Verifying Compliance of Trusted Programs

Sandra Rueda, Dave King and Trent Jaeger
Systems and Internet Infrastructure Security Laboratory
Department of Computer Science and Engineering
The Pennsylvania State University
{ruedarod,dhking,tjaeger}@cse.psu.edu

Abstract

In this paper, we present an approach for verifying that trusted programs correctly enforce system security goals when deployed. A trusted program is trusted to only perform *safe* operations despite have the authority to perform *unsafe* operations; for example, initialization programs, administrative programs, `root` network daemons, etc. Currently, these programs are trusted without concrete justification. The emergence of tools for building programs that guarantee policy enforcement, such as security-typed languages (STLs), and mandatory access control systems, such as user-level policy servers, finally offers a basis for justifying trust in such programs: we can determine whether these programs can be deployed in compliance with the reference monitor concept. Since program and system policies are defined independently, often using different access control models, compliance for all program deployments may be difficult to achieve in practice, however. We observe that the integrity of trusted programs must dominate the integrity of system data, and use this insight, which we call the PIDSI approach, to infer the relationship between program and system policies, enabling automated compliance verification. We find that the PIDSI approach is consistent with the SELinux reference policy for its trusted programs. As a result, trusted program policies can be designed independently of their target systems, yet still be deployed in a manner that ensures enforcement of system security goals.

1 Introduction

Every system contains a variety of trusted programs. A *trusted program* is a program that is expected to *safely* enforce the system's security goals despite being authorized to perform *unsafe* operations (i.e., operations that can potentially violate those security goals). For example, the X Window server [37] is a trusted program because enables multiple user processes to share access to the system display, and the system trusts it to prevent one user's data from being leaked to another user. A system has many such trusted processes, including those for initialization (e.g., `init` scripts), administration (e.g., software installation and maintenance), system services (e.g., windowing systems), authentication services (e.g., remote login), etc. The SELinux system [27] includes over 30 programs specifically-designated as trusted to enforce multilevel security (MLS) policies [14].

An important question is whether trusted programs actually enforce the system's security goals. Trusted programs can be complex software, and they traditionally lack any declarative access control policy governing their behavior. Of the trusted programs in SELinux, only the X server currently has an access control policy. Even in this case, the system makes no effort to verify that the X server policy corresponds to the system's policy in any way. Historically, only formal assurance has been used to verify that a trusted program enforces system security goals, but current assurance methodologies are time-consuming and manual. As a result, trusted programs are given their additional privileges without any concrete justification.

Recently, the emergence of techniques for building programs with declarative access control policies motivates us to develop an automated mechanism to verify that such programs correctly enforce security goals. Programs written in security-typed languages [23, 26, 28] or integrated with user-level policy servers [34] each include program-specific access control policies. In the former case, the successful compilation of the program proves its enforcement of an associated policy. In the latter case, the instrumentation of the program with a policy enforcement aims to ensure comprehensive enforcement of mandatory access control policies. In general, we would want such programs to enforce system security goals, in which case we say that the program *complies* with the system's security goals.

We use the classical *reference monitor concept* [1] as the basis for the program's compliance requirements[1]: (1) the program policy must enforce a policy that represents the system security goals and (2) the system policy must ensure that the program cannot be tampered. We will show that both of these problems can be cast as policy verification problems, but since program policies and system policies are written in different environments, often considering different security goals, they are not directly comparable. For example, program policy languages can differ from the system policy language. For example, the security-typed language Jif [26] uses an information flow policy based on the Decentralized Label Model [24], but the SELinux system policy uses an extended Type Enforcement policy [5] that includes multilevel security labeling [2]. Even where program policies are written for SELinux-compatible policy servers [34], the set of program labels is often distinct from the set of system labels. In prior work, verifiably-compliant programs were developed by manually joining a system policy with the program's policy and providing a mapping between the two [13]. To enable general programs to be compliant, our goal is to develop an approach by which compliant policy designs can be generated and verified automatically.

As a basis for an automated approach, we observe that trusted programs and the system data upon which it operates have distinct security requirements. For a trusted program, we must ensure that the program's components, such as its executable files, libraries, configuration, etc., are protected from tampering by untrusted programs. For the system data, the system security policy should ensure that all operations on that data satisfy the system's security goals. Since trusted programs should enforce the system's security goals, their integrity must dominate the system data's integrity. If the integrity of a trusted program is compromised, then all system data is at risk. Using the insight that *program integrity dominates system integrity*, we propose the PIDSI approach to designing program policies, where we assign trusted program objects to a higher integrity label than system data objects, resulting in a simplified program policy that enables automated compliance verification. Our experimental results justify that this assumption is consistent with the SELinux reference policy for its trusted programs. As a result, we are optimistic that trusted program policies can be designed independently of their target systems, yet still be deployed in a manner that ensures enforcement of system security goals.

After providing background and motivation for the policy compliance problem in Section 2, we detail the following novel contributions:

1. In Section 3, we define a formal model for *policy compliance problem*.
2. In Section 4, we propose an approach called *Program Integrity Dominates System Integrity* (PIDSI) where trusted programs are assigned to higher integrity labels than system data. We show that compliance program policies can be composed by relating the program policy labels to the system policy on the target system using the PIDSI approach.
3. In Section 5.1, we describe policy compliance tools that automate the proposed PIDSI approach such that a trusted program can be deployed on an existing SELinux system and we can verify enforcement of system security goals.
4. In Section 5.2, we show the trusted programs for which there are Linux packages in SELinux are compatible with the PIDSI approach with a few exceptions. We show how these can be resolved using a few types of simple policy modifications.

This work is the first that we are aware of that enables program and system security goals to be reconciled in a scalable (automated and system-independent) manner.

2 Background

The general problem is to develop an approach for building and deploying trusted programs, including their access control policies, in a manner that enables automated policy compliance verification. In the section, we specify the current mechanisms for these three steps: (1) trusted program policy construction; (2) trusted program deployment; and (3) trusted program enforcement. We will use the SELinux system as the platform for deploying trusted programs.

2.1 Program Policy Construction

There are two major approaches for constructing programs that enforce a declarative access control policy: (1) security-typed languages [26, 28, 33] (STLs) and (2) application reference monitors [22, 34]. These two approaches are quite different, but we aim to verify policy compliance for programs implemented either way.

Programs written in an STL will compile only if their information flows, determined by type inferencing, are authorized by the program's access control policy. As a result, the STL compilation guarantees, modulo bugs in the program interpreter, that the program enforces the specified policy. As an example, we consider the Jif STL. A Jif program consists of the program code plus a program policy file [12] describing a Decentralized Label Model [24] policy for the program. The Jif compiler ensures that the policy is enforced by the generated program. We would use the policy file to determine whether Jif program complies with the system security goals.

For programs constructed with application reference monitors, the program includes a reference monitor interface [1] which determines the authorization queries that must be satisfied to access program operations. The queries are submitted to a reference monitor component that may be internal or external to the program. The use of a reference monitor does not guarantee that the program policy is correctly enforced, but a manual or semi-automated evaluation of the reference monitor interface is usually performed [17]. As an example, we consider the SELinux Policy Server [34]. A program that uses the

SELinux Policy Server, loads a *policy package* containing its policy into the SELinux Policy Server. The program is implemented with its own reference monitor interface which submits authorization queries to the Policy Server. We note that the programs that use an SELinux Policy Server may share labels, such as the labels of the system data, with other programs.

