Toward Automated Information-Flow Integrity Verification for Security-Critical Applications

Umesh Shankar
ushankar@cs.berkeley.edu
University of California at Berkeley

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

Reiner Sailer
sailer@us.ibm.com
IBM T. J. Watson Research Center

Abstract

We provide a largely automated system for verifying Clark-Wilson interprocess information-flow integrity. Information-flow integrity properties are essential to isolate trusted processes from untrusted ones, but system misconfiguration can easily create insecure dependences. For example, an untrusted user process may be able to write to `sshd_config` *via a cron script. A useful notion of integrity is the Clark-Wilson integrity model [7], which allows trusted processes to accept necessary untrusted inputs (e.g., network data or print jobs) via filtering interfaces that sanitize the data. However, Clark-Wilson has the requirement that programs undergo formal semantic verification; in practice, this kind of burden has meant that no information-flow integrity property is verified on most widely-used systems. We define a weaker version of Clark-Wilson integrity, called CW-Lite, which has the same interprocess information-flow guarantees, but which requires less filtering, only small changes to existing applications, and which we can check using automated tools. We modify the SELinux user library and kernel module in order to support CW-Lite integrity verification and develop new software tools to aid developers in finding and enabling filtering interfaces. Using our toolset, we found and fixed several integrity-violating configuration errors in the default SELinux policies for OpenSSH and vsftpd.*

1 Introduction

1.1 Motivation and Goals

While operating systems provide isolation through separate memory spaces, processes still interact via files, pipes, network connections, shared memory, and other mechanisms. We say that there is an *information flow* between a process A and a process B if A can write to some resource (e.g., a file or pipe) on which B depends. (We do not consider side-channel attacks in this paper.) The *information-flow integrity verification problem* is to prove that a security-critical, or *high integrity*, process does not depend on information flows from untrusted, or *low integrity*, processes.

Let us consider an example. If an untrusted user can write to the trusted OpenSSH configuration file, `sshd_config`, that is a violation of information-flow integrity and a clear security breach. Transitive flows must also be checked: if an untrusted user can run a `cron` job that writes `sshd_config`, there is obviously still an integrity violation. Merely setting file permissions does not prevent attacks that operate via, say, pipes or shared memory: we must consider all kinds of inputs. In general, if a trusted program depends on untrusted inputs, an attacker may be able to gain escalated privilege or compromise the system. To maintain information-flow integrity, a system must be properly *configured*, i.e., its set of permissions must be such that illegal flows from untrusted processes to trusted ones are not possible. Such is the approach taken by the Biba [2] model, where the trusted process is said to depend on a resource merely by reading it. That is, no untrusted inputs were allowed to trusted processes.

This picture is complicated by the fact that many trusted processes must accept some untrusted input to function. We say that each `open()` call (or equivalent, such as `connect()` or `accept`) in the program constitutes an *input interface*, or simply an *interface*. Network daemons must accept some input, such as HTTP requests or session logins, from the network. Input to network interfaces may be controlled by an attacker. The programs running on system must perform sanitization or *filtering* of inputs that come from untrusted sources. By using *filtering interfaces*, the program can read from an untrusted resource, while controlling the extent to which it depends on that resource.

In short, information-flow integrity requires a combination of two elements: (1) proper configuration, which ensures that inputs that a program trusts (like config files) cannot be written by untrusted users, and (2) filtering code, which ensures that inputs that a program does not trust (like network input) are checked for well-formedness and application-specific restrictions. Without the ability to communicate with trusted processes except by very narrow interfaces, untrusted users' attack options, and therefore potential exploits, are limited.

The Clark-Wilson integrity model [7] covers both aspects and is a good match for current trusted processes, requiring that all of processes' inputs be filtered or sanitized. However, Clark-Wilson is, like Biba, relatively heavyweight, requiring formal verification for programs. These and other integrity models were developed at a time when deep, complete program analysis for security was thought to be coming in the near future. That vision has not been realized and, as a result, most systems in widespread use operate without any kind of information-flow integrity verification: this long-identified problem is not solved in practice.

Our goal in this paper is to change that; to do so, we define a lighter-weight version of Clark-Wilson integrity, which we term *CW-Lite*, that retains the same interprocess dependency semantics but omits the requirement that programs undergo full formal verification. We then show how a combination of new and existing tools allows practical verification of CW-Lite. These tools address both aspects of integrity verification: they help administrators to find and fix configuration errors, and application developers to find and annotate interfaces that require input sanitization. Verifying CW-Lite is largely automated: administrators must only make manual decisions when violations are found, and developers must only annotate untrusted input interfaces, which are identified with our tools, with a simple macro. Naturally, developers must implement filtering code on these interfaces in any case; the annotation serves to allow static verification of CW-Lite.

Rather than have a tool that simply says that a system has violations, we have tried where possible to make resolving the problems easier as well. We demonstrate the effectiveness of our tools—and thus, the feasibility of achieving Clark-Wilson-style information-flow guarantees—by applying them to privilege-separated OpenSSH, which interacts with many system objects, and has the challenge of containing trusted and untrusted components within one application. We also analyze vsftpd to illustrate the general applicability of our approach. We found several security policy configuration errors that permitted unnecessary, possibly insecure flows. We also determined that certain other programs, such as `rlogind` and `xdm`, caused insecure flows, and should not be run on systems that desire information-flow integrity guarantees. Indeed, one of the benefits of checking information flows on a system is that it makes the formerly implicit TCB of the system explicit, and highlights programs whose presence on the system can cause insecure flows. Although not every insecure information flow leads to an exploitable hole, by eliminating all such flows, we eliminate all related exploits as well.

1.2 Contributions

In this work, we make the following contributions:

- We develop an information-flow integrity property, CW-Lite, which captures the interprocess dependency semantics of Clark-Wilson integrity, but is verifiable on real systems using tools and only a modest amount of manual effort;
- We develop a suite of tools as well as modifications to SELinux to support CW-Lite enforcement;
- We apply our approach to OpenSSH and vsftpd, and find several integrity-violating permissions in their default SELinux policies;
- In short, we have demonstrated practical verification of Clark-Wilson interprocess information-flow integrity.

1.3 Roadmap

Section 2 contains a high-level overview of the CW-Lite model and its verification process. In Section 3, we define CW-Lite formally, starting with the Clark-Wilson model and weakening certain requirements. Section 4 describes our system modifications and the algorithm used to verify the CW-Lite of a trusted application. In Section 5 we apply our approach to OpenSSH and vsftpd on Linux with SELinux, describing how we used our tools to (1) identify filtering interfaces necessary to handle low integrity inputs; and (2) resolve potentially harmful information flows that would not be filtered. We discuss related work in Section 6 and summarize our findings and future work in Section 7.

