From Trusted to Secure: Building and Executing Applications that Enforce System Security

Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel
Systems and Internet Infrastructure Security Laboratory (SIIS)
Computer Science and Engineering, Pennsylvania State University
{*phicks,ruedarod,mcdaniel,tjaeger*}*@cse.psu.edu*

Abstract

Commercial operating systems have recently introduced mandatory access controls (MAC) that can be used to ensure system-wide data confidentiality and integrity. These protections rely on restricting the flow of information between processes based on *security levels*. The problem is, there are many applications that defy simple classification by security level, some of them essential for system operation. Surprisingly, the common practice among these operating systems is simply to mark these applications as "trusted", and thus allow them to bypass label protections. This compromise is not a limitation of MAC or the operating system services that enforce it, but simply a fundamental inability of any operating system to reason about how applications treat sensitive data internally—and thus the OS must either restrict the data that they receive or trust them to handle it correctly.

These practices were developed prior to the advent security-typed languages. These languages provide a means of reasoning about how the OS's sensitive data is handled *within* applications. Thus, applications can be shown to enforce system security by guaranteeing, in advance of execution, that they will adhere to the OS's MAC policy. In this paper, we provide an architecture for an operating system service, that integrate security-typed language with operating system MAC services. We have built an implementation of this service, called SIESTA, which handles applications developed in the security-typed language, Jif, running on the SELinux operating system. We also provide some sample applications to demonstrate the security, flexibility and efficiency of our approach.

1 Introduction

The problem of building secure systems with mandatory policy to ensure data confidentiality and integrity is coming into the forefront of systems development and research. Mandatory access controls (MAC) for type enforcement (TE) along with support for multi-level security (MLS) are now available in the mainline Linux distributions known as Security Enhanced (SE)Linux [25]. Trusted Solaris [21] and TrustedBSD [9] also provide MAC security. A recent release [35] made SELinux-like security available for Mac OS X, as well. Other projects such as NetTop [19] (which is being built with SELinux) seek to provide strong assurance of data separation. The goals of data separation (called *noninterference* elsewhere in the literature [10]) hold an important place also in the recent efforts towards virtualization with VMware [8], Xen [2] and others. All of these efforts run into a critical problem—they seek to enforce security only at the granularity of application inputs and outputs. *They cannot monitor how data is handled within an application.*

This approach would be acceptable if each application instance only handled data at a single security level. If that were the case, the operating system could prevent an application from reading or writing at any other level. The reality, however, is that many applications must simultaneously handle inputs and outputs with different security levels. This problem has led to two ad hoc approaches, both of which have serious limitations. The first approach sacrifices security to improve flexibility and efficiency. By marking an application as "trusted," it is given a special status to handle inputs and outputs with varying security policies. The operating system must then presume that the application will internally handle the data correctly. At best, the application's code is subjected to a manual inspection. The second solution compromises flexibility and efficiency in order to ensure security. Applications that must handle inputs with differing security levels are split into multiple executions, with one to handle each level. This complicates legitimate communication between processes and expends system resources, making it slow, error-prone and not always sufficiently expressive.

The first solution has been applied primarily to system utilities. A quick check indicates that SELinux, for example, trusts dozens of applications (30-40 SELinux application types have special privileges for this purpose) to correctly handle data of multiple levels: consider passwd, iptables, sshd, auditd, and logrotate, to name a few (a complete list is given in Figure 1). As an example, logrotate handles the data from many different levels of logs as well as its own configuration files. It also runs scripts and can send out logs via email. Even after a thorough inspection of the code, it is hard to say with certainty that it never leaks log data to its (publically readable) configuration files or improperly sends mail to a public recipient (in fact, as of v. 2.7.1, this could actually happen).

There are a number of user applications, for which the second approach is more common. Email clients are a classical example, but web browsers, chat clients and others also handle secrets such as credit card numbers and passwords along with other mundane data. Some servers (web servers, chat servers, email servers) also fall into this category, handling requests from various levels of users and thus requiring multiple versions of the same application to run simultaneously.