As an example, we previously reimplemented one of the trusted programs in an SELinux/MLS distribution, `logrotate`, using the Jif STL [13]. `logrotate` ages logs by writing them to new files and is trusted in SELinux/MLS because it can read and write logs of multiple MLS secrecy levels. Our experience from `logrotate` is that ensuring system security goals requires the trusted program to be aware of the system's label for its data. For example, if `logrotate` accesses a log file, it should control access to the file data based on the system (e.g., SELinux) label of that file. We manually designed the `logrotate` information flow policy to use the SELinux labels and the information flows that they imply. Further, since `logrotate` variables may also originate from program-specific data, such as configurations, in addition to system files, the information flow policy had to ensure that the information flows among system data and program data was also correct. As a result, the information flow policy required a manual merge of program and system information flow requirements.

2.2 Program Deployment

We must also consider how trusted programs are deployed on systems to determine what it takes to verify compliance. In Linux, programs are delivered in packages. A package is a set of files including the executable, libraries, configuration files, etc. A package provides new files that are specific to a program, but a program may also depend on files already installed in the system (e.g., system shared libraries, such as `libc`). Some packages may also export files that other packages depend on (e.g., special libraries and infrastructure files used by multiple programs).

For a trusted program, such as `logrotate`, we expect that a Linux package would include two additional, noteworthy files: (1) the program policy and (2) the SELinux policy module[2]. The program policy is the file that contains the declarative access control policy to be enforced by the program's reference monitor or STL implementation, as described above.

In SELinux, the system policy is now composed from the policy modules. SELinux policy modules specify the contribution of the package to the overall SELinux system policy [20]. While SELinux policy modules are specific to programs, they are currently designed by expert system administrators. Our `logrotate` program policy is derived from the program's SELinux policy module, and we envision that program policies and system policy modules will be designed in a coordinated way (e.g., by program developers rather than system administrators) in the future, although this is an open issue.

An SELinux policy module consists of three components that originate from three policy source files. First, a `.te` file defines a set of new SELinux types[3] for this package. It also defines the policy rules that govern program accesses to its own resources as well as system resources. Second, a `.fc` file specifies the assignment of package files to SELinux types. Some files may use types that are local to the policy module, but others may be assigned types defined previously (e.g., system types like `etc_t` is used for files in `/etc`). A `.if` file defines a set of interfaces that specify how other modules can access objects labeled with the types defined by this module.

When a package is installed, its files are downloaded onto the system and labeled based on the specification in the `.fc` file or the default system specification. Then, the trusted program's module policy is integrated into the SELinux system policy[4], enabling the trusted program to access system objects and other programs to access the trusted program's files. There are two ways that another program can access this package's files: (1) because a package file is labeled using an existing label or (2) another module is loaded that uses this module's interface or types. As both are possible for trusted programs, we must be concerned that the SELinux system policy may permit an untrusted program to modify a trusted program's package file.

For example, the `logrotate` package includes files for its executable, configuration file, documentation, `man` pages, execution status, etc. Some of these files are assigned new SELinux types defined by the `logrotate` policy module, such as the executable (`logrotate_exec_t`) and its status file (`logrotate_var_lib_t`), whereas others are assigned existing SELinux types, such as its configuration file (`etc_t`). The `logrotate` policy module uses system interfaces to obtain access to the system data (e.g., logs), but no other processes access `logrotate` interfaces. As a result, `logrotate` is only vulnerable to tampering because some of the system-labeled files that it provides may be modified by untrusted processes.

We are also concerned that a `logrotate` process may be tampered by the system data that it uses (e.g., Biba read-down [4]). For example, `logrotate` may read logs that contain malicious data. We believe that systems and programs should provide mechanisms to protect themselves from the system data that they process. Some interesting approaches have been proposed to protect process integrity [19, 30], so we consider this an orthogonal problem that we do not explore further here.

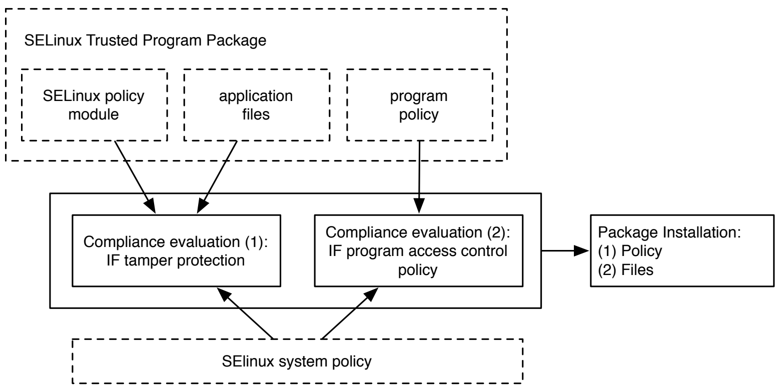

SELinux Trusted Program Package
SELinux policy module
application files
program policy
Compliance evaluation (1): IF tamper protection
Compliance evaluation (2): IF program access control policy
Package Installation: (1) Policy (2) Files
SElinux system policy

Figure 1: Deployment and Installation of a trusted package. First, we check two compliance goals: (1) the system protects the application and (2) the application enforces system goals. Second the package is installed: the policy module is integrated into the system policy and application files are installed.

2.3 Program Enforcement

To justify a system's trust, any trusted program must enforce a policy that complies with the system's security goals. The *reference monitor concept* [1] has been the guide for determining whether a system enforces its security goals, and we leverage this concept in defining compliance. A reference monitor requires three guarantees to be achieved: (1) *complete mediation* must ensure that all security-sensitive operations are authorized; (2) a reference monitor must be *tamperproof* to enforce its policy correctly; and (3) a reference monitor must be *simple enough to verify* enforcement of security goals. While the reference monitor concept is most identified with operating system security, a trusted program must also satisfy these guarantees to ensure that a system's security goal is enforced. As a result, we define that a program enforces a system's security goals if it satisfies the reference monitor guarantees in its deployment on that system.

In prior work, we developed a verification method that partially fulfilled these requirements. We developed a service, called SIESTA, that compares program policies against SELinux system policies, and only executes programs whose policies permit information flows authorized in the system policy [13]. This work considered two of the reference monitor guarantees. First, we used the SIESTA service to verify trust in the Jif STL implementation of the `logrotate` program. Since the Jif compiler guarantees enforcement of the associated program policy, this version of `logrotate` provides complete mediation, modulo the Java Virtual Machine. Second, SIESTA performs a policy analysis to ensure that the program policy complies with system security goals (i.e., the SELinux MLS policy). Compliance was defined as requiring that the `logrotate` policy only authorized an operation if the SELinux MLS policy) also permitted that operation. Thus, SIESTA is capable of verifying a program's enforcement of system security goals.

