitions and uses for all macros within a given source code package. Metadata regarding constants is used by the taint-based source identification tool to label data accordingly.

Dynamic Endorser Placement. We prototyped automation for endorser placement for verifying values. This analysis was implemented with ~700 LoC as another LLVM 9.0 optimization pass along with ~100 LoC of Python scripting to bridge the output from the information flow analysis with the dynamic endorser placement. This analysis leveraged the parameter access analysis of automated privilege separation [44] to determine the fields to verify against expected values. In order to explore some of the performance implications of endorsement, as described in Section VIII-F, we found that we needed to explore more options to assess their performance impact. As a result, the endorsement code evaluated in Section VIII-F is implemented manually.

VIII. Evaluation

Talisman was evaluated on three popular LSMs, namely SELinux [31], Tomoyo [32] and AppArmor [30]. We perform the analysis leveraging the LLVM toolchain, which is applied to the Linux 3.19[5]. Although this kernel version was released in 2015, the changes to LSM hook functions are not substantial. Comparing version 3.19 with a modern LTS version (5.15.119), among the 102, 23 and 20 analyzed hook functions across SELinux, Tomoyo and AppArmor, only 18, 2 and 7 functions underwent any substantial structural change in their logic, respectively. The remaining functions either stayed intact or only experienced minor adjustments such as modifications due to an API change, which we define manually anyway. We leave migrating the outdated toolchain in order to analyze modern Linux kernels as future work.

A. Hook Verification Results

Table I shows the results of the Talisman hook verification for the three LSMs. There are 173, 28 and 34 hook functions in total for SELinux, Tomoyo and AppArmor, respectively. Talisman was evaluated on 102, 23 and 20 of them, respectively, omitting only the hooks that lack any authorization function call. After both information flow analysis and expected endorsement, the *Hooks Verified* column shows that 63 hooks were verified for SELinux, but none were verified successfully for Tomoyo or AppArmor because they use pathnames, which creates a vulnerability described in Section VIII-C. The *Hooks Remaining* column lists the number of hook functions whose validation requires manual effort, possibly including code changes.

The columns for Hooks with Integrity Violations of Table I show the number of hooks with Case I and Case II integrity violations, with respect to Definition V.2. Case II violations are further distinguished by the specific authorization query arguments that have an integrity violation (i.e., subject, object, operation). Note that a hook may have an integrity violation for multiple sink calls and/or multiple arguments to each sink call, which is why the sum of the number of hooks with integrity violations across these four columns is greater than the number of hooks remaining to be verified.

For Case I violations, the number of hooks that have statements that "return 0" (i.e., authorized) without invoking

[5]The main reason for analyzing a relatively old version is the limited compatibility of the points-to analysis used by Talisman. We implemented Talisman on top of PIDGIN [41], which uses DSA [45] as its points-to analysis. Unfortunately, DSA is only supported up to LLVM 3.7, which prevents us from analyzing newer kernel versions directly.

TABLE II. ARGUMENT VERIFICATION RESULTS - THE NUMBER OF ARGUMENTS WITH INTEGRITY VIOLATIONS, WITH THE INTEGRITY VIOLATIONS LISTED BY CATEGORY. THREE CATEGORIES, NAMELY SUBJECT LOOKUP, OBJECT LOOKUP (SELINUX ONLY), AND LABEL CREATION ARE EXPECTED ENDORSEMENTS. THE REMAINDER CURRENTLY REQUIRE MANUAL ENDORSEMENT AND/OR CODE REFACTORING.

	Sink Arg Verification				Args with Violations by Category						
	Sink Args	Args Verified	Args Endorsed	Args Remaining	Subject Lookup	Object Lookup	Label Creation	Op Use	Subject Metadata	Multiple Authorizations	False Positives
SELinux	282	243	188	39	0	0	0	17	7	7	8
Tomoyo	33	13	11	20	0	12	0	4	0	4	0
AppArmor	36	25	12	11	0	11	0	0	0	0	0

any authorization query are shown. A significant fraction of the authorization hooks for each LSM exhibit this violation, requiring additional work to validate that the interaction between the LSM hook functions and the kernel correctly respond to errors detected by the hook function even though they return a successful authorization code. Our manual inspection has not revealed any vulnerabilities caused by this coding practice, but we discuss remediation of these cases in Section VI-D.

For Case II violations, TALISMAN detects the number of hooks in which subjects, objects, and/or operations cannot be verified to be high integrity, which are shown in the last three columns of Table I. For subjects and operations, SELinux hook functions have the most hooks with remaining integrity violations, because their hook functions are more complex, as we discuss in Section VIII-B. For objects, all LSMs have integrity violations, although every Tomoyo and AppArmor hook function fails, as described below.

B. Argument Verification Results

Table II shows the impact of TALISMAN's verification at a per argument granularity for the authorization queries. Recall that we require all arguments passed to authorization queries (sinks) are high integrity per Case II of Definition V.2. The left-hand side shows the number of arguments that are verified, including those that can be endorsed, 80% overall and 39-86% across LSMs. Note that the number of argument violations can be less than the number of hook violations because some functions are used in multiple hooks. The right-hand side shows the argument violations by category of causes. We determined these causes manually based on study of the code. There are six such causes, plus false positives, detailed below.