2 Overview of CW-Lite and its Verification

There are two motivating observations behind CW-Lite. The first is that because the Clark-Wilson model contains a formal verification requirement in addition to an interprocess data flow model, it has proven too heavyweight for widespread use. However, the two goals are separable, and we may profitably try to solve the latter goal independently. The second observation is that Clark-Wilson requires filtering of all interfaces, but most trusted programs only need to open untrusted interfaces at a small number of locations.

The first observation led us to focus our work here on a concrete solution to the first Clark-Wilson goal: securing interprocess information flows in an application-independent way. Accordingly, CW-Lite duplicates the interprocess information-flow semantics of Clark-Wilson. Full formal verification of the programs themselves is a separate and difficult problem; also, verifying semantic correctness is an application-specific task. Obviously this kind of verification is very useful and can prevent other kinds of integrity compromises (e.g., those resulting from buffer overflows), but we believe that separating the two problems will allow simpler, more flexible solutions to each.

Verifying CW-Lite means ensuring that no unfiltered information flows exist from untrusted processes to trusted ones. To do this, we must first identify all possible interprocess information flows. We do so by using a mandatory access-control (MAC) system that interposes access checks on all interprocess flows; with a fine-grained MAC, we can make meaningful statements about which flows are and are not possible. In this paper, we use the SELinux [23] module for Linux, a fine-grained MAC system for Linux that implements Role-Based Access Control with Type Enforcement. SELinux is now a standard part of Fedora Core Linux and is being integrated into many other Linux distributions.

Our second observation led us to extend the MAC system, so developers would not have to filter on trusted inputs. In order to enforce least privilege for both trusted (non-filtering) interfaces and untrusted (filtering) interfaces in one process, the MAC must distinguish between the two kinds of interfaces, allowing only trusted flows to trusted interfaces but allowing additional, untrusted, inputs to the others. By modifying SELinux to associate two security contexts, or *subject types*, with each process instead of one, we can allow only inputs from trusted processes by default, while enabling a special context, a *filtering subject type*, for filtering interfaces that allows necessary kinds of untrusted inputs as well. This separation reduces the burden on the developer relative to Clark-Wilson by requiring filtering only of untrusted inputs, which they must do in any case. The only requirements are that developer annotate filtering interfaces with a simple macro and put relevant permissions in the filtering subject type. To facilitate this process, we also developed a tool that enables developers to identify which inputs should be annotated filtering interfaces, based on the extra permissions they need. These small changes to the development process and SELinux are also what enable automatic verification by administrators on end systems.

Now that we have isolated the trusted interfaces into a separate subject type, we need a way for administrators to detect illegal flows to it from untrusted sources, i.e., verify the CW-Lite property on their systems. In previous work, we developed the Gokyo tool [14, 15], which can determine information flows from an SELinux policy by looking at read-type and write-type permissions, then flag illegal ones when supplied with the system's TCB (assuming other kinds of processes are untrusted). We leverage Gokyo here by having it ignore flows to the filtering subject type for the target application, reporting information-flow violations only for the base subject type.

3 CW-Lite

In this section, we state the CW-Lite model more formally. As we noted previously, CW-Lite is a weakened version of the Clark-Wilson [7] integrity model, but the focus is on controlling interprocess information flows, rather than formal verification of the programs themselves (which is a separate, important problem). In particular, we do not discuss the application-specific task of verifying the semantic correctness of filtering interfaces in this paper. That is, we seek to provide assurance that filtering code has not been omitted, but not assurance of that code's semantic correctness.

3.1 Basic Information-Flow Integrity

In basic information-flow integrity models, dependence on low integrity data is defined in terms of information flows. Such models require that no low integrity information flows may be input to a high integrity subject.

We start with a definition of information flow based on two standard operators, *modify* and *observe* where: (1) $mod(s, o)$ is the modify operator where a *subject* (e.g., a process or user) with subject label s writes to an *object* (e.g., a file or socket) with object label o and (2) $obs(s, o)$ is the observe operator where a subject of subject label s reads from an entity of object label o. We use S to refer to the set of all subjects.

Definition 1 (Basic information flow) $flow(s1, s2)$ *specifies that information flows from subject* $s1$ *to subject* $s2$.

$$flow(s1, s2) := \exists o : mod(s1, o) \wedge obs(s2, o)$$

Next, the operator $int(x)$ defines the integrity level of x where it may be either a subject or an object. In information flow integrity models, integrity levels are related by a lattice [8] where $int(x) > int(y)$ means that y may depend on x, but not vice versa. For our purposes, this means that trusted processes may not depend on untrusted ones.

Definition 2 (Biba integrity) *Biba integrity [2] is preserved for a subject* s *if (1) all high integrity objects meet integrity requirements initially and (2) all information flows to* s *be from subjects of equal or higher integrity:*

$$\forall s_i \in S, flow(s_i, s) \Rightarrow (int(s_i) \geq int(s)).$$

Some information-flow based integrity models, such as LOMAC [12], operate differently but have the same information-flow integrity semantics as Biba.

A note on transitivity. We note that while information flow is transitive in general, only intransitive information flows need to be examined to detect a Biba integrity violation. Suppose that A and B are untrusted and X and Y are trusted. If we have a transitive information flow from $A \rightarrow B \rightarrow X \rightarrow Y$, only the flow from B to X is needed to trigger a Biba integrity violation, i.e., there is always some flow that crosses boundary between untrusted and trusted. It does not impact Biba integrity further that information can flow from A to B. While we find Biba too restrictive, we want to preserve the need only to check flows independently.

3.2 Clark-Wilson Integrity

The Clark-Wilson integrity model [7] provides a different view of dependence. Security-critical processes may accept low integrity information flows (*unconstrained data items or UDIs*), but the program must either discard or upgrade all the low integrity data from all input interfaces. The key to eliminating dependence on low integrity information flows is the presence of *filtering interfaces* that implement the discarding or upgrading of low integrity data. The Clark-Wilson integrity model does not distinguish among program interfaces, but treats the entire security-critical program as a highly assured black box. As a result, all interfaces must be filtering interfaces.