We find two limitations to this work. First, we had to construct the program access control policy relating system and program objects in an ad hoc manner. As the resultant program policy specified the union of the system and program policy requirements, it was much more complex than we envisioned. Not only is it difficult to design a compliant program access control policy, but that policy may only apply to a small number of target environments. As we discussed in Section 2.1, program policies should depend on system policies, particularly for trusted programs that we expect to enforce the system's policy, making them non-portable unless we are careful. Second, this view of compliance does not protect the trusted program from tampering. As described above, untrusted programs could obtain access to the trusted program's files after the package is installed, if the integrated SELinux system policy authorizes it. For example, if an untrusted program has write access to the `/etc` directory where configuration files are installed, as we demonstrated was possible in Section 2.2, SIESTA will not detect that such changes are possible.

In summary, Figure 1 shows that we aim to define an approach that ensures the following requirements:

- For any system deployment of a trusted program, automatically construct a program policy that is compliant with the system security goals, thus satisfying the reference monitor guarantee of being *simple enough to verify*.
- For any system deployment of a trusted program, verify, in a mostly-automated way, that the system policy does not permit tampering of the trusted program by any untrusted program, thus satisfying the reference monitor guarantee of being *tamperproof*. The typical number of verification errors must be

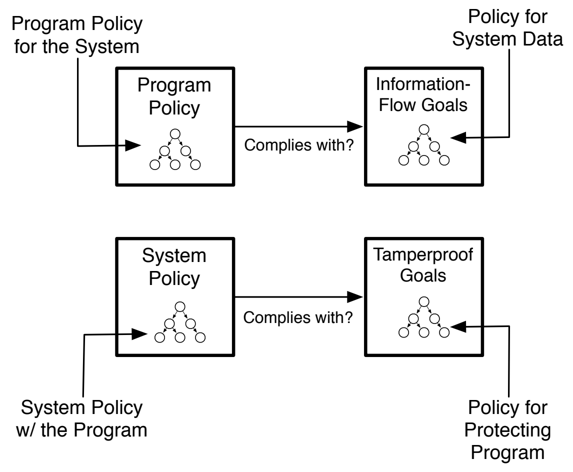


Figure 2: The two policy compliance problems: (1) verify that the *program policy* complies with the system's *information flow goals* and (2) verify that the *system policy*, including the program contribution (e.g., SELinux policy module), enforces the *tamperproofing goals* of the program.

small and there must be a set of manageable resolutions to any such errors.

In the remainder of the paper, we present a single approach that solves both of these requirements.

3 Policy Compliance

Verification of these two trusted program requirements results in the same conceptual problem, which we call *policy compliance problems*. Figure 2 shows these two problems. First, we must show that the program policy only authorizes operations permitted by the system's security goal policy. While we have shown a method by which such compliance can be tested previously [13,14], the program policy was customized manually for the system. Second, we also find that the system policy must comply with the program's tamperproof goals. That is, the system policy must not allow any operation that permits tampering the trusted program. As a result, we need to derive the tamperproof goals from the program (e.g., from the SELinux policy module).

In this section, we define the formal model for verifying policy compliance suitable for both the problems above. However, as can be seen from Figure 2, the challenge is to develop system security goals, program policies, and tamperproof goals in a mostly-automated fashion that will encourage successful compliance. The PIDSI approach in Section 4 provides such guidance.

3.1 Policy Compliance Model

We specify system-wide information-flow goals as a security lattice $\mathcal{L}$. We assume that elements of $\mathcal{L}$ have both an integrity and a confidentiality component: this is the case for both MLS labels in SELinux [11] and labels from the DLM [25]. Let $\mathsf{Integrity}(l)$ and $\mathsf{Conf}(l)$ be the integrity and confidentiality projections of a label $l \in \mathcal{L}$, respectively. Let the lattice $\mathcal{L}$ have both a top element, $\top$, and a bottom element $\bot$. We use $\mathsf{high} = \mathsf{Integrity}(\bot)$ and $\mathsf{low} = \mathsf{Integrity}(\top)$ to denote high and low integrity and write $\mathsf{high} \sqsubseteq \mathsf{low}$ to indicate that high integrity data can flow to a low integrity security label, but not the reverse.

An *information-flow graph* is a directed graph $G = (V, E)$ where V is the set of vertices, each associated with a label from a security lattice $\mathcal{L}$. We write $V(G)$ for the vertices of G and $E(G)$ for the edges of G, and for $v \in V(G)$ we write $\mathsf{Type}(v)$ for the label on the vertex v. Both subjects (e.g., processes and users) and objects (e.g., files and sockets) are assigned labels from the same security lattice $\mathcal{L}$. The edges in G describe the information flows that a policy permits.

We now formally define the the concept of compliance between a graph G and a security lattice $\mathcal{L}$. For $u, v \in V(G)$, we write $u \rightsquigarrow v$ if there is a path between vertices u and v in the graph G. An information-flow graph G is compliant with a security lattice $\mathcal{L}$ if all paths through the combined information-flow graph imply that there is a flow in $\mathcal{L}$ between the types of the elements in the graph.

Definition 3.1 (Policy Compliance)**.** An information-flow graph G is *compliant* with a security lattice $\mathcal{L}$ if for each $u, v \in V(G)$ such that $u \rightsquigarrow v$, then $\mathsf{Type}(u) \sqsubseteq \mathsf{Type}(v)$ in the security lattice $\mathcal{L}$.

With respect to MAC policies, a positive result of the compliance test implies that the information-flow graph for a policy does not permit any operations that violate the information-flow goals as encoded in the lattice $\mathcal{L}$. If G is the information flow graph of a trusted program together with the system policy, then a compliance test verifies that the trusted program only permits information flows allowed by the operating system, as we desire.

3.2 Difficulty of Compliance Testing

The main difficulty in compliance testing is in automatically constructing the program, system, and goal policies shown in Figure 2. Further, we prefer design constructions that will be likely to yield successful compliance.

The two particularly difficult cases are the *program policies* (i.e., upper left in the figure) and the *tamperproof goal policy* (i.e. lower right in the figure). The program policy and tamperproof goal policies require program requirements to be integrated with system requirements, whereas the system policy and system security goals are largely (although not necessarily completely) independent of the program policy. For example, while the system policy must include information flows for the program, the SELinux system includes policy modules for the `logrotate` and other trusted programs that can be combined directly.

First, it is necessary for *program policies* (i.e., upper left in the figure) to manage system objects, but often program policy and system policy are written with disjoint label sets. Thus, some mapping from program labels to system labels is necessary to construct a system-aware program policy before the information flow goals encoded in $\mathcal{L}$ can be evaluated. Let P be an information-flow graph relating the program subjects and objects and and S be information-flow graph relating the system subjects and objects. Let $P \oplus S$ be the policy that arises from combining P and S to form one information-flow graph through some sound combination operator $\oplus$; that is, if there is a runtime flow in the policy S where the program P has been deployed, then there is a flow in the information flow graph $P \oplus S$. Currently, there are no automatic ways to combine such program and system graphs into a system-aware program policy, meaning that $\oplus$ is implemented in a manual fashion. A manual mapping was used in previous work on compliance [13].