Subject Lookup. This refers to violations due to looking up the subject's security identifiers (e.g., ID number or namespace) from the kernel. Violations from this category can be endorsed, which we discuss in Section VI-D.

Object Lookup. This relates to violations due to looking up the object identifiers (e.g., ID numbers or pathnames). We can endorse violations based on numeric object identifiers (SELinux only), which we also discuss further in Section VI-D. The violations for *Object Lookup* in Table II for AppArmor and Tomoyo result from the vulnerabilities described in Section VIII-C.

Label Creation. This refers to cases where objects are labeled upon creation. SELinux (primarily) uses subject identifier information to generate object identifiers when objects are created. This approach simplifies Object Lookup endorsement, as described in VI-D, but raises an information flow violation on the *purpose* dimension (i.e., a subject-to-object information flow). These cases are endorsed by validating that the ID assigned to the newly created object matches the value stored in the subject's `task_security_struct`.

Op Use. This covers the violations caused by an explicit data flow to the operation argument of the authorization function from a low integrity source. Endorsing these cases requires knowing the acceptable operation values for a given hook function, which in most cases requires manual effort. However, once known, these values can be checked statically. We provide an example of this scenario in a case study in Appendix A-C.

Subject Metadata. This refers to cases where SELinux uses subject metadata in capability hooks (e.g., uses `CAP_MAC_ADMIN`) that creates an information flow from the subject to the object being checked. In these cases, a subject security ID is being used in place of an object. The subject being passed as an object could be the same subject making the request or another subject. In either case, we must determine manually which subject label matches the subject label used as an object in the authorization function to endorse.

Multiple Authorization. This covers cases where some SELinux and Tomoyo authorization functions perform multiple types of authorization within one hook function, where an object in one authorization may be used as a subject in another authorization. This causes information flow violations in the *purpose* dimension because we have to map the input subjects and objects among multiple authorizations. Identifying the specific subjects and objects for the individual authorizations must be determined manually for each of these cases.

False Positives. These are due to imprecision in the static analysis. In these cases, multiple separate data structures are merged into a single memory block in the points-to analysis employed by TALISMAN, leading to subject and object data to appear to flow to each other.

These categories describe the primary causes of integrity violations that we observed through manual investigation of the violations. The misuse of pathnames in AppArmor and Tomoyo in *Object Lookup* is the main cause of those verification failures.

As described in Section VI-D, integrity violations that involve retrieving LSM-generated data, such as security identifiers for subjects (*Subject Lookup*) and objects (*Object Lookup*), can be endorsed by recording security identifiers on creation and retrieving that data when needed. Label assignment is also of a well-defined style in LSMs, so these can be endorsed as well (*Label Creation*). As a result, all the integrity violations for *Subject Lookup*, *Object Lookup*, and *Label Creation*, which retrieve such security identifiers for subjects and objects (for SELinux only) do not result in violations in Table II. However, we find that other categories

TABLE III. The number of gaps for Case II violations involving implicit flows when strict and relaxed noninterference are applied - For each LSM and violation dimension, two different counts are compared (i.e., strict noninterference/relaxed noninterference). Substantial reduction of violations can be observed with relaxed noninterference.

	`sub` ↓ `obj`	`sub` ↓ `op`	`obj` ↓ `sub`	`obj` ↓ `op`	`op` ↓ `sub`	`op` ↓ `obj`	`dynamic` ↓ `static`	`input` ↓ `monitor`	`external` ↓ `monitor`	% of reduction
SELinux	84/15	18/0	92/31	15/9	295/124	413/169	51/28	298/134	651/295	58.0%
Tomoyo	53/0	48/0	14/4	81/8	27/3	69/16	149/11	216/19	113/12	90.5%
AppArmor	28/0	1/0	34/0	3/1	9/0	45/0	9/7	46/6	99/14	89.8%

are too ad hoc to expect endorsement at present. Fortunately, the number of arguments impacted by the these cases is modest.

One case of interest is *Multiple Authorizations*, where a single hook authorizes multiple operations. Examples of such hook functions are `selinux_msg_queue_msgsend` in SELinux and `tomoyo_socket_connect` for Tomoyo. The former hook function authorizes whether a process can send a message to a queue and whether a message may be put in the queue. Considering multiple inputs interacting with subjects and objects, integrity violations result. However, we find that these separate authorization tasks could be factored into independent hooks that could be run in parallel (i.e., akin to LSM stacking [47]), which would enable verification of the construction of each of the independent authorization queries. Perhaps multiple LSM authorization queries could be coded independently and integrated safely into a single, efficient implementation to enable maintainable tamper analysis.