In the original Clark-Wilson model, trusted processes are known as *transformation procedures (TPs)*, typically operate on CDIs (trusted inputs), but may also accept UDIs if it is assured to filter all its inputs. Thus, our notion of trusted applications maps closely to Clark-Wilson's transformation procedures. Clark-Wilson also defined special trusted processes, called *integrity verification procedures (IVPs)*, that check the integrity of CDIs by performing appropriate integrity checks on each data item. These are used to establish systemwide integrity at the start of operation; we do not specifically consider such programs in our model.

We now define information flow in terms of a connection between subject labels and their program interfaces. For this we need a more precise obs operator: $obs(s, I, o)$ means that the subject s reads an object of type o on interface I. An interface for a subject is a distinct input information channel, and is created by, e.g., a particular `open()` call in a program.

Definition 3 (Interface information flow) $flow(s_i, s, I)$ *specifies that information flows from subject s_i to subject s through an interface I in a program running as subject s.*

$$flow(s_i, s, I) := mod(s_i, o) \wedge obs(s, I, o)$$

We also define the predicate $filter(s, I)$ to mean that a subject s filters or sanitizes input on an interface I.

We are now ready to state the Clark-Wilson property.

Definition 4 (Clark-Wilson integrity) *Clark-Wilson integrity is preserved for a subject s if (1) all high integrity objects meet integrity requirements initially; (2) the behavior of the programs of subject S are assured to be correct; and (3) all interfaces filter (i.e., upgrade or discard) low integrity information flows:*

$$\forall s_i \in S, flow(s_i, s, I) \Rightarrow filter(s, I).$$

While the Clark-Wilson model does not require separate multi-level secure processes for upgrading, as does Biba, it requires a significant assurance effort. An important point to note is that since a Clark-Wilson application programmer does not know the system's information flows in advance, all interfaces must be assured to be filtering interfaces. In practice, often only a small number of interfaces actually need to capable of filtering in the context of a real system. This set can be derived from analyzing the system's security policy; that is, by using system knowledge in application development (since the developer can ship a security policy with the application), we can reduce the filtering burden on the developer. We use this observation in developing CW-Lite.

3.3 CW-Lite

Definition 5 (CW-Lite) *CW-Lite is preserved for a subject s if: (1) all high integrity objects meet integrity requirements initially; (2) all trusted code is identifiable as high integrity (e.g., from its hash value as for NGSCB [11]); and (3) all information flows are from subjects of equal or higher integrity unless they are filtered:*

$$flow(s_i, s, I) \wedge \neg filter(s, I) \rightarrow (int(s_i) \geq int(s))$$

That is, CW-Lite requires that the application's information flows either adhere to classical integrity or that untrusted (low-integrity) inputs are handled by a filtering interface. Note that this provides equivalent integrity to Clark-Wilson, since the only flows not being filtered come from trusted sources. Recall that CW-Lite also does not provide for formal verification of filtering interfaces; it simply requires the developer to mark them as such so they can be handled correctly by the MAC system.

4 Developing CW-Lite-Compliant Systems

In this section, we tackle the CW-Lite tasks implied by the previous section. Recall that the CW-Lite property is one that is verified for a particular target application running on a particular system. Application developers must enable verification with small changes to their programs and security policies, while the administrators perform the actual verification on their systems. (See Figures 1 and 2 for

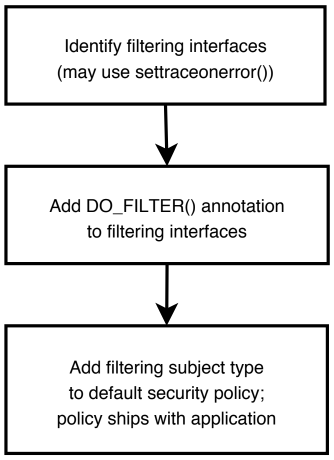
Identify filtering interfaces
(may use settraceonerror())
Add DO_FILTER() annotation
to filtering interfaces
Add filtering subject type
to default security policy;
policy ships with application

Figure 1. Application developer tasks required to enable CW-Lite verification. Filtering interfaces are those that accept inputs from untrusted sources, and must sanitize, or *filter* the input. An interface is marked by a distinct call to `open()`, `accept()`, or other call that enables data input. The `DO_FILTER()` annotation on an interface tells the access-control system to grant additional permissions allowed by the filtering subject type to that interface. The default subject type, used on all other input interfaces, only allows inputs from the the system TCB.

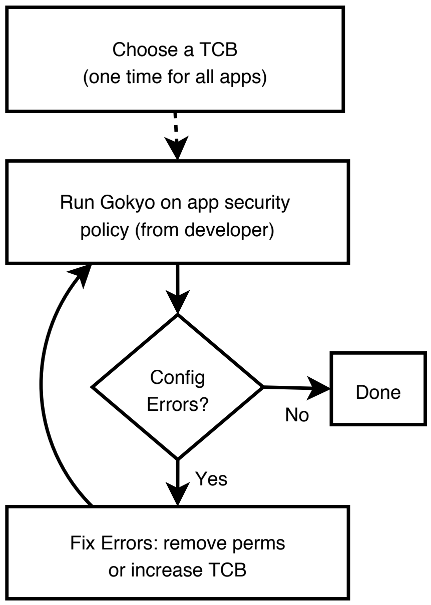

Figure 2. System Administrator tasks required to verify CW-Lite. The system administrator decides on a system TCB initially. Then, when she wants to verify CW-Lite for a particular trusted application, she runs Gokyo on its security policy. If no errors are reported, CW-Lite integrity is verified. If it reports an illegal flows, the offending permissions must be removed, or the TCB expanded to include the source of the illegal flows.

flowcharts of these tasks.) For our discussion, we use the term *TCB* (trusted computing base) to indicate the set of subjects that must be trusted on the system in order to trust the set of target applications (e.g., `sshd`, Apache, bind).

An application developer must:

1. Assuming some TCB and application configuration, identify untrusted inputs to the program and implement filtering interfaces for each. This may be done using the process in Section 4.4.

2. Annotate those interfaces with the `DO_FILTER()` annotation. These annotations are used by the access control system as described in Section 4.2.

3. (Possibly in conjunction with a distribution maintainer:) Construct a default security policy for the application that has two subject types: the default only allows inputs from the TCB, and the other, for filtering interfaces, allows required types of untrusted inputs as well.

Since application developers should be sanitizing their untrusted inputs anyway, this represents only a small amount of additional work to enable system-level integrity verification.

A system administrator must:

- One time only, choose a system TCB. (A TCB may be chosen per-application for a multilevel trust model, but this is not necessary or common. In this scenario, each target app would be associated with only the set of subjects on which it depended, independent of other applications.)