What we would like is to be able to communicate the operating system's data labels (the label on files, sockets, user input, etc.) into the application and ensure that, throughout the application, the labeled data flows properly (i.e., in compliance with the operating system policy). Fortunately, a new technology has become available to aid in this process. Emerging security-typed languages, such as Jif [22], provide *automatic verification* of information flow enforcement *within* an application. Through an efficient, compile-time type analysis, a security-typed language compiler can guarantee that labeled data flowing through an application never flows contrary to its label. This provides a formal basis for trusting applications to handle data with multiple levels of sensitivity. Admittedly, these technologies are still in development and thus still challenging to use. Programming in Jif can be a frustrating endeavor. To aid this, we are investigating tools for semi-automatic labeling of programs. That said, we found that Jif is tractable in its current form for programming some small utilities and user applications which require handling of multilevel data. Such utility presents an as yet unrealized opportunity to improve broader systems security. To our knowledge, there has been no investigation of the ways in which these application guarantees could be used to augment greater system security.

To this end, we have designed and built an infrastructure that 1) allows an operating system with mandatory access controls to pass labeled data into an application and 2) to be certain that the data will not flow through the application contrary to the operating systems policy. For our investigation, we have focused on the most mature security-typed language, Jif/Pol (Jif enhanced with our policy system [13]) and the widely-studied, open source, SELinux operating system. Because these languages have not yet been widely used, there is no infrastructure available for interacting with secure operating systems. To remedy this, we have provided 1) an API for Jif by which labeled data such as sockets, files and user I/O can be received from and passed out of the OS—this API ensures consistency between operating system and application labels. 2) We provide a compliance analysis that ensures that the labeled data will be handled securely within the application, in compliance with the OS's mandatory policy. We integrated these changes into an operating system service we call SIESTA, that can be used to securely execute multi-level applications written in Jif, by first verifying that they will not violate the operating system's security policy.

To demonstrate the effectiveness of our approach, we used Jif/Pol to build some prototype applications: a security-typed version of the logrotate utility and an email client that can handle multiple email accounts of varying security levels. For logrotate, we were able to determine that it is possible to have total separation between log files of different programs and between log files and configuration files, so long as the configuration files have a lesser or equal confidentiality than the log files.

In this work, we make the following contributions. We identify a fundamental limitation in MAC systems security and we show how recent advances in programming languages can be applied to solve this problem. More specifically, we give a clean, general architecture for using security-typed language technology to enforce system security *within* applications. To test our approach, we implemented this architecture for Jif and SELinux and provide some reusable software artifacts. Namely, we extend the Jif Runtime environment to provide a reusable API for reading and writing OS resources labeled with SELinux security contexts. We also give a policy analysis which tests Jif policy for compliance with SELinux policy. Additionally, we provide a system service, SIESTA, that incorporates this analysis tool and uses it to determine whether a Jif application can be securely executed in a given SELinux operating system. Finally, we evaluate our implementation for security, flexibility and efficiency using some example applications we constructed.

In Section 2, we give some background on MAC security and security-typed languages, we also describe the problems involved with integrating security-typed languages into MAC OS's. We give our architecture in Section 3 and describe the implementation of this architecture and some demonstrative applications in Section 4. We evaluate these applications, as well as our approach, for its usability, efficiency and security in Section 5. We examine some related work in Section 6 and we conclude in Section 7.

2 Problem

2.1 Security background

Security lattices Standard information flow models, on which we base our work, arrange labels on data as a lattice of principals, sometimes called a *principal hierarchy*. The traditional model [3, 7] allows data only to flow up the lattice (i.e. data can become more secure, but not less secure). If, for some reason, data must flow down the lattice, a *declassification* must take place. These policy violations should be infrequent or non-existent and if occurring at all should be carefully regulated. Filters for regulating declassification are called *declassifiers*.

Lattices may have a variety of principals and structures. A standard military lattice is simply a vertical line containing five levels[1]: unclassified, classified, confidential, secret and top secret, with top secret placed on the top of the lattice and unclassified on the bottom. While unclassified data can be written to classified files, the opposite is not true.

We use the term "MLS" broadly throughout this paper. Although the term has traditionally referred to military levels of sensitivity, such as secret or top secret, more general lattices can also be expressed [7]. For example, consider the lattice in Figure 3. In this lattice, data labeled `configP` can be written up to `xserver_log_t:s1`. The principals at the top of this lattice are all *incomparable* with each other. This means that data cannot flow between these labels at all. They could only be written to a mutually higher principal.