C. Object Lookup Vulnerabilities

We encountered an issue with Object Lookup endorsement for AppArmor and Tomoyo, which both use pathnames as security identifiers for objects in authorization. The problem is that the file pathnames checked against the access control policy by AppArmor and Tomoyo may differ from the file pathnames used by the file system to retrieve the file, which may result in unauthorized access in general. We have reported this issue to the Linux Ext4, AppArmor, and Tomoyo maintainers. These vulnerabilities may not have been recognized without tamper analysis.

Recent work found that name canonicalization differences among file systems can lead to vulnerabilities when copying files from one file system to another [35]. Both AppArmor and Tomoyo ignore the canonicalization rules of the underlying file systems, which can lead to incorrect policy enforcement. For example, AppArmor permits the specification of "deny" rules, so we created a file named `config` on a case-insensitive directory in an Ext4 file system and added a rule to deny access to `config`. We then submitted a file open request for a file named `CONFIG` in the same directory. AppArmor will allow access to `CONFIG` as the name differs from `config`, but the Ext4 file system canonicalizes `CONFIG` in that case-insensitive directory to `config`, allowing access to that file when access should be denied. As a result, we are unable to endorse the violations until the AppArmor and Tomoyo canonicalizations are aligned with or directly uses the canonicalization of the physical file system. Fixing this problem is non-trivial because Ext4 now allows case-sensitivity to be changed dynamically.

The TALISMAN information flow analysis finds information flow integrity violations that cannot be addressed by TALISMAN's endorsers. In examining the remaining violations, we found that some violations could allow unauthorized access (i.e., are vulnerabilities) because AppArmor and Tomoyo ignore case (in)sensitivity. Once it became clear that path canonicalization in file systems differs from AppArmor and Tomoyo, the vulnerability could be identified. Using pathnames in authorization has long been a controversial topic [48]. At a minimum, keeping the LSMs' use of pathnames consistent with the file system is required.

D. Impact of Relaxed Noninterference

To evaluate the effectiveness of relaxed noninterference, we compare the number of violations involving implicit flows (i.e., an integrity violation whose information flow path contains at least one implicit flow edge) when strict and relaxed noninterference are applied. The result is shown in Table III, where each cell shows the number of violations under strict and relaxed noninterference, respectively. Note that we count only the integrity violations where implicit flows are involved, as others are unaffected by the noninterference definition.

We observe that relaxed noninterference significantly reduces the number of violations, by ~90% for both Tomoyo and AppArmor, and by ~60% for SELinux. Thanks to relaxed noninterference, the otherwise prohibitively high false positive rate is significantly reduced. SELinux saw a lower reduction rate compared to the others. The primary reason is similar to our observation in Section VIII-A: SELinux uses a coding pattern with nested branches followed by many authorization query calls. This pattern introduces many implicit flows that cannot be removed by relaxed noninterference.

It is worth emphasizing that most of the reported violations are true positives with regard to relaxed noninterference. As acknowledged in Table II, there are only 8 false positives (all from SELinux), at the granularity of arguments, out of the 70 remaining positives. Violations of relaxed noninterference do not necessarily imply exploitable vulnerabilities. Rather, they are potential vulnerabilities that entail further analysis (i.e., endorsement). Even though endorsement addresses most potential vulnerabilities, they remain true positives as they pose potential risks.

E. Impact of Field-Sensitivity

In Section VI-C, several improvements of static analysis are introduced to support fine-grained field-sensitivity in order to reduce false positive rates. Next, we investigate the effectiveness of those improvements by comparing the number of violations reported by the baseline analysis (i.e., the implementation in [41]) and TALISMAN. Since the baseline analysis lacks support for tainting constants and has minor

TABLE IV. The number of gaps for Case II violations between the baseline analysis and the field-sensitive analysis - For each LSM and violation dimension, two different counts are compared (i.e., baseline/field-sensitive). Substantial reduction of violations can be observed with field-sensitivity enabled.

	sub $\downarrow$ obj	sub $\downarrow$ op	obj $\downarrow$ sub	obj $\downarrow$ op	op $\downarrow$ sub	op $\downarrow$ obj	dynamic $\downarrow$ static	input $\downarrow$ monitor	external $\downarrow$ monitor	% of reduction
SELinux	270/55	4/0	203/18	6/3	13/0	16/0	18/11	515/231	357/177	64.7%
Tomoyo	54/0	48/0	85/0	134/0	44/0	74/0	227/24	448/62	149/33	90.6%
AppArmor	58/0	0/0	38/6	0/0	3/0	3/0	2/2	110/54	66/29	67.5%

TABLE V. File system latency benchmarks in LMBench 3.0 - Benchmarks compare baseline and endorsed version to compute performance overhead.