- Run the security policy analysis tool for the target application as described in Section 4.3.

- If no integrity-violating permissions are detected, then skip the next step.

- Classify each integrity-violating permission found by the tool to decide how to remove the illegal flow. See Section 4.5 for details on how to do this.

Note that verifying the CW-Lite property is done automatically using Gokyo; it is only resolution of problems that requires manual intervention. In addition, our approach allows each sysadmin to decide which applications trust on her system. She can evaluate the risk of running a particular application in terms of what must be trusted in order to run it.

In the remainder of this section, we will first describe the SELinux access control system, then show how we modified SELinux to support filtering interfaces and, therefore, CW-Lite verification. We continue by addressing the developer and sysadmin tasks above, including performing policy analysis and finding filtering interfaces.

4.1 SELinux

The SELinux module [23] is a Linux Security Module (LSM) [26] that provides fine-grained, comprehensive MAC enforcement. It ships standard with Fedora Linux, among others, and it is quickly becoming standard to include attendant SELinux policies with applications. SELinux implements an extended form of Type Enforcement (TE) [4] with domain transitions that enables expression of policies covering over 30 different kinds of objects with about 10 operations each. SELinux is comprehensive because it aims to control all programs' accesses to all security-relevant system objects. In this paper, we do not examine verifying that the SELinux/LSM implementation is a correct reference monitor. Previous work verified the LSM reference monitor interface [27], but verifying the correctness of the SELinux implementation properties remains.

Key notions in SELinux are those of *subject types* and *object types*. A process' security context is determined by its subject type, much as the security context of an ordinary UNIX process is determined by its effective UID. Likewise, non-process objects like files are associated with an object type. Permissions are attached to a subject type in policy files; if an Apache process has the subject type `apache_t`, and its configuration file has object type `apache_config_t`, we might say something like

```
allow apache_t apache_config_t:file
   {stat read}
```

to allow Apache to call `stat()` on or read from its configuration file. SELinux does not include a "deny" operation; all permissions are denied by default.

Although there are several access control concepts in the SELinux policy model besides `allow` permissions by subjects on objects, only one other is relevant to information flow. The `relabel` operations[1] enable a subject to change the object label of an object. This enables information flow from the old object label to the new one.

[1]A subject needs the *relabelfrom* and *relabelto* permissions to implement a relabel.

While it allows us great control and flexibility, such fine-grained, comprehensive control results in very large and complex access control policies. Frank Mayer describes the SELinux policy model as an "assembler level" policy. In the August 19, 2004 release, the default build results in a 500 KB compiled policy file. There are over 5,000 permission assignment (allow) rules in the policy itself (in the file policy.conf). Note that this policy contains just the base subjects; the complete policy, including policies for all shipping applications, is about ten times greater in size. As a result, understanding the higher-level properties that a policy implies, such as information flow, cannot be done manually.

4.2 Supporting filtering interfaces in the MAC Policy

SELinux cannot distinguish among input interfaces in a single process. Some interfaces may only have to process high integrity data, such as the interface that reads a configuration file. Others have to be able to validate and upgrade certain types of low-integrity data such as network input: these are filtering interfaces. In order to support filtering interfaces (and therefore to check CW-Lite), we modified the SELinux user space library and kernel module to support two subject types per process instead of one. The default subject type is used for ordinary operation and allows inputs only from subjects in the application's TCB; this is enforced by the Gokyo policy analysis in Section 4.3. The new *filtering subject type*, with additional permissions, is used for interfaces with the appropriate `DO_FILTER()` source code annotation.

```
DO_FILTER(interface creation code) :=
  use_filtering_subject_type();
  interface creation code
  use_default_subject_type();
```

The annotation serves as a contract with the programmer, who stipulates that input from the interface is filtered. Our macro-like approach is deliberate, to discourage running a large amount of code with higher privilege. Typically, only a single `open()`-type call requires the permissions. An example of the required changes to the program and the security policy for filtering interfaces is given in Figure 4.2.

Note that the `accept()` system call is still constrained by the MAC policy for the filtering subject type. For example, the filtering subject type permissions for the application might allow accepting connections from one network card, but not another.

Before

Source Code

```
conn = accept()
// accept() fails
get_http_request_sanitized(conn)
```

Security Policy (default DENY)

```
Apache:  ALLOW read httpd.conf
// Problem:  network ∉ TCB!
Apache:  ALLOW accept
```

After

Source Code

```
DO_FILTER(conn = accept())
// accept() succeeds
get_http_request_sanitized(conn)
```

Security Policy (default DENY)

```
Apache:  ALLOW read httpd.conf
// network officially ∉ TCB
Apache-filter:  ALLOW accept
```

Figure 3. Supporting filtering interfaces. Initially, the program above is not allowed to accept network input, because the network is not in the TCB. In order to accept such input, the source code must filter it and the programmer must supply an annotation indicating that the interface is filtered. Then the policy must be modified to allow the network input only for the filtering interface. The `DO_FILTER()` macro tells the MAC system to use the filtering subject type permissions for the enclosed operations. We annotate `accept()` (which implies a read/write socket), rather than subsequent socket read/write operations, because that is where the MAC system performs access checks. This is analogous to how file access checks, including read/write permission checks, are performed once on `open()`, not for every `read()` or `write()` call.

4.3 MAC Policy Analysis

Once the target application's untrusted inputs have been isolated into its filtering subject types, we need only check that there are no untrusted inputs to the application's default subject type s.

We employ the Gokyo tool to compute information flows from an SELinux policy [14]. Gokyo represents access control policies as graphs where the nodes are the SELinux subject types and permissions, and the edges are assignments of permissions to subject types. Based on whether the permission allows a mod operation, an obs operation, or both, Gokyo computes all information flows to s. That is, it computes the set of subject types

$$F = \{s' : mod(s', o) \wedge obs(s, o), o \text{ is an object type}\}.$$

Gokyo also correctly handles flows implied by object relabeling; see the Appendix for details.

Some of the non-target subjects may be designated as *trusted subjects*, and they form the system's TCB. The TCB includes subjects such as those that bootstrap the system (e.g., *kernel* and *init*), define the MAC policy (e.g., *load_policy*), and do administration (e.g., *sysadm*).

Given the set of information flows and the TCB, the untrusted subjects with flows to s are given by

$$U = F \cap \neg TCB.$$

If this set is empty, then CW-Lite holds for the target application. If not, Gokyo outputs the set of permission assignments P that need to be examined, i.e., those that allow the offending mod and obs operations:

$$P = \{p : flow(u, s), p \Rightarrow mod(u, o) \vee p \Rightarrow obs(s, o)\}$$

where p is a permission assignment, $u \in U$, and o is some object type.