2.2 Enforcing MAC policies within applications

In an OS with MAC security, the OS can monitor its resources (such as files, sockets, etc.) and when an application tries to read them, write them, delete them, etc. it can prevent the application from performing one of these security-sensitive operations. To accomplish this, OS entities are divided into subjects and objects. Every operating system resource (socket, file, program, etc.) is an object, labeled with a type and an MLS level[2], such as `system_u:object_r:user_t:s0` for a public object owned by the reader. We will abbreviate this as `user_t:s0` since the user and role labels are always the same for system resources. The running process is considered a subject and also has a label, such as `system_u:system_r:logrotate_t:s0-s1`, where the colon-separated label consists of user, role, type and MLS level (reading the label from left to right). Notice that the MLS level may consist of a range, indicating that a particular process can handle a range of levels. In this case, the subject would have access to objects in the range `s0` to `s1`.

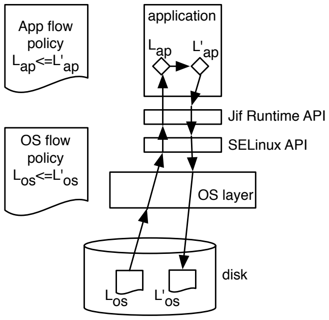


Figure 2: As data passes from a disk through the OS into an application and again when it is written back out, there must be consistency in labels and permitted flows at the OS and application levels. This requires proper labeling and compliance of the application policy with the operating system policy.

If a running process for `logrotate`, for example, has this label and attempts to read from a user's file labeled with `user_t:s4`, a security check will be triggered by this security-sensitive operation and (under a typical policy) it will be stopped by the Linux Security Module (LSM). However, if `logrotate` has permission to read from a log file labeled `var_log_t:s1` and to write to a configuration file labeled `logrotate_var_lib_t:s0` (which it normally does), the OS cannot stop it from reading the log data and leaking it down to the lower security configuration file. This could leak secrets stored in the log data to the publicly readable configuration file. Currently, the `logrotate` utility and the other utilities in Figure 1 are merely trusted not to leak data and it is not easy to verify, by manual inspection, that the C code for these utilities does not contain such a leak.

What is needed is 1) a way to pass the security labels into the application along with the resources, and 2) an automated way to ensure that the application honors the flow requirements on the labels. Furthermore, both of these conditions should be checked prior to executing the application. This situation is illustrated in Figure 2. Because the second requirement is precisely what security-typed languages, such as Jif, do well, we consider how they might be used to meet this need.

In Jif, when a variable is declared, it is tagged with a security label. An automated type analysis ensures no leakage can occur through implicit or explicit flows. For example, consider a program which has been executed by Alice (who can enter information through `stdin` and read from `stdout`), but which also has access to files, some of which can not be accessed by Alice (like persistent state such as statistics from others' executions) and some of which are publicly readable (like `config`

Type of utility	Trusted applications
Policy management tools	`secadm`, `load_policy`, `setrans`, `setfiles`, `semanage`, `restorecon`, `newrole`
Startup utilities	`bootloader`, `initrc`, `init`, `local_login`
File tools	`dpkg_script`, `dpkg`, `rpm`, `mount`, `fsadm`
Network utilities	`iptables`, `sshd`, `remote_login`, `NetworkManager`
Auditing, logging services	`logrotate`, `klogd`, `auditd`, `auditctl`
Hardware, device management	`hald`, `dmidecode`, `udev`, `kudzu`
Miscellaneous services	`passwd`, `tmpreaper`, `insmod`, `getty`, `consoletype`, `pam_console`

Figure 1: A list of trusted applications in the SELinux release for Fedora Core 5 using mls-strict policy version 20.

files). In the following code (written in Jif syntax), data is read from the keyboard on line 1 and properly stored in a variable labeled with Alice's policy. In line 2, the label on `leak` can be inferred as `{alice:}`. Then a file is opened to write out configuration information (which is publicly readable). A leak occurs, however, when the program attempts to write Alice's data out to the configuration file. This code also contains a security violation in line 10, because statistics, which Alice should not be able to access, are printed to the screen. The typechecker would flag these errors and prevent this program from compiling.

```
1.  String{alice:} secret = stdin.read();
2.  String leak = secret;
3.  FileOutputStream[config] conf =
4.    Runtime.openFileWrite("tool.conf",{config:});
5.  conf.write(leak);
6.  FileInputStream[state] statsFile =
7.    Runtime.openFileRead("stats.dat",{state:});
8.  String stats = statsFile.readLine();
9.  if (stats.split[0].equals("bob"))
10.   stdout.write(stats);
```

Returning to Figure 2, we can presume that objects stored in the system are already labeled (with an OS label, L_{os}, for example), but we still need an OS API to get the labels and provide them to the application. Additionally, this must connect into a language-provided API to translate these labels into labels that the application can enforce (L_{ap}). This must be a carefully controlled interface so that the labels cannot be manipulated or spoofed. Finally, the information flows that the application will enforce must *comply* with the information flows enforced in the operating system. In the example, if the operating system policy were to state that $L'_{os} \leq L_{os}$ (L' is less confidential than L), but the application policy still has $L_{ap} \leq L'_{ap}$ (L is less confidential than L'), then the application would violate the operating system's policy.