	read	write	stat	fstat	open/close
SELinux (Same)					
Base	0.1081	0.1055	0.4350	0.1083	0.9752
Endorsed	0.1397	0.1408	0.4565	0.1328	0.9903
Overhead (μs)	0.0316	0.0353	0.0216	0.0245	0.0151
Overhead (%)	29.19%	33.48%	4.96%	22.59%	1.55%
#bytes / #fields	16 / 4	16 / 4	16 / 4	16 / 4	16 / 4
SELinux (All)					
Base	0.1081	0.1055	0.4350	0.1083	0.9752
Endorsed	0.1618	0.1624	0.6015	0.1328	1.2424
Overhead (μs)	0.0537	0.0569	0.1666	0.0245	0.2672
Overhead (%)	49.68%	53.93%	38.29%	22.59%	27.40%
#bytes / #fields	32 / 8	32 / 8	16* / 4*	16 / 4	32* / 8*
AppArmor					
Base	0.1112	0.1099	0.5391	0.2512	1.0784
Endorsed	0.1295	0.1283	0.6422	0.3603	1.2171
Overhead (μs)	0.0184	0.0184	0.1030	0.1092	0.1387
Overhead (%)	16.51%	16.77%	19.11%	43.46%	12.86%
#bytes / #fields	128+ / 14	128+ / 14	128+ / 14	128+ / 14	128+ / 14
Tomoyo					
Base	0.1036	0.1021	1.6543	1.3478	2.0849
Endorsed	0.1041	0.1028	1.7824	1.4751	2.1913
Overhead (μs)	0.0005	0.0006	0.1281	0.1273	0.1064
Overhead (%)	0.53%	0.61%	7.74%	9.45%	5.10%
#bytes / #fields	- / -	- / -	56+ / 11	56+ / 11	56+ / 11

SELinux (Same) and SELinux (All) — Since AppArmor and Tomoyo only support a subset of the LSM file authorization hooks, we compare the *same* hooks (i.e., those supported by other LSMs) and *all* hooks used by SELinux.
* — SELinux (All) must endorse the security metadata for every inode accessed in `stat` and `open`, adding 8 bytes and 2 fields per inode.
+ — Tomoyo and AppArmor additionally must endorse variable length pathnames, ranging from 5–108 and 1–192 bytes for AppArmor and Tomoyo, respectively.
read/write — Note that Tomoyo does not authorize read/write and AppArmor often only authorizes the task, only requiring endorsement of 13 fields and 96 bytes.

flaws in capturing some implicit flows, we disabled these two features for a fair comparison. The result is shown in Table IV.

We observe that across all three LSMs, the improvements in static analysis reduce the number of reported violations in Tomoyo by 90% and those in SELinux and AppArmor by about 65%. Since both the baseline and TALISMAN are sound, the reductions remove false positives in the baseline implementation. For the higher reduction rate in Tomoyo, the primary reason is that Tomoyo, as shown in Figure 8, uses complex data structures with nested unions to organize all three components of the authorization request and pass the single data structure into its authorization functions. Our fine-grained field-sensitivity handles this scenario well by tainting only the corresponding fields with high accuracy while maintaining soundness.

F. Performance Overhead

We implemented run-time endorsers across the three LSMs and evaluated the performance with endorsement on file system latency benchmarks of LMBench 3.0 on a server with Intel i5-7400@3.0 GHz with 16 GB RAM. For each LSM, we gather measurements for a baseline system without any endorsers and a TALISMAN-modified system with endorsement for subjects and objects dynamically, added as described in Section VI-D. We focus on the impact of endorsement on the system call latency for file system calls, including open, stat, and fstat, which invoke LSM hook functions to authorize access, which use endorsers in the TALISMAN version. Specifically, the read and write benchmarks call the hook function for `file_permission`, the fstat and stat benchmarks call the hook function for `inode_getattr`, and the stat and open benchmarks call the hook function for `file_open`, for which we measure the endorsement overhead.

The detailed results are shown in Table V, which shows the overhead and the number of bytes and fields endorsed for each benchmark. First, we compare SELinux (Same), AppArmor, and Tomoyo when utilizing the same set of LSM hook functions. Note that the authorization implementations differ significantly, such that the base latencies vary significantly among LSMs. For example, the base latency for `fstat` is more than 10X greater for Tomoyo than for SELinux. Thus, we find it useful to compare the raw overhead in μs, seeing that SELinux (Same) has the lowest overhead latency for `stat`, `fstat`, and `open/close`, mainly due to the fact that AppArmor and Tomoyo must endorse pathnames, which may be up to 192 bytes long. Tomoyo and AppArmor have lower latencies for `read` and `write` because they do not authorize those system calls or only authorize a small fraction, respectively.

However, SELinux authorizes paths used to open a file by authorizing access to each inode accessed in resolving a pathname, rather than by checking the pathname string. In SELinux (All), we include endorsement for `inode_permission`, which SELinux uses to authorize inodes, but is not used by the other LSMs[6]. When we include the endorsement for authorizing inodes, we find that the overhead latency for SELinux (All) exceeds that of AppArmor and Tomoyo for `stat` and `open/close`. One reason for the additional overhead is that the hook `inode_permission` is invoked on each inode, causing redundant endorsement of the subject lookup for each inode in a path. We estimate the overhead of redundant subject lookup endorsement to be as much as half of the overhead latency by removing subject lookup endorsement in `inode_permission`. Based on results via `perf`, over 90% of system calls invoke `inode_permission` no more than five times, but the maximum is 40 invocations. Second, we expect that some inodes must be endorsed frequently (e.g., the root directory inode), and we do not yet implement a

[6]SELinux also enforces the `inode_follow_link`, but the LMBench cases do not use links, which should be infrequent in general.

fast path for endorsing commonly-authorized inodes. Thus, opportunities for significant optimizations remain, which we leave as future work.