4.4 Finding filtering interfaces

Although we modified SELinux to support mediation for filtering interfaces separately from other interfaces (Section 4.2, above), the developer still needs to make annotations to tell SELinux whether a given interface performs filtering or not. As part of this process, the developer needs to determine which interfaces require filtering. Some may be obvious, but there may be permissions to access untrusted data that are used in a subtle way. The developer can find these by running the security policy analysis on the default policy and analyzing all integrity-violating permissions for the application.

The problem of determining where in a program a permission is used is outside the scope of SELinux's goals, so we implemented our own mechanism. We defined a new operation in SELinux called `settraceonerror` using the *sysfs* interface and made appropriate changes to both SELinux's user library and its kernel module. When `settraceonerror(true)` is called from user space, our modified SELinux kernel module signals the process whenever a violation of the SELinux policy is found. The user library catches the signal and traps the process into a separate xterm debugger (gdb). If the process forks, additional xterm windows with debuggers on the child processes are launched. Once in the debugger, it is much easier, using stack traces and data inspection, to determine where and why a permission error occurred and take appropriate action, either removing the offending operation or implementing a filtering interface. If the permission is never actually needed, then it can simply be removed from the policy.

Some filtering interfaces may not need to actually filter the incoming data contents, since some interfaces do not interpret the incoming data. For example, *logrotate* enables automatic rotation of log files, but does not depend on the data in the files. Likewise, the *cp* utility copies files, but does not consider their contents. In these cases, a `DO_FILTER()` annotation is still appropriate, rather than allowing the program to accept all inputs. This is because (1) filtering based on metainformation (like input length) may still be needed; and (2) the kinds of inputs may need to be restricted (for example, disallowing copies from named pipes). Naturally, if the program semantics change later to include interpretation of the untrusted data, the programmer should implement additional filtering code.

One may wonder why we use a dynamic approach to finding filtering interfaces. A simple example is revealing: consider the interface `fd = open(filename)`. In order to decide statically if filtering is required, we would need to know the value of `filename`. This may be partially addressed with a program analysis, though of course it is undecidable and may come from dynamic data, say from the system's configuration. The mapping of filename to object type (which is what matters for integrity) is also system dependent, and each administrator may keep files in different locations. Our approach works well enough in practice, since the number of filtering interfaces is usually relatively small; while it may have the coverage problem of dynamic analysis, it does not have the scalability and decidability problems of static analysis.

4.5 Handling illegal information flows

If a sysadmin's invocation of the policy analysis tool detects illegal information flows implied by a set of permissions, one of a few actions is required. Some such permissions are simply unneeded and may be removed. Some information flows may be generated by programs that are untrusted, but optional to the system. An easy way to remove this information flow is to exclude the offending code and subject types from the system. Some permissions are needed by optional components of the target app; the options may be disabled, and the permissions removed. If the permission is used by the core application, then either the sysadmin may be assuming a smaller TCB than the developer or the developer has not added a `DO_FILTER()` annotation. The sysadmin can either not run the target application or get the developer to write and annotate additional filtering interfaces.

5 Example: CW-Lite Integrity Verification

5.1 Goal

So far, we have defined CW-Lite and shown in general how applications can be constructed to satisfy its requirements. We have several goals in applying CW-Lite to our primary example application, OpenSSH, and to vsftpd. First, we want to see how easy it is to build applications to meet CW-Lite in practice. Since OpenSSH is a very popular, complex, and security-critical program that has been architected to preserve the integrity of its privileged components, verifying a useful integrity property can be of value to of millions of systems and validate the security efforts of its developers. vsftpd is a somewhat simpler example, and illustrative of a common case. Second, we want to see how close the default SELinux policy is to enabling satisfaction of CW-Lite. Third, if either application (with the standard shipping policy) does not initially meet CW-Lite integrity, we want to see why it fails and how difficult it is modify the application or policy to enable success.

5.2 Setup

Provos *et al.* decomposed the server-side daemon of OpenSSH into privileged and unprivileged components in order to minimize the amount of code that needs to run with privilege. The privileged component exports a narrow interface to the unprivileged components, such that only specific operations in a specific order may be requested, which reduces the risk of the privileged component being compromised by a hijacked unprivileged component. Privilege-separation has been added as an option to the main OpenSSH distribution.

The process graph for privilege-separated OpenSSH is shown in Figure 4. One privileged component, *listen*, listens for new connections via port 22 and forks a new privileged component, *priv*, per connection. This *priv* component performs the privileged operations required by OpenSSH: authentication of the remote user, creation of pseudo-terminals, and transition to a particular, authenticated userid. The *priv* component in turn spawns unprivileged components to handle various types of user interaction. The *net* component is used to perform the remote interaction part of the authentication phase, which has in the past been subject to compromise; it uses the *priv* component as a privileged server to handle secret data operations. After successful authentication, *priv* spawns a shell or other process requested by the user in that user's security context.

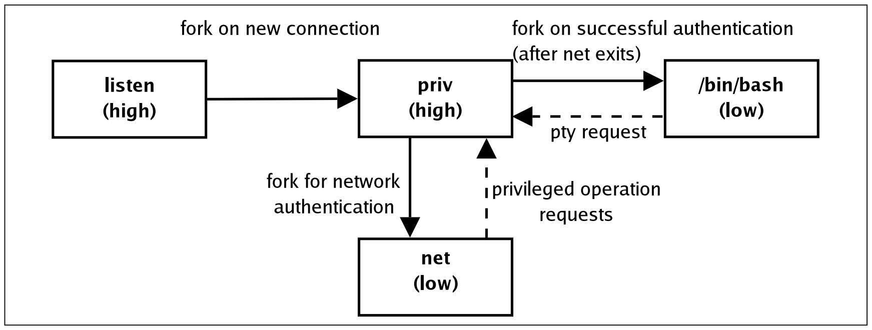
fork on new connection
fork on successful authentication (after net exits)
listen (high)
priv (high)
/bin/bash (low)
pty request
fork for network authentication
privileged operation requests
net (low)

Figure 4. Process structure of privilege separated OpenSSH. The *listen* process simply processes new connection requests, forking a *priv* process to handle each one. The *priv* process forks a *net* process to perform the network portion of user authentication, providing a narrow interface to privileged operations necessary to complete authentication. After *net* completes its task, *priv* spawns the authenticated user's requested process, in our example a bash shell.

vsftpd is the FTP server included with Fedora Core Linux. It too employs separate trusted and untrusted processes, though its policy treats both the same. We do not discuss its analysis in as much detail, but give a summary of the analysis and the results.