Second, the *tamperproof goal policy* (i.e., lower right in the figure) derives from the program's integrity requirements for its objects. Historically, such requirements are not explicitly specified, so it is unclear which program labels imply high integrity and which files should be assigned those high integrity labels. With the use of packages and program policy modules, the program files and labels are identified, but we still lack information about what defines tamperproofing for the program. Also, some program files may be created at installation time, rather than provided in packages, so the integrity of these files needs to be determined as well. We need a way to derive tamperproof goals automatically from packages and policy modules.

4 PIDSI Approach

We propose the *PIDSI approach* (**Program Integrity Dominates System Integrity**), where the trusted program objects (i.e., package files and files labeled using the labels defined by the module policy) are labeled such that their integrity is relative to all system objects. The information flows between the system and the trusted program can then be inferred from this relationship. We have found that almost all trusted program objects are higher integrity than system objects (i.e., system data should not flow to trusted program objects). One exception that we have found is that both trusted and untrusted programs are authorized to write to some log files. However, a trusted program should not depend on the data in a log file. While general cases may eventually be identified automatically as low integrity, at present we may have a small number of cases where the integrity level must be set manually.

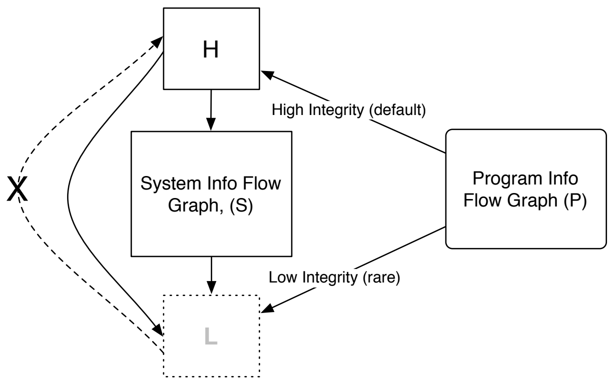


Figure 3: The PIDSI approach relates program labels P to system labels S, such that the program-defined objects are higher integrity than the system data objects (assigned to H), with some small number of low integrity exceptions (assigned to L).

Our approach takes advantage of a distinction between the protection of the trusted program and protection of the data to which it is applied. Trusted program packages contain the files necessary to execute the program, and the integrity of the program's execution requires protection of these files. On the other hand, the program is typically applied to data whose protection requirements are defined by the system.

4.1 PIDSI Definition

By using the PIDSI approach between trusted program and the system, we can deploy that trusted program on different systems, ensuring compliance. Figure 3 demonstrates this approach. First, the program defines its own set of labels, which are designed either as high or low integrity. When the program is deployed, the system labels are placed in between the program's high and low integrity labels. This allows an easy check of whether a program is compliant with the system's policy, regardless of the specific mappings from system inputs and outputs to program inputs and outputs.

In the event that the trusted program allows data at a low integrity label to flow to a high label, then this approach can trick the system into trusting low integrity data. To eliminate this possibility, we automatically verify that no such flows are present in the program policy.

For confidentiality, we found that the data stored by most trusted programs was intended to be low secrecy. The only exception to this rule that we found in the trusted program core of SELinux was `sshd`; this program managed SSH keys at type `sshd_key_t`, which needed to be kept secret[5]. We note that if program data is low secrecy as well as high integrity the same information flows result, system data may not flow to program data, so no change is required to the PIDSI approach. Because of this, we primarily evaluate the PIDSI approach with respect to integrity.

In this context, the compliance problem requires checking that the system's policy, when added to the program, does not allow any new illegal flows. We construct the composed program policy P' from P and S. To composte P and S into $P' = P \oplus S$, first, split P into subgraphs H and L as follows: if $u \in P$ is such that $\mathsf{Integrity}(\mathsf{Type}(u)) = \mathsf{high}$, then $u \in H$, and if $u \in P$ is such that $\mathsf{Integrity}(\mathsf{Type}(u)) = \mathsf{low}$, then $u \in L$. P' contains copies of S, H, L, with edges from each vertex in H to each vertex in S, and edges from each vertex in S to L. The constructed system policy P' corresponds to the deployment of the program policy P on the system S.

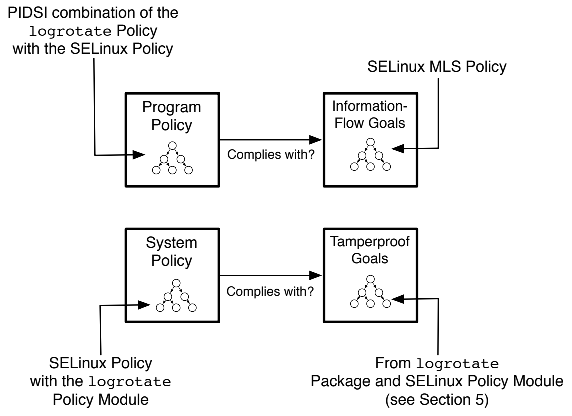
Theorem 4.1. *Assume for all* $v \in V(P)$, $\mathsf{Conf}(\mathsf{Type}(v)) = \mathsf{Conf}(\perp)$. *Given test policy P and target policy S, if for all $u \in H$, $v \in L$, there is no edge $(v, u) \in P$, then the test policy P is compliant with the constructed system policy P'.*

Given the construction, the only illegal flow that can exist in P' is from a vertex $v \in L$, which has a low integrity label, to one of the vertices $u \in H$, which has a high integrity label. The graph S is compliant with P' by definition, and the edges that we add between subgraphs are from H to S and S to L: these do not upgrade integrity.

We argue that the PIDSI approach is consistent with the view of information flows in the trusted programs of classical security models. For example, MLS guards are trusted to downgrade the secrecy of data securely. Since an MLS guard must not lower the integrity of any downgraded data, it is reasonable to assume that the integrity of an MLS guard must exceed the system data that it processes. In the Clark-Wilson integrity model [7], only trusted *transformation procedures* (TPs) are permitted to modify high integrity data. In this model, TPs must be certified to perform such high integrity modifications securely. Thus, they also correspond to our notion of trusted programs. We find that other trusted programs, such as *assured pipelines* [5], also have a similar relationship to the data that they process.

4.2 PIDSI in Practice

In this section, we describe how we use the PIDSI approach to construct the two policy compliance problems defined in Section 3 for SELinux trusted programs. Our proposed mechanism for checking compliance of a trusted program during system deployment was presented in Figure 1: we now give the specifics how this procedure would work during an installation of



PIDSI combination of the logrotate Policy with the SELinux Policy
SELinux MLS Policy
Program Policy
Complies with?
Information-Flow Goals
System Policy
Complies with?
Tamperproof Goals
SELinux Policy with the logrotate Policy Module
From logrotate Package and SELinux Policy Module (see Section 5)

Figure 4: `logrotate` instantiation for the two policy compliance problems: (1) the *program policy* is derived using the PIDSI approach and the SELinux MLS policy forms the system's *information flow goals* and (2) the *system policy* is combined with the `logrotate` SELinux policy module and the *tamperproofing goals* are derived from the `logrotate` Linux package.