IX. Discussion

In this section, we discuss a few issues about the design and evaluation of tamper analysis in reference monitors.

One question is whether future reference monitors should store all their high integrity state themselves to enhance tamper verification, rather than on data structures maintained by the host (i.e., kernel or server) program. When the current reference monitor implementations were built in the mid-2000s, the assumption was that the host program was as trusted as the reference monitor. However, researchers now assume that the host program may be compromised and have additional defenses (e.g., KASLR/ASLR) to protect critical data from compromise even if the host program has a vulnerability. Talisman stores the mappings between subjects/objects and their security identifiers for endorsement. An advantage of isolating LSMs is that we would only need to endorse a single security object reference (8 bytes) rather than the entire security state (can be over 300 bytes) because the security object would be isolated from illicit modification in an isolated LSM.

Another question is whether tamper analysis of reference monitors is worthwhile if the host program will just do whatever it wants after authorization. We argue that verifying that a reference monitor will operate in a tamperproof manner is still a valuable guarantee, as it has been assumed for years as part of the reference monitor concept [2], [3]. However, currently, there is no mechanism deployed in the host programs that restricts them to obey the reference monitor. The DATS architecture for web applications [49] limits request processing to a restricted set of permissions. Given recent efforts in privilege separation [44], [50], restricting the permissions of host programs after authorization is future work.

In this paper, we evaluate tamper analysis on three Linux Security Modules. Given that reference monitor implementations have been applied to a variety of server applications, such as X Server [5] and Apache [7], a question is whether these should be considered in the evaluation. However, we find that these LSMs differ more significantly than the reference monitor implementations across programs. The LSMs evaluated were all developed and are all maintained by separate organizations, resulting in significant differences. The reference monitors cited above were all developed in collaboration with the SELinux community and reflect the same approach to generating and storing security identifiers and designing hook functions. Some prior work on verifying complete mediation [17], [18] examined multiple programs, but the placement of hooks in programs is much more program-specific than the implementation of hook functions, which are often reused across programs. Thus, we believe that examining multiple LSMs provides more diversity than examining multiple reference monitors developed by the same community.

To apply Talisman to reference monitors other than LSMs, there are two major sources of manual efforts. First, we need to identify the authorization functions. For each of them, the expected integrity labels of each argument need to be specified. The labeling effort may vary based on the complexity of each reference monitor. For the evaluated LSMs, the effort is quite small thanks to only 9 authorization functions in total. Second, we need to categorize and apply appropriate endorsers (when possible) to violations of relaxed noninterference, which requires a deeper understanding of the reference monitors. We observed that each violation category exhibits similar coding patterns, facilitating this task. We might also need to develop domain-specific endorsers, but the 3 generic endorsement function templates for LSMs likely can be reused.

In the future, researchers should explore tamper analysis with stronger threat models to account for other threats, such as memory errors that may tamper the reference monitor data or execution. Traditionally, reference monitors trust the program in which they are deployed, but that assumption is no longer acceptable. Modern operating systems and server programs now aim to protect security-critical code and data from tampering (e.g., via KASLR and ASLR, respectively). Researchers have also advocated *privilege separation* [12], [10], [11] as a means to isolate a component from memory errors, and recent work automates many tasks in privilege separation for applications [46] and the kernel [44], [50]. Ideally, future tamper analyses can leverage such techniques.

X. Related Work

Reference Monitor. A variety of prior work analyze to what extent production code adheres to the reference monitor concept; however, their focus is on whether a reference monitor provides complete mediation of all security-sensitive operations, rather than whether a reference monitor is tamperproof, the research question of this paper.

Muthukumaran et al. [17], [18] present methods for automatically identifying authorization hook placements for C code using the observation that all choices made by users through input must be authorized. Ganapathy et al. [14], [15], [16] rely on code-level specification of the sensitive operations to infer which hooks are necessary to guard a sensitive operation. In [20], Tan et al. rely on the assumption that mature code bases that contain security code typically have most security hooks in place. They developed a static analysis to infer sensitive operations from the current hooks and verified the code base for missing checks.

Zhang et al. [9] develop a type-based static analysis tool to verify the correctness of hook placement in LSMs, while Edwards et al. [8] leverage dynamic analysis to do the same. Son et al. [13] verify the consistency of authorization hooks in web applications. Their approach relies on common patterns to identify security sensitive objects and operations within a database, and it takes advantage of web applications that are designed around predefined roles.