For testing, we used OpenSSH 3.6 and vsftpd 2.1.3 on an Intel x86 platform with the Linux 2.6 kernel installed. We use SELinux (see Section 4.1) as our MAC system, using the strict (not targeted) policy configuration for Fedora Core 4.

5.3 Roadmap for OpenSSH

The problem of verifying CW-Lite for privilege-separated OpenSSH is addressed by ensuring that all information flows into the privileged components (*listen* or *priv*) either contain only high integrity data or discard/upgrade the data via declared filtering interfaces.

Enabling OpenSSH to satisfy CW-Lite requires work by the application developer to modify OpenSSH to find where filtering interfaces are necessary, build acceptable filtering interfaces, and declare the presence of the filtering interfaces to SELinux. Then, the administrator of the SELinux system needs to configure an SELinux policy that enables satisfaction of CW-Lite. Recall that this policy will have a base set of permissions (subject type) allowing only trusted input for normal interfaces and additional subject type to accept requires types of untrusted input at the filtering interfaces.

The first step is to use the Gokyo policy analysis tool to identify the illegal information flows to *priv* and *listen*. The next task is to determine whether the remaining low integrity flows can be handled by filtering interfaces. We use our tools (see Section 4.4) to find the interfaces that accept low integrity data in the privileged components and add the `DO_FILTER()` annotation.

5.4 Inter-process Flow Analysis

Given the new SELinux policy for the OpenSSH components and the remainder of the SELinux example policy for the rest of the system, we are ready to use Gokyo to find low integrity information flows to the privileged OpenSSH components *priv* and *listen* and revise the policy to remove any unnecessary flows. Gokyo computes the information flows in the SELinux policy that violate the policy analysis constraints given the sets of trusted subjects, excluded subjects, and filter rules. A short introduction to Gokyo is in Section 4.3.

We define a TCB including the system bootstrap components, such as *bootloader*, *kernel* and *init*, and components that modify the SELinux policy itself (e.g., *checkpolicy*, *load_policy*, *setfiles*, etc.) or other objects upon which the system integrity depends (e.g., administrative subjects *sysadm*, *staff*, *rpm*, etc.).

We then run Gokyo and identify several information flow conflicts shown in Table 1. The table shows each instance where the target subjects (*priv* and/or *listen*) have a read permission on an object that may be modified by untrusted source subjects (*write-up subjects*); some objects may be written to by many write-up subjects. The problem then is to find a resolution that prevents the target subjects from being dependent on the write-up subjects. For each entry, one of these resolutions is applied in the following order of precedence: (1) we can EXCLUDE the write-up subject from the system if it is not required on the system (2) we can identify that the permission does not actually result in a data dependency (FILTER_NO_DEP), which requires a lightweight filtering interface that prevents only metainformation attacks like buffer overflows; (3) we can FILTER the

Target Subject	**Permission** (object:class)	**Source Subjects** (names or count)	**Resolution**
Resolutions requiring primarily system knowledge			
priv, listen, ftpd	devlog:sock	privlog	FILTER_NO_DEP
priv	lastlog:file	6	FILTER_NO_DEP
priv, ftpd	etc_runtime:file	xdm, hotplug	EXCLUDE
listen	initrc_var_run:file	7 (includes rlogind)	EXCLUDE
listen, ftpd	net_conf:file	dhcpc	EXCLUDE
priv, ftpd	wtmp:file	7 (includes rlogin)	EXCLUDE
Resolutions requiring application knowledge			
listen	sshd_listen:tcp (`accept()`)	[network]	FILTER
listen	userpty:chr_file	7	FILTER
priv	sshd_priv:unix	sshd_net	FILTER
ftpd	ftp_port_t:tcp (`accept()`)	[network]	FILTER
listen	sshd_listen:tcp (`read()`)	[network]	REMOVE
priv	xserver_port:tcp	165	REMOVE
priv, listen	devtty:chr_file	200	REMOVE (This row used for a sample walkthrough.)
priv,listen	port_type:tcp	[network]	REMOVE
listen	sshd_listen:unix	unpriv_userdomain	REMOVE
listen	sshd_listen_devpts:chr_file	5	REMOVE
priv	sshd_tmp:file	(staff/sysadm/user)ssh	REMOVE
priv	sshd_tmp:lnk	(staff/sysadm/user)ssh	REMOVE
priv	sshd_tmp:sock	(staff/sysadm/user)ssh	REMOVE
priv	sshd_tmp:fifo	(staff/sysadm/user)ssh	REMOVE
priv	system_chkpwd:fd	27	REMOVE
priv, listen	unpriv_domain:fd	33	REMOVE
ftpd	ftp_port_t:tcp (`read()`)	[network]	REMOVE*
ftpd	nfs_t/cifs_t:file	27	REMOVE*
ftpd	user_home:file	4	REMOVE*

FILTER = The permission is necessary, but requires a filtering interface. It should be put in the filtering subject type.
FILTER_NO_DEP = Necessary permissions that requires a filtering interface, but no semantic filtering is needed since the input is not interpreted. The filtering subject type is still needed to properly handle metainformation and enforce least privilege for other trusted inputs.
EXCLUDE = Exclude the source subject from the SELinux policy, as it causes insecure flows; any associated programs must not be run on the system. This is a judgment call; sysadmins may instead decide to add the source subject to the TCB.
REMOVE = Remove the permission assignment from the target subject, breaking the information flow.
* = The vsftpd policy did not fully reflect its process structure; see Section 5.6 for details.

Walkthough for shaded row: "priv, listen" indicates that the illegal flows were inputs to both the *priv* and *listen* components of OpenSSH. The object that they have permission to read from is devtty:chr_file, that is, a TTY from `/dev/tty`. Two hundred untrusted subjects have permission to write to that object. The illegal flows are broken by removing the read permissions, since they are not necessary: the TTY is actually read only by the *net* component, which handles remote user input.

Table 1. Information flows to our target subjects (*priv* and *listen* for OpenSSH and *ftpd* for vsftpd) that may lead to integrity problems. The permissions leading to these flows were identified by the Gokyo tool. The top half of the table indicates conflicts resolved based on system knowledge. The bottom half required examining the behavior of the target application using the tools described in Section 4.4. Each target subject was analyzed independently.

use of the permission via a filtering interface; or (4) we can REMOVE the permission assignment from the target subject or the write-up subject if not required by the subject.