`logrotate`. Figure 4 shows how we construct both problems for `logrotate` on an SELinux/MLS system. For testing compliance against the system security goals, we use the PIDSI approach to construct the `logrotate` program policy and use the SELinux/MLS policy for the system security goals. For testing compliance against the tamperproof goals, we use the SELinux/MLS policy that includes the `logrotate` policy module for the system policy and we construct the tamperproof goal policy from the `logrotate` package. We argue why these constructions are satisfactory for deploying trusted programs, using `logrotate` on SELinux/MLS as an example.

For system security goal compliance, we must show that the program policy only permits information flows in the system security goal policy. We use the PIDSI approach to construct the program policy as described above. For the Jif version of `logrotate`, this entails collecting the types (labels) from its SELinux policy module, and composing a Jif policy lattice where these Jif version of these labels are higher integrity (and lower secrecy) than the system labels. Rather than adding each system label to the program policy, we use a single label as a template to represent all of the SELinux/MLS labels [13]. We use the SELinux/MLS policy for the security goal policy. This policy clearly represents the requirements of the system, and `logrotate` adds no additional system requirements. While some trusted programs may embody additional requirements that the system must uphold (e.g., for individual users), this is not the case for `logrotate`. As a result, to verify compliance we must show that there are no information flows in the program policy from system labels to program labels, a problem addressed by previous work [13].

For tamperproof goal compliance, we must show that the system policy only permits information flows that are authorized in the tamperproof goal policy. The system policy includes the `logrotate` policy module, as the combination defines the system information flows that impact the trusted program. The tamperproof policy is generated from the `logrotate` package and its SELinux policy module. The `logrotate` package identifies the labels of files used in the logrotate program. In addition to these labels, any new labels defined by the `logrotate` policy module, excepting process labels which are protected differently as described in Section 2.2, are also added to the tamperproof policy. The idea is that these labels may not be modified by untrusted programs. That is, untrusted process labels may not have any kind of write permission to the `logrotate` labels. Unlike security goal compliance, the practicality of tamperproof compliance is clear. It may be that system policies permit many subjects to modify program objects, thus making it impossible to achieve such compliance. Also, it may be difficult to correctly derive tamperproof goal policies automatically. In Section 5, we show precisely how we construct tamperproof policies and test compliance, and examine whether tamperproof compliance, as we have defined it here, is likely to be satisfied in practice.

5 Verifying Compliance in SELinux

In this section, we evaluate the PIDSI approach against actual trusted programs in the SELinux/MLS system. As we discussed in Section 4.2, we want to determine whether it is possible to automatically determine tamperproof goal policies and whether systems are likely to comply with such policies. First, we define a method for generating tamperproof goal policies automatically and show how compliance is evaluated for the `logrotate` program. Then, we examine whether eight other SELinux trusted programs meet satisfy tamperproof compliance as well. This group of programs was selected because: (1) they are considered MLS-trusted in SELinux and (2) these programs have Linux packages and SELinux policy modules. Our evaluation finds that there are only 3 classes of exceptions that result from our compliance checking for all of these evaluated packages. We identify straightforward resolutions for each of these exceptions. As a result, we find that the PIDSI approach appears promising for trusted programs in practice.

5.1 Tamperproof Compliance

To show how tamperproof compliance can be checked, we develop a method in detail for the `logrotate` program on a Linux 2.6 system with a SELinux/MLS strict reference policy. To implement compliance checking with the tamperproof goals, we construct representations of the system (SELinux/MLS) policy and the program's tamperproof goal policy. Recall from Section 3 that all the information flows in the system policy must be authorized by the tamperproof goal policy for the policy to comply.

5.1.1 Build the Tamperproof Goal Policy

To build the tamperproof goal policy, we build an information-flow graph that relates the program labels to system labels according to the PIDSI approach. Building this graph consists of the following steps:

1. Find the high integrity program labels.
2. Identify the trusted system subjects.
3. Add information flow edges between the program labels, trusted subject labels, and remaining (untrusted) SELinux/MLS labels authorized by the PIDSI approach.

Find the high integrity program labels. This step entails collecting all the labels associated with the program's files, as these will all be high integrity per the PIDSI approach. These labels are a union of the package file labels determined by the file contexts (`.fc` file in the SELinux policy module and the system file context) and the newly-defined labels in the policy module itself. First, the `logrotate` package includes the files indicated in Table 1. This table presents lists a set of files, the label assigned to each, whether such label is a program label (i.e., defined by the program's policy module) or a system label, and the result of the tamperproof compliance check, described below. Second, some program files may be generated after the package is installed. These will be assigned new labels defined in the program policy module. An example of a `logrotate` label that will be assigned to a file that is not included in the package is `logrotate_lock_t`. In Section 6, we discuss other system files that a trusted program may depend upon.

Identify trusted subjects. Trusted subjects are SELinux subjects that are entrusted with write permissions to trusted programs. Based on our experience in analyzing SELinux/MLS, we identify the following seven trusted subjects: `dpkg_script_t`, `dpkg_t`, `portage_t`, `rpm_script_t`, `rpm_t`, `sysadm_t`, `prelink_t`. These labels represent package managers and system administrators; package managers and system administrators must be authorized to modify trusted programs. These subjects are also trusted by programs other than `logrotate`. We would want to control what code is permitted to run as these labels, but that is outside the scope of our current controls.

File	SELinux Type	Policy	Writers	Exceptions
/etc/logrotate.conf	etc_t	system	18	integrity
/etc/logrotate.d	etc_t	system	18	integrity
/usr/sbin/logrotate	logrotate_exec_t	module	8	no
/usr/share/doc/logrotate/CHANGES	usr_t	system	7	no
/usr/share/man/logrotate.gz	man_t	system	8	no
/var/lib/logrotate.status	logrotate_var_lib_t	module	8	no

Table 1: `logrotate` Compliance Test Case and Results: there are two exceptions, but they originate from the same system label `etc_t`.

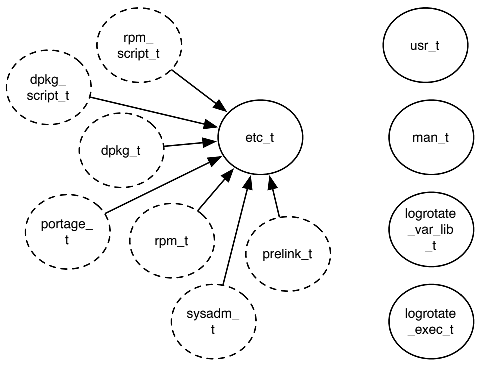


Figure 5: Part of the *tamperproof goal policy's* information-flow graph for `logrotate`. Only trusted labels (dotted line circles) and the program labels themselves are allowed to write to files with the program labels (solid line circles), which represent the high-integrity files according to the PIDSI approach. Not shown: edges from the trusted subjects to each of the program labels to the the right.

Add information flow edges. This step involves adding edges between vertices (labels) in the tamperproof goal information-flow graph based on the PIDSI approach. The PIDSI approach allows program labels to read and write each other, but the only SELinux/MLS labels that may write program labels are the trusted subjects (and read as well). Other SELinux labels are restricted to reading the program labels only. Figure 5 presents an example of a tamperproof goal policy's information-flow graph. Notice that only the system trusted labels (dotted circles) are allowed to write to program labels (solid line circles). The application has high integrity requirements for `etc_t`; the graph therefore includes edges that represent these requirements. The same set of edges are also added for the other program labels (presented to the right in the figure).