Information Flow Control. We develop a static information flow analysis to validate if a reference monitor is tamperproof. While using information flow analysis to validate an integrity policy is not a new idea, to the best of our knowledge, this is the first comprehensive study of integrity issues in LSMs.

Moreover, the relaxed noninterference definition is novel. A *declassification policy* [51], [52], [53], [54], [55], [56], [57], [58], [59] weakens noninterference by deliberately releasing (i.e., declassifying) sensitive information. On the other hand, an endorsement policy weakens noninterference by deliberately tampering (i.e., endorsing) high-integrity information. Unlike well-studied declassification policies, how to weaken noninterference for data integrity is less studied in the literature. Cecchetti et al. [60] propose transparent endorsement, a mechanism for downgrading integrity while defending against adversarial exploitation. However, transparent endorsement is defined on systems where both confidentiality and integrity interplay, while our paper focuses on integrity only. It is also possible to interpret a declassification policy under the duality of confidentiality and integrity, but as discussed in Section V-B, the dual of relaxed noninterference in the confidentiality realm is inappropriate, as the absence of a sink call also leaks information via either implicit flows or side channels.

XI. Conclusions

In this paper, we present Talisman, a tamper analysis for reference monitor implementations. Talisman implements a precise field-sensitive, information-flow integrity analysis using a novel relaxed noninterference, which both greatly reduce false implicit flow violations. We evaluate Talisman on three popular Linux Security Modules: SELinux, Tomoyo, and AppArmor. For SELinux, we find that 62% of its hook functions and over 86% of its arguments to authorization queries can be verified as safe from tampering. Due to the complexity of some hook functions, some arguments to authorization queries in other hook functions cannot be verified, but Talisman identifies these cases automatically. Using Talisman, we find that Tomoyo and AppArmor use a vulnerable approach to determine the security identifier for file objects, which could be exploited. Thus, Talisman provides a valuable tamper analysis to detect and remediate issues in reference monitor implementations.

Acknowledgments

The authors would like to thank our anonymous reviewers. This research was sponsored by National Science Foundation grants CNS-1816282 and CNS-1942851. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell, "The inevitability of failure: The flawed assumption of security in modern computing environments," in *Proceedings of the 21st national information systems security conference*, vol. 10, 1998, pp. 303–314.

[2] J. P. Anderson, "Reference monitor," *Computer security technology planning study*, vol. 2, 1972.

[3] T. Jaeger, "Reference monitor." 2011.

[4] 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 Symposium (USENIX Security 02)*, 2002.

[5] "X Access Control Extension Specification," https://www.x.org/releases/X11R7.5/doc/security/XACE-Spec.html, accessed: 2022-12-1.

[6] K. Kohei, "Security enhanced postgresql," 2013.

[7] D. Walsh, "Selinux/apache." [Online]. Available: https://fedoraproject.org/wiki/SELinux/apache

[8] 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*, 2002, pp. 225–234.

[9] X. Zhang, A. Edwards, and T. Jaeger, "Using {CQUAL} for static analysis of authorization hook placement," in *11th USENIX Security Symposium (USENIX Security 02)*, 2002.

[10] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, vol. 57, no. 72, 2004.

[11] A. Berman, V. Bourassa, and E. Selberg, "Tron: Process-specific file protection for the unix operating system." in *USENIX*, 1995, pp. 165–175.

[12] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation." in *USENIX Security Symposium*, vol. 9, 2003.

[13] 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*, 2011, pp. 1069–1084.

[14] 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*, 2005, pp. 330–339.

[15] ——, "Retrofitting legacy code for authorization policy enforcement," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 15–pp.

[16] V. Ganapathy, D. King, T. Jaeger, and S. Jha, "Mining security-sensitive operations in legacy code using concept analysis," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 458–467.

[17] D. Muthukumaran, T. Jaeger, and V. Ganapathy, "Leveraging" choice" to automate authorization hook placement," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 145–156.

[18] D. Muthukumaran, N. Talele, T. Jaeger, and G. Tan, "Producing hook placements to enforce expected access control policies," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2015, pp. 178–195.

[19] S. Son, K. S. McKinley, and V. Shmatikov, "Fix me up: Repairing access-control bugs in web applications." in *NDSS*. Citeseer, 2013.

[20] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "Autoises: Automatically inferring security specification and detecting violations." in *USENIX Security Symposium*, 2008, pp. 379–394.

[21] K. L. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences," in *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

[22] D. Liu, Q. Wu, S. Ji, K. Lu, Z. Liu, J. Chen, and Q. He, "Detecting missed security operations through differential checking of object-based similar paths," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1627–1644.

[23] K. Lu, A. Pakki, and Q. Wu, "Automatically identifying security checks for detecting kernel semantic bugs," in *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part II 24*. Springer, 2019, pp. 3–25.

[24] D. King, S. Jha, D. Muthukumaran, T. Jaeger, S. Jha, and S. A. Seshia, "Automating security mediation placement," in *European Symposium on Programming*. Springer, 2010, pp. 327–344.