The table groups the resolutions into two categories: (1) resolutions based on information flow only and (2) resolutions based on application configuration and information flow. The decision between removal of permission assignments and filtering generally requires application knowledge; some permissions are needed to support optional components of the application and some are needed for core operation. An administrator would need to decide which options were required on her system and trust the corresponding inputs. 14 of the 20 conflicts require some understanding of the needs of the OpenSSH application.

Next, we recognize that the use of *devlog* and *lastlog* does not result in any form of dependence. They manipulate log data which is not interpreted, for example by rotating log files.

Finally, we exclude a few write-up subjects from the system if they cause illegal flows to OpenSSH. This is a judgment call; a sysadmin may decide to trust these subjects instead. If they are trusted, then they must implement appropriate filtering interfaces. *dhcpc* is an example of this as some vulnerabilities have been found for it. Dynamic system extension via *hotplug* is not necessary in our environment, so our judgment is to simplify administration and exclude it. We also eliminate the untrusted subjects that write to the login records of *wtmp*, such as *rlogind*.

Only 3 of the 15 remaining read-type permissions are actually needed in our OpenSSH configuration: the permissions identified by Provos *et al* for creating the pseudo-terminal; initiating OpenSSH connections (by *listen*); and processing user commands via the socket from *net* to *priv*. We remove the 10 unnecessary permission assignments. We note that the *port_type:tcp* permission which permits access to most systems sockets is much coarser-grained than necessary. *listen* only needs access to *sshd_listen:tcp* on port 22. We note that the replication of some permissions for *listen* and *priv* was unnecessary. For example, there is no need for *listen* to accept requests from *net*.

Figure 1 shows how challenging it can be to get the permission assignments correct for a given system. The hand-constructed SELinux policy shipped with Fedora Core 4 contained several permissions that needed to be removed. (Our hand-distribution of OpenSSH permissions to *net*, *priv*, and *listen* did not impact these.) Some, such as for *sshd_tmp*, enable actions that we do not want in our configuration (e.g., user administration). Others, though, are simply mistakes that enable information flows that could compromise the integrity of our privileged components. While investigating the source of these errors, we found that, often, large blocks of permission assignments were made using SELinux convenience macros when only a subset were actually needed. SELinux policies only allow assignment of permissions, not their removal, so we urge policy writers to be careful in their use of such macros.

5.5 OpenSSH Filtering Interfaces

We now describe how we identified which permission assignments to classify as FILTER. First, the OpenSSH application developer needs to find where filtering interfaces are necessary. A filtering interface is necessary where low integrity data may be input. For OpenSSH, the interfaces where *listen* receives connection requests from the network and where *priv* receives commands from *net* are the two obvious cases. However, other interfaces may also require filtering in OpenSSH. To find all filtering interfaces, an analysis of the SELinux policy is necessary to see if low integrity inputs may be used by other OpenSSH interfaces.

We use the `settraceonerror` mechanism described in Section 4.4 to test our configuration against the default SELinux policy to determine if other interfaces besides the two above require filtering interfaces. We located one: the *userpty* pseudo terminal used by *priv* to communicate with the user shell process.

Next, the application developer must construct effective filtering interfaces. It is the application developer's task to build the filtering interfaces and prove effectiveness to the community. For OpenSSH, the construction of a filtering interface for *priv* to read commands from *net* is one of the main tasks in the privilege-separation done by Provos *et al* [21]. The interface to accept connections in *listen* does not have any special filtering per se, as the connection is not interpreted by *listen*. Also, the *userpty* pseudo terminal in *priv* is only used to pass data to the remote user from the shell process with an encryption step; the contents are not examined.

Finally, once filtering interfaces are found, they must be declared to SELinux in order to use the low integrity permissions. We use the `DO_FILTER()` annotation to declare such interfaces as described in Section 4.2.

5.6 Verifying vsftpd

We applied the same approach to verifying vsftpd; the results are in Table 1. One difference is that the SELinux policy did not reflect the nature of the FTP daemon, which forks per-connection helper processes in a manner very similar to OpenSSH. Instead, there was one subject type for all processes. The child processes do drop Linux privileges (versus than SELinux ones), so they are still largely confined (if a permission is denied in either model, it is denied to the process). The three starred permissions in the table are those that should belong to the unprivileged child processes only, not to the trusted server process, which is why we specify their disposition vis-a-vis the trusted subject type as "REMOVE". The interface between the two is a filtered domain socket. The additional violating permissions were eliminated by excluding some of the same excluded subjects as for OpenSSH, like *rlogind* and *xdm*.

6 Related Work

6.1 Integrity Models

System integrity has been a difficult problem for security researchers over the years. Most work on integrity has focused on information flow models, supplemented by high assurance (i.e., formal, validation of program correctness, such as Common Criteria EAL7 evaluation).

The Biba integrity model [2] is essentially a dual of the Bell-LaPadula secrecy model [1], where information flows from low integrity subjects to high integrity subjects are prohibited. Like Bell-LaPadula, high assurance components are required to overcome restrictions, but unlike the case for secrecy, illegal (low-to-high) integrity information flows are common (e.g., user requests).

Attempts in subsequent models have not grappled with the fundamental problem that low-to-high integrity flows are common. Denning's work on secure information flow models [8] models information flows between subjects of different labels as a lattice, but models illegal flows as any flow that violates the lattice structure. The LOMAC (low watermark) integrity model also prevents high integrity subjects from acting on low integrity information flows, in this case by downgrading the level of a high integrity subject upon receipt of a low integrity information flow [12]. The Clark-Wilson model acknowledges that interfaces are required that can sanitize (or discard) low integrity data, but all interfaces must be capable of sanitization (or discard) and the basis for trusting these interfaces is still high assurance [7]. We are significantly influenced by the Clark-Wilson model's view of requirements on interfaces of high integrity processes, though. Lastly, the recent Caernarvon model allows subjects to span multiple integrity levels such that a subject (i.e., process running with an integrity label range) may be able to read from lower integrity data within its integrity range securely while writing to higher integrity data within its integrity range [22]. Unfortunately, the integrity ranges must be justified by assurance, where significantly broad ranges will still require high assurance. Even after 25 years, we cannot escape the requirement for high assurance, which places too high a burden on too many applications to be practical.