5.1.2 Build the System Policy

The system policy is represented as an information-flow graph (see Section 3). Building this graph consists of the following steps:

1. Create an information-flow graph that represents the current SELinux/MLS policy.
2. Add `logrotate` program's information flow vertices and edges based on its SELinux policy module.
3. Remove edges where neither vertex is in the tamperproof goal policy.

Create an information flow graph. We convert the current SELinux/MLS policy into an information-flow graph. Each of the labels in the SELinux/MLS policies is converted to a vertex. Information-flow edges are created by identifying *read-like* and *write-like* permissions [10, 29] for subject labels to objects labels. The following example illustrates the process we follow to create a small part of the graph. Rules 1-3 and 6 are system rules, rules 4-5 are module rules (defined in the `logrotate` policy module).

```
1.  allow init_t init_var_run_t:file
  {create getattr read append write
  setattr unlink};
2.  allow init_t bin_t:file
  {{read getattr lock execute ioctl}
  execute_no_trans};
3.  allow init_t etc_t:file
  {read getattr lock ioctl};
4.  allow logrotate_t etc_t:file
  {read getattr lock ioctl};
5.  allow logrotate_t bin_t:file
  {{read getattr lock execute ioctl}
  execute_no_trans};
6.  allow chfn_t etc_t:file
  {create ioctl read getattr write
  setattr append link unlink rename};
```

Figure 6 shows the result of the parsing of the previous rules. In this example, subjects with type `init_t` are allowed to read from and write to `init_var_run_t` and `logrotate_t` is allowed to read from `etc_t` and `bin_t`.

We note that Figure 6 shows that `chfn_t` has write access to `etc_t` which `logrotate_t` can read. While `logrotate` cannot write any file with the label `etc_t`,

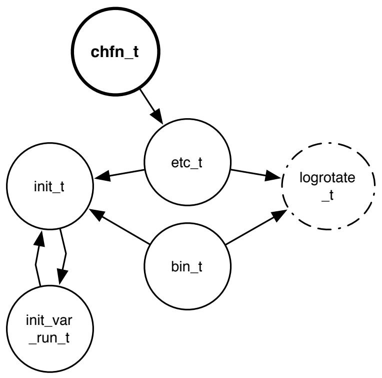


Figure 6: Information-flow graph for the system policy, including the `logrotate` program's policy module. `chfn_t` is not trusted to modify other trusted programs, but it has write access to `logrotate`'s files labeled `etc_t`.

it provides such a file via its package installation, so it depends on the integrity of files of the label. This will be identified as a tamperproof compliance exception below.

We are able to parse the text version of an SELinux policy (file `policy.conf`) with a C program integrated with Flex and Bison. We are also able to analyze the binary version of the SELinux system policy.

Add `logrotate` program's information flows. In a similar fashion to the method above, we extend the information flow graph with the vertices (labels) and edges (read and write flows) from the `logrotate` policy module.

Remove edges where neither vertex is in the tamperproof goal policy. As these flows cannot tamper the `logrotate` program, we remove these edges from the system policy for compliance testing.

5.1.3 Evaluating `logrotate`

This section presents how we automatically test tamperproof compliance. Tamperproof compliance is based checking the system policy for information flow integrity as defined by the tamperproof goal policy.

Integrity Compliance Checking. To detect integrity violations, we identify information flows that violate the Biba integrity requirement [4]: an information flow from a low integrity label (`type` in SELinux) to a high integrity label. `read` and `write` arguments are subject and object.

$$NonBibaFlows_{SELinux}(Policy) = \{(t_1, t_2) : t_1, t_2 \in types(Policy).\ highintegrity(t_1) \wedge lowintegrity(t_2) \wedge (read(t_1, t_2) \vee write(t_2, t_1))\}$$

We use the XSB Prolog engine [32] as the underlying platform. We developed a set of prolog queries based on the *NonBiba Flows* rule to detect the labels that affect compliance (i.e., the high integrity requirement that are not enforced by the system policy).

As mentioned in the previous section, we evaluate tamperproof compliance at installation time. Each time we load the policy graphs generated above into the Prolog engine and we run the integrity Prolog queries to determine if any flows satisfy (negatively) the *NonBiba Flows*, thus violating compliance.

Results. Table 1 presents the results for compliance checking `logrotate` against the generated tamperproof goal policy (see column 4). Only `etc_t` has unauthorized writers. In the SELinux/MLS reference policy, these writers are programs with legitimate reasons to write to files in the `/etc` directory, but none have legitimate reasons to write to `logrotate` files. For example, `chfn`, `groupdadd`, `passwd`, and `useradd` are programs that modify system files that store user information in `/etc`, `kudzu` is an program that detects and configures new and/or changed hardware in a system and requires to update its database stored in `/etc/sysconfig/hwconf`, and `updfstab` is designed to keep `/etc/fstab` consistent with the devices plugged in the system.

The obvious solution would be to refine the labels for files in `/etc` to eliminate these kinds of unnecessary and potentially-risky operations.

5.2 Evaluating other Trusted Programs

Table 2 shows a summary of the results from applying the PIDSI approach to eight SELinux trusted programs for which policy modules and packages are defined. The table shows: (1) trusted package, (2) file labels (SELinux types) used per package, (3) number of writers detected per type (Writers) and (4) exceptions. The integrity requirement assigned by default is high integrity for all types, except for the ones marked with **; because of the semantics associated to `/var`, various applications write to this directory, we assign low integrity requirement to `var_log_t` and `var_run_t`.

The common system types (`bin_t, etc_t, lib_t, man_t, sbin_t` and `usr_t`) are marked with † in the last two columns. The results for these types are displayed in Table 3. The results show only two exceptions, none in Table 2 and two in Table 3.

These reasons behind and resolutions for these exceptions are shown in Table 4. One good resolution would be a refinement of the policies: programs should have particular labels for their files, even if they are installed in system directories, instead of using general system labels. The use of a general system label gives all system

Package	SELinux Label	Writers	Exception
cups	initrc_exec_t	8	no
	textrel_shlib_t	9	no
	lpr_exec_t	8	no
	dbusd_etc_t	7	no
	system types	†	†
	var_log_t**	14	no
	var_run_t**	10	no
	var_spool_t	10	no
dmidecode	dmidecode_exec_t	8	no
	system types	†	†
hald	locale_t	7	no
	initrc_exec_t	8	no
	hald_exec_t	8	no
	dbusd_etc_t	7	no
	system types	†	†
iptables	iptables_exec_t	8	no
	initrc_exec_t	8	no
	system types	†	†
kudzu	locale_t	7	no
	initrc_exec_t	8	no
	system types	†	†
Network Manager	initrc_exec_t	8	no
	NetworkManager_var_run_t	8	no
	NetworkManager_exec_t	8	no
	dbusd_etc_t	7	no
	system types	†	†
rpm	rpm_exec_t	8	no
	rpm_var_lib_t	7	no
	system types	†	†
	var_spool_t	10	no
sshd	sshd_exec_t	8	no
	sshd_var_run_t	8	no
	ssh_keygen_exec_t	8	no
	ssh_keysign_exec_t	8	no

Table 2: Results of applying the PIDSI approach to SELinux Trusted Packages. Columns with a '†' are displayed in table 3

programs access to these files (case APP LABELS in Table 4). However, this option is not always possible, as sometimes a program actually requires access to system files. In such cases, the programs have to be trusted (case ADD in Table 4). For example, some trusted programs read information from the `/etc/passwd` file, so those subjects permitted to modify that file must be trusted. Only a small number of such programs must be trusted.