[25] C. Skalka, D. Darais, T. Jaeger, and F. Capobianco, "Types and abstract interpretation for authorization hook advice," in *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*. IEEE, 2020, pp. 139–152.

[26] D. Aucsmith, "Tamper resistant software: An implementation," in *International Workshop on Information Hiding*. Springer, 1996, pp. 317–333.

[27] H. Chang and M. J. Atallah, "Protecting software code by guards," in *ACM Workshop on Digital Rights Management*. Springer, 2002, pp. 160–175.

[28] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic self-checking techniques for improved tamper resistance," in *ACM Workshop on Digital Rights Management*. Springer, 2002, pp. 141–159.

[29] M. Jacob, M. H. Jakubowski, and R. Venkatesan, "Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings," in *Proceedings of the 9th workshop on Multimedia & security*, 2007, pp. 129–140.

[30] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor, "{SubDomain}: Parsimonious server security," in *14th Systems Administration Conference (LISA 2000)*, 2000.

[31] S. Smalley, C. Vance, and W. Salamon, "Implementing selinux as a linux security module," *NAI Labs Report*, vol. 1, no. 43, p. 139, 2001.

[32] T. Harada, T. Horie, and K. Tanaka, "Towards a manageable linux security," in *Linux Conference*, vol. 2005, 2005.

[33] "Apache HTTPD Server Project," https://httpd.apache.org/docs/2.4/, accessed: 2022-12-1.

[34] P. G. D. Group, Jun 2023. [Online]. Available: https://www.postgresql.org/

[35] A. Basu, J. Sampson, Z. Qian, and T. Jaeger, "Unsafe at any copy: Name collisions from mixing case sensitivities," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023.

[36] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 228–241.

[37] A. C. Myers and B. Liskov, "A decentralized model for information flow control," *ACM SIGOPS Operating Systems Review*, vol. 31, no. 5, pp. 129–142, 1997.

[38] G. Barthe, P. R. D'argenio, and T. Rezk, "Secure information flow by self-composition," *Mathematical Structures in Computer Science*, vol. 21, no. 6, pp. 1207–1252, 2011.

[39] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[40] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[41] A. Johnson, L. Waye, S. Moore, and S. Chong, "Exploring and enforcing security guarantees via program dependence graphs," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 291–302, 2015.

[42] J. Rehof and T. Mogensen, "Tractable constraints in finite semilattices," *Science of Computer Programming*, vol. 35, no. 2, pp. 191–221, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642399000118

[43] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[44] Y. Huang, V. Narayanan, D. Detweiler, K. Huang, G. Tan, T. Jaeger, and A. Burtsev, "{KSplit}: Automating device driver isolation," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 613–631.

[45] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 278–289, 2007.

[46] S. Liu, G. Tan, and T. Jaeger, "Ptrsplit: Supporting general pointers in automatic program partitioning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2359–2371.

[47] J. Edge, "Lsm stacking and the future." [Online]. Available: https://lwn.net/Articles/804906/

[48] Corbet, "The apparmor debate begins." [Online]. Available: https://lwn.net/Articles/181508/

[49] C. Hunger, L. Vilanova, C. Papamanthou, Y. Etsion, and M. Tiwari, "Dats-data containers for web applications," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 722–736.

[50] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakc," in *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, vol. 22, 2022, pp. 1–17.

[51] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 2007, pp. 207–221.

[52] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive declassification policies and modular static enforcement," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 339–353.

[53] N. Broberg and D. Sands, "Paralocks: Role-based information flow control and beyond," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010, pp. 431–444. [Online]. Available: http://doi.acm.org/10.1145/1706299.1706349

[54] A. Sabelfeld and D. Sands, "A per model of secure information flow in sequential programs," *Higher-order and symbolic computation*, vol. 14, no. 1, pp. 59–91, 2001.

[55] R. Giacobazzi and I. Mastroeni, "Abstract non-interference: Parameterizing non-interference by abstract interpretation," *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 186–197, 2004.

[56] ——, "Adjoining declassification and attack models by abstract interpretation," in *European Symposium on Programming*. Springer, 2005, pp. 295–310.

[57] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005, pp. 158–170.

[58] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *International Symposium on Software Security*. Springer, 2003, pp. 174–191.

[59] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. IEEE, 2005, pp. 255–269.

[60] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable information flow control," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1875–1891.