More recent work by Li and Zdancewic [18] present a formal type system that captures intraprogram labeled information flow, with provisions for downgrading of data; type-checking may be used to ensure information-flow security. One may imagine applying their method to interprocess flow, which is controlled by a security policy rather than program source. The `DO_FILTER()` primitive we present may be seen as a downgrading operation in this context.

6.2 Static Analysis

Several access control policy analysis tools have emerged, particularly in the context of SELinux. While the early tools mainly supported query handling, recent tools, such as Gokyo [14], SLAT [13], and Apol [25], now support different kinds of information flow analysis. For example, SLAT enable verification of particular information flow policies, and Gokyo identifies and enables resolution of illegal information flows [16]. Understanding information flows is key to achieving CW-Lite integrity.

Static analysis has also been used to separate trusted and untrusted code in other ways. The Privtrans system [5] uses source code analysis to try to automate privilege separation, dividing a program into trusted and untrusted processes.

6.3 Whole-system Analysis

While there is widespread agreement that whole-system analysis is desirable, there have been relatively few efforts that actually do so on widely-used operating systems. Recently, Chow et al. used hardware-level simulation on a virtual machine in order to perform a dynamic cross-process taint analysis [6]. By contrast, our work focuses on static analysis to prove certain properties about applications' information flow rather than infer them dynamically. Thus, we see our approach as complementary to theirs.

7 Conclusion and Future Work

Maintaining information-flow integrity is an old and important problem, but as yet an unsolved one in practice: most administrators have not verified that untrusted users cannot compromise inputs to trusted programs on their systems. In this paper, we have developed a way to automatically verify a meaningful information-flow integrity property with very small changes to existing trusted applications. We call this property CW-Lite, since it has the same interprocess dependency semantics as the well-established Clark-Wilson model, but does not address Clark-Wilson's whole-program formal verification requirement. CW-Lite requires filtering only for untrusted input interfaces, as determined by the system's security policy, and just simple annotations to existing applications to enable least-privilege enforcement and automatic verification. Only conflict resolution requires manual effort by the sysadmin. We modified the SELinux access control system to enforce CW-Lite and developed tools that support the implementation of compatible program. We verified the practicality of our tools by analyzing privilege-separated OpenSSH and vsftpd, finding and fixing several integrity-violating configuration errors in the shipping SELinux policy.

The main future work is to extend CW-Lite to the entire software trusted computing base of the system. In addition, a natural extension to our model would be a way to prove statically that filtering interfaces' upgrade/discard operation is semantically correct, an application-specific check that may be tractable if the types of allowed filtering are restricted.

Acknowledgements

David Wagner, Rob Johnson, Naveen Sastry, Monica Chew, AJ Shankar, David Molnar, and Chris Karlof provided valuable technical and stylistic feedback on this paper.

References

[1] D. Bell and L. La Padula. Secure Computer Systems: Mathematical Foundations (Volume 1). Technical Report ESD-TR-73-278, Mitre Corporation, 1973.

[2] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, Mitre Corp, Bedford MA, June 1975.

[3] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2003

[4] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8*th *National Computer Security Conference*, Gaithersburg, Maryland, 1985.

[5] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, August 2004.

[6] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, August 2004.

[7] D. . Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, May 1987.

[8] D. E Denning. A Lattice Model of Secure Information Flow. In *Communications of the ACM*, vol. 19, no. 5 (May 1976), pp. 236-243.

[9] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S. W. Smith, L. van Doorn, and S. Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, (34)10:57-66, October 2001.

[10] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, Jan/Feb 2002.

[11] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *IEEE Computer*, pp. 55-62, July 2003.

[12] T. Fraser. LOMAC: Low Water-Mark integrity protection for COTS environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000.

[13] J. Guttman, A. Herzog, and J. Ramsdell. Information flow in operating systems: Eager formal methods. In *Workshop on Issues in the Theory of Security (WITS)*, 2003.

[14] T. Jaeger and A. Edwards and X. Zhang. Policy management using access control spaces. *ACM Transactions on Information and System Security (TISSEC)*, 6(3), August 2003.

[15] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, August 2003.

[16] T. Jaeger, R. Sailer, and X. Zhang. Resolving constraint conflicts. In *Proceedings of the* 9^{th} *ACM Symposium on Access Control Models and Technologies*, June 2004.

[17] D. Kilpatrick. Privman: A library for partitioning applications. In *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX Track*, April 2003.

[18] P. Li and S. Zdancewic. "Downgrading Policies and Relaxed Noninterference". In *Proceedings of the 2005 Symposium on Principles of Programming Languages*, January 2005.

[19] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, May 1998.

[20] NIST, Computer Security Division. Common criteria for IT security evaluation. Available from `csrc.nist.gov/cc/`, 2004.

[21] N. Provos, M. Friedl and P. Honeyman. Preventing privilege escalation. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, August 2003.

[22] G. Schellhorn, W. Reif, A. Schairer, P. Karger, V. Austel, D. Toll. Verification of a formal security model for multiapplicative smart cards. In *Proceedings of the* 6^{th} *European Symposium on Research in Computer Security (ESORICS)*, 2000.

[23] National Security Agency. Security-Enhanced Linux (SELinux). `http://www.nsa.gov/selinux`, 2001.

[24] S. Smith. Outbound authentication for programmable secure coprocessors. In *Proceedings of the* 8^{th} *European Symposium on Research in Computer Security (ESORICS)*, 2002.

[25] Tresys Technology. Security-Enhanced Linux research. www.tresys.com/selinux.html, 2004.

[26] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.

[27] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.

Appendix: Gokyo support for SELinux object relabeling

Because the SELinux policy model also permits object relabeling, we must consider information flows caused by modifying an object and relabeling it to another object type. The $relabel(s, obj, o, o')$ operation enables subject s to change an object obj's label from o to o'. Since relabeling does not change the contents of an object, we do not really care who does the relabel, just that it can occur. Also, it does not matter which specific object can be relabeled, since all objects of the same object type are equivalent from an information flow perspective. Thus, we use a refined predicate $relabel(o, o')$.

Next, we consider successive relabeling operations $o_1 \rightarrow o_2 \rightarrow ... \rightarrow o_i$. The transitive closure of the $relabel$ operation is defined by $\overline{relabel}(o_1, o_i)$. The *relabel information flow rule* states that

$$mod(s_1, o_1) \wedge \overline{relabel}(o_1, o_i) \wedge obs(s_i, I, o_i) \\ \rightarrow flow(s_1, s_i, I).$$

Gokyo accounts for information flows due to arbitrary relabeling.