6 Discussion

Trusted programs may use system files, such as system libraries or the password file, in addition to the files provided in their packages. Because some of our trusted program packages installed their own libraries under the system label `lib_t` our analysis included system libraries. Therefore, application integrity not only depends on the integrity of the files in the installation package but also on some other files. In general, the files that the program execution depends on should be comprehensively identified. These should be well-known per system.

An issue is whether a trusted program may create a file whose integrity it depends upon that has a system label. For example, a trusted program generates the password file, but this used by the system, so it has a system label. We did not see a case where this happened for our trusted programs, but we believe that this is possible in practice. We believe that more information about the integrity of the contents generated by the program will need to be used in compliance testing. For example, if the program generates data it marks as high integrity, then we could leverage this in addition to package files and program policy labels to generate tamperproof goal policies.

An issue with our approach is the handling of low integrity program objects. Since low integrity program objects are the lowest integrity objects in the system, any program can write to these objects. We find that we want low integrity program objects to be relative to the trusted programs; lower than all trusted programs, but still higher than system data. Further investigation is required.

The approach in this paper applies only to trusted programs. We make no assumptions about the relationship between untrusted program and the system data. In fact, we are certain that there is system data that should not be accessed by most, if not all, untrusted programs. Note that there is no advantage to verifying the compliance of untrusted program, because the system does not depend on untrusted programs to enforce its security goals. Such programs have no special authority.

SELinux Label	Writers	Exceptions
bin_t	9	no
etc_t	18	integrity
lib_t	8	no
man_t	8	integrity
sbin_t	8	no
usr_t	7	no

Table 3: System labels referenced by the packages presented in Table 2. Only `etc_t` and `man_t` have conflicts; the number of conflicting types per case can not be high (Writers column is an upper limit since it includes trusted writers), so we can precisely examine each exception and suggest resolutions (shown in Table 4).

7 Related Work

Policy Analysis. Policies generally contain a considerable number of rules that express how elements in a given

SELinux Label	Conflicting Labels	Type of Exception	Resolution Method	Comment
etc_t	groupadd_t, passwd_t, useradd_t, chfn_t	Integrity	ADD	The conflicting labels require access to the the same file `/etc/passwd`
etc_t	updfstab_t, ricci_modstorage_t, firstboot_t	Integrity	ADD	The first two labels have legitimate reasons to modify `/etc/fstab`. The last type modifies multiple files in `/etc`
etc_t	postgresql_t,kudzu_t	Integrity	APP LABELS	The conflicting types need access to application files labeled with system labels
man_t	system_crond_t	Integrity	REMOVE	crond does not need to write manual pages
ADD: Add conflicting types to the set of trusted readers (confidentiality) or writers (integrity). APP LABELS: The associated application requires access to a file that is application specific but was labeled using system labels. Adding application specific labels to handle those files solves the conflict. REMOVE: The permission requested is not required				

Table 4: **Compliance Exceptions and Resolutions**. This table details the exceptions to tamperproof compliance presented in Table 3. It shows the list of conflicting, untrusted subjects and the resolution method, per case.

environment must be controlled. Because of the size of a policy and the relationships that emerge from having a large number of rules, it is difficult to manually evaluate whether a policy satisfies a given property or not. As a consequence, tools to automatically analyze policy are necessary. APOL [35], PAL [29], SLAT [10], Gokyo [16] and PALMS [15] are some of the tools developed to analyze SELinux policies; however, each of these tools focuses on the analysis of single security policies. Of these, only PALMS offers mechanisms to compare policies; in particular it addresses compliance evaluation, but our approach to compliance is broader and allows the compliance problem to be automated.

Policy Modeling. We need a formal model to reason about the features of a given policy. Such a model should be largely independent of particular representation of the targeted policies and should enable comparisons among different policies. Multiple models have been proposed and each one of them defines a set of components that need to be considered when translating a policy to an intermediate representation. Cholvy and Cuppens [6] focus on permissions, obligations, prohibitions and provide a mechanism to check regulation consistency. Bertino et al. [3] focus on subjects, objects and privileges, as well as the organization of these components and the set of authorization rules that define the relationships among components and the set of derived rules that may be generated because of a hierarchical organization. Kock et al. [18] represent policies as graphs with nodes that represent components(processes, users, objects) and edges that represent rules and a set of constraints that globally applied to the system. In any case, policy modeling becomes a building block in the process of evaluating compliance. Different policies must be translated to an intermediate representation (a common model) so they can be compared and their properties evaluated.

Policy Reconciliation. Policy compliance problems may resemble policy reconciliation problems. Given two policies A and B that define a set of requirements, a reconciliation algorithm looks for a specific policy instance C that satisfies the stated requirements. Policy compliance in a general sense, i.e. 'Given a policy A and a policy B, is B compliant with A?' means 'is any part of A in conflict with B?'. Previous work [21] shows that reconciliation of three or more policies is intractable. Compliance is also a intractable problem since this would require to checking all possible paths in B against all possible paths in A. Although both of these problems are similar in that they both test policy properties and are non-tractable in general cases (no restrictions), they differ in their inputs and expected outputs. While in the case of reconciliation, an instance that satisfies the requirements has to be calculated, in the case of compliance, policy instances are given and one is evaluated against the other one.

Policy Compliance. The security-by-contract paradigm resembles our policy compliance model. It is one of the mechanisms proposed to support installation and execution of potentially malicious code from a third party in a local platform. Third party applications are expected to come with a security contract that specifies application behavior regarding security issues. The first step in the verification process is checking whether the behaviors allowed by the contract are also allowed by the local policy [8]. In the most recent project involving contract matching, contract and policy are security automatons and the problem of contract matching becomes a problem of testing language inclusion for automatons. While there is no known polynomial technique to test language inclusion for non-deterministic automatons, determining language inclusion for deterministic automatons is known to be polynomial [9]. One main advantage of our representation is that we are verifying policies that are actually implemented by the enforcing mechanism, not high level statements that may not be actually implemented because of the semantic gap between specification and implementation. In addition, the enforcing mechanism is part of the architecture.

8 Conclusion

This work is driven by the idea of unifying application and system security policies. Since applications and systems policies are independently developed, they use different language syntax and semantics. As a consequence, it is difficult to prove or disprove that programs enforce system security goals. The emergence of mandatory access control systems and security typed languages makes it possible to automatically evaluate whether applications and systems enforce common security goals. We reshape this problem as a verification problem: we want to evaluate if applications are *compliant* with system policies.