APPENDIX A
CASE STUDIES

```
1  static int selinux_msg_queue_msgsnd(struct msg_queue *
        msq, struct msg_msg *msg, int msqflg)
2  {
3      isec = msq->q_perm.security;
4      msec = msg->security;
5
6      /* Can this process write to the queue? */
7      rc = avc_has_perm(sid, isec->sid, SECCLASS_MSGQ,
8                MSGQ__WRITE, &ad);
9      if (!rc)
10         /* Can this process send the message */
11         rc = avc_has_perm(sid, msec->sid, SECCLASS_MSG,
12                   MSG__SEND, &ad);
13     if (!rc)
14         /* Can the message be put in the queue? */
15         rc = avc_has_perm(msec->sid, isec->sid,
              SECCLASS_MSGQ,
16                   MSGQ__ENQUEUE, &ad);
17
18     return rc;
19 }
```

Fig. 10. Source code for `selinux_msg_queue_msgsnd`

A. *selinux_msg_queue_msgsnd*

The function `selinux_msg_queue_msgsnd` governs access to create messages and place them on the queue. The code for this hook function can be found in Figure 10 and has been truncated for brevity. This function takes two objects, a message queue and corresponding message, as arguments. The authorization process performs three checks. The first check on line 7 authorizes the process to write to the queue, the second check on line 11 authorizes the process to send the message, and finally a third check on line 15 authorizes the message to be placed on the queue.

Our analysis identifies a violation on line 15, where the security field of the message is passed to the first argument of the policy checking function `avc_has_perm`. This case demonstrates an added complexity where objects are sometimes treated as subjects. Endorsing this information flow requires validating that the object security ID generated by the LSM from the subject's metadata corresponds to the subject security ID passed to the authorization function on line 15. We argue that this violation and need for endorsement could be avoided entirely by separating the authorization function and pushing the check to authorize the message to be placed on the queue into its own authorization function to preserve the integrity of sensitive data.

B. *selinux_inode_mkdir*

There are a number of SELinux hooks which are dedicated to authorizing the creation of filesystem objects. These functions include `selinux_inode_mkdir` (directory creation), `selinux_inode_mknod` (filesystem node creation), `selinux_inode_symlink` (symbolic link creation), and `selinux_inode_create` (file creation). In all cases, the function `may_create` is called.

```
1   static int may_create(struct inode *dir,
2             struct dentry *dentry,
3             u16 tclass)
4   {
5     const struct task_security_struct *tsec =
          current_security();
6     u32 sid, newsid;
7
8     rc = security_transition_sid(sid, dsec->sid, tclass,
9                 &dentry->d_name, &newsid);
10
11    rc = avc_has_perm(sid, newsid, tclass, FILE__CREATE, &
          ad);
12    if (rc)
13        return rc;
14
15    return avc_has_perm(newsid, sbsec->sid,
16                SECCLASS_FILESYSTEM,
17                FILESYSTEM__ASSOCIATE, &ad);
18  }
```

Fig. 11. Source code for `may_create`

The code snippet in Figure 11 has been truncated for readability and depicts the two final authorization checks performed. Our analysis identifies an information flow violation regarding the subjects and objects passed as input. More specifically the analysis identifies that a subject B, (`newsid`), is created using subject A's metadata and passed to a call to the policy checking function `avc_has_perm` on line 11 as the second argument, which is expected to be an object. This data is subsequently passed to another call to `avc_has_perm` on line 15 as the first argument, which is expected to be a subject.

This cross-domain interaction should be considered benign and expected for any hook authorizing operations for creating objects, as these objects require some form of ownership or subject association. As mentioned above, a number of SELinux hooks rely on `may_create`, and thus this problem affects all of them. In order to endorse that a subject can be used as an object we deploy a run-time endorser to verify that the security ID from the subject's metadata that is being used to generate the new object ID corresponds to the object security ID that is passed to the authorization function on line 15.

```
static inline u32 aa_map_file_to_perms(struct file *file
    )
{
  int flags = file->f_flags;
  u32 perms = 0;
  if (file->f_mode & FMODE_WRITE)
    perms |= MAY_WRITE;
  if (file->f_mode & FMODE_READ)
    perms |= MAY_READ;
  if ((flags & O_APPEND) && (perms & MAY_WRITE))
    perms = (perms & ~MAY_WRITE) | MAY_APPEND;
  if (flags & O_TRUNC)
    perms |= MAY_WRITE;
  if (flags & O_CREAT)
    perms |= AA_MAY_CREATE;
  return perms;
}
```

Fig. 12. Source code for `aa_map_file_to_perms`, a function used to determine file operations

C. *apparmor_file_open*

Figure 12 shows `aa_map_file_to_perms`, a function used by `apparmor_file_open`, which is an AppArmor hook function that authorizes a subject to open a file with a varying set of operations (e.g., read, write, append, etc). In this example operations that are stored in the `f_flags` and `f_mode` members of the `file` structure are compared against a variety of external constants. External constants, such as `MAY_READ`, `FMODE_WRITE`, and `O_TRUNC` are all globally defined in the kernel. This control block contributes to 19 information flow violations related to selecting the operation.

Endorsement in this example requires determining the set of expected values that are compared to the permissions in the `f_flags` and `f_mode` members of the `file` structure. The comparisons are performed using bitwise operations and require manual effort to determine the correct set of values. If all values can be vetted this can be endorsed statically. The implicit flows created by the comparisons discussed above decide which external constant is assigned to the variable `perms`. Assignments are performed through bitmasking, a common practice in the Linux kernel. Appropriate permission values need to be determined through manual effort in order to properly endorse the permissions that are passed to the authorization function.