We found that compliance verification involves two tasks: we must ensure that the system protects application from being tampered with, as well as verify that the application enforces system security goals. In order to automate the mapping between the program policy and the system policy, we proposed the PIDSI (**Program Integrity Dominates System Data Integrity**) approach. The PIDSI approach relies on the observation that in general program objects are higher integrity than system objects. We tested the trusted program core of the SELinux system to see if its policy was compatible with the PIDSI approach. We found that our approach accurately represents the SELinux security design with a few minor exceptions, and requires little or no feedback from administrators in order to work.

Notes

[1]The program verification (e.g., STL compilation) enforces the *complete mediation* guarantee.

[2]At present, module policies are not included in Linux packages, but RedHat, in particular, is interested in including SELinux module policies in its `rpm` packages in the future [36].

[3]SELinux uses the term *type* for its labels, as it uses an extended Type Enforcement policy [5].

[4]As described above, this must be done manually now, via `semodule`, but the intent is that when you load a package containing a module policy, someone will install the module policy.

[5]In this case, violating the confidentiality of SSH keys enables a large class of integrity attacks. This phenomenon has been discussed more generally by Sean Smith [31].

References

[1] ANDERSON, J. P. Computer security technology planning study. Tech. Rep. ESD-TR-73-51, The Mitre Corporation, Air Force Electronic Systems Division, Hanscom AFB, Badford, MA, 1972.

[2] BELL, D. E., AND LAPADULA, L. J. Secure computer systems: Unified exposition and multics interpretation. Tech. rep., MITRE MTR-2997, March 1976.

[3] BERTINO, E., CATANIA, B., FERRARI, E., AND PERLASCA, P. A logical framework for reasoning about access control models. In *Proceedings of SACMAT* (2001).

[4] BIBA, K. J. Integrity considerations for secure computer systems. Tech. Rep. MTR-3153, MITRE, April 1977.

[5] BOEBERT, W. E., AND KAIN, R. Y. A practical alternative to heirarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference* (1985).

[6] CHOLVY, L., AND CUPPENS, F. Analyzing Consistency of Security Policies. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (Oakland, CA, USA, May 1997), pp. 103–112.

[7] CLARK, D. D., AND WILSON, D. A comparison of military and commercial security policies. In *1987 IEEE Symposium on Security and Privacy* (May 1987).

[8] DESMET, L., JOOSEN, W., MASSACCI, F., NALIUKA, K., PHILIPPAERTS, P., PIESSENS, F., AND VANOVERBERGHE, D. A flexible security architecture to support third-party applications on mobile devices. In *Proceedings of the ACM Computer Security Architecture Workshop* (2007).

[9] DRAGONI, N., MASSACCI, F., NALIUKA, K., SEBASTIANI, R., SIAHAAN, I., QUILLIAN, T., MATTEUCCI, I., AND SHAEFER, C. Methodologies and tools for contract matching. Security of Software and Services for Mobile Systems.

[10] GUTTMAN, J. D., HERZOG, A. L., RAMSDELL, J. D., AND SKORUPKA, C. W. Verifying information flow goals in Security-Enhanced Linux. *J. Comput. Secur. 13*, 1 (2005), 115–134.

[11] HANSON, C. SELinux and MLS: Putting the pieces together. In *Proceedings of the 2nd Annual SELinux Symposium* (2006).

[12] HICKS, B., KING, D., MCDANIEL, P., AND HICKS, M. Trusted declassification: High-level policy for a security-typed language. In *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '06)* (Ottawa, Canada, June 10 2006), ACM Press.

[13] HICKS, B., RUEDA, S., JAEGER, T., AND MCDANIEL, P. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference* (2007).

[14] HICKS, B., RUEDA, S., JAEGER, T., AND MCDANIEL, P. Integrating SELinux with security-typed languages. In *Proceedings of the 3rd SELinux Symposium* (Baltimore, MD, USA, March 2007).

[15] HICKS, B., RUEDA, S., ST. CLAIR, L., JAEGER, T., AND MCDANIEL, P. A logical specification and analysis for SELinux MLS policy. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)* (Antipolis, France, June 2007).

[16] JAEGER, T., EDWARDS, A., AND ZHANG, X. Managing access control policies using access control spaces. In *SACMAT '02: Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies* (2002), ACM Press, pp. 3–12.

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

[18] KOCK, M., MACINI, L., AND PARISI-PRESICCE, F. On the specification and evolution of access control policies. In *Proceedings of SACMAT* (2001).

[19] LI, N., MAO, Z., AND CHEN, H. Usable mandatory integrity protection for operating systems. In *IEEE Symposium on Security and Privacy* (2007).

[20] MAYER, F., MACMILLAN, K., AND CAPLAN, D. *SELinux by Example*. Prentice Hall, 2007.

[21] MCDANIEL, P., AND PRAKASH, A. Methods and limitations of security policy reconciliation. *ACM Transactions on Information and System Security V*, N (May 2006), 1–32.

[22] McGraw, G., and Felten, E. *Java Security*. Wiley Computer, 1997.

[23] Myers, A. C. JFlow: Practical mostly-static information flow control. In *POPL '99*, pp. 228–241.

[24] Myers, A. C., and Liskov, B. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles* (October 1997).

[25] Myers, A. C., and Liskov, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology 9*, 4 (2000), 410–442.

[26] Myers, A. C., Nystrom, N., Zheng, L., and Zdancewic, S. Jif: Java + information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[27] Security-enhanced Linux. Available at `http://www.nsa.gov/selinux`.

[28] Pottier, F., and Simonet, V. Information Flow Inference for ML. In *Proceedings ACM Symposium on Principles of Programming Languages* (Jan. 2002), pp. 319–330.

[29] Sarna-Starosta, B., and Stoller, S. Policy analysis for Security-Enhanced Linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)* (April 2004), pp. 1–12. Available at http://www.cs.sunysb.edu/˜stoller/WITS2004.html.

[30] Shankar, U., Jaeger, T., and Sailer, R. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium (NDSS'06)* (San Diego, CA, USA, Feb. 2006).

[31] Smith, S. W. Outbound authentication for programmable secure coprocessors. In *European Symposium on Research in Computer Security (ESORICS)* (2002), pp. 72–89.

[32] Stony Brook University. Computer Science Department. XSB: Logic programming and deductive database system for Unix and Windows. Available at `http://xsb.sourceforge.net`.

[33] Swamy, N., Corcoran, B., and Hicks, M. Fable: A language for enforcing user-defined security policies. In *In Proceedings of the IEEE Symposium on Security and Privacy (Oakland), May 2008. To appear.*

[34] Tresys Technology. SELinux Policy Server. Available at http://www.tresys.com/selinux/selinux_policy_server.

[35] Tresys Technology. SETools - policy analysis tools for SELinux. available at http://oss.tresys.com/projects/setools.

[36] Walsh, D. SELinux Mailing List. `http://www.engardelinux.org/modules/index/list_archives.cgi?list=selinu%x&page=0609.html&month=2007-12`, December 2007.

[37] The X Foundation: `http://www.x.org`.