Compliance testing is complicated by the mismatches between the lattices used by the operating system and those used by the application for enforcing information flow. Firstly, there may be principals in each lattice that are not found in the other. These cannot merely be removed, because they might connect shared principals and be involved in information flows. Secondly, there may be a mismatch in the kinds and granularity of permissions that the OS handles (set attribute, open, link, delete, read, etc.) compared to the application. Finally, the application policy could be more restrictive than the OS policy, but the reverse should not be true.

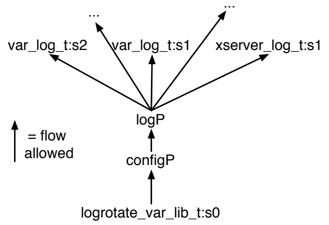


Figure 3: The lattice of principals describing all possible flows for `logrotate`. The two basic levels are logs and configuration files. The data in configuration files affect the logs but the reverse should not be true. Also, the logs should not be able to flow into each other.

For example, consider the lattice we constructed for `logrotate` in Figure 3. `logrotate` only needs to handle two kinds of files—the log files and the configuration and state files. Furthermore, the log files should be disjoint from each other and more sensitive than the configuration and state files. In this lattice, `configP` and `logP` do not have corresponding principals in the operating system. Also, we can see that this policy is more restrictive than the OS policy (notice that `var_log_t:s1` is normally be able to flow into `var_log_t:s2` according to OS policy, but not in this lattice). Also, this policy does not capture all possible OS principals. Finally, this policy only describes basic information flows–read and write.

We identify the following tasks. (1) We need a mechanism by which an application can prove that its information flow enforcement does not violate the system information flow policy. Both Jif/Pol (JP) applications and SELinux express the information flow policies that they are enforcing, so we need an approach to compliance testing the application policy against the system policy. This must include some control of application-level declassification preventing both unacceptable declassification filters and also the overuse of applications with declassifying filters. (2) We need mechanisms for the application to determine the label of its input channels necessary to enforce information flow. If JP applications cannot distinguish between secret and public inputs, it must label them all secret to enforce information flow requirements, thus impacting usability. (3) The system must be able to determine the label of all JP application outputs. Again, the lack of an accurate label would either result in overly conservative enforcement (i.e., the application may only send secrets) or possible vulnerabilities (i.e., the application sends a secret to a public entity).

To summarize, these considerations motivate the following concrete problems:

1. How can we pass operating system resources along with their labels into an application?
2. How can we pass application data along with their labels out into the operating system?
3. How can we be sure that the application will faithfully enforce the operating system's policy on these labels?

With the solution of these problems, we have a guarantee of system information flow enforcement, based on reconciliation of information flow enforcement and accurate communication of information flow labels between application and system layers. In the next section, we provide an architecture that solves these problems. In Section 4, we give the details of our implementation of this architecture for Jif and SELinux.

3 Architecture

In this section, we provide a general architecture for solving the problems described in Section 2. Namely, we describe the necessary steps for ensuring that a security-typed application can handle data with a range of security levels and still enforce the information flow goals of the OS. Note that this architecture is independent of any particular language or OS. We describe, in general, the features that are required for our approach. In Section 4, we will describe our implementation of this architecture for Jif and SELinux.

3.1 Process steps

We begin with a description of the overall process and then focus on the details of various steps in the subsequent subsections. Our five process steps are illustrated in Figure 4.

0) Initial state The OS must have a MAC policy implementing some information flow security goals. We focus on SELinux in this paper, but this could include other high assurance operating systems such as Trusted-Solaris or TrustedBSD. The key is that there must be an explicit MAC policy that is accessible to a system daemon for analysis of confidentiality policies. (Here we focus on ensuring confidentiality, but other information flow goals could also be examined.)

1) Program secure application An application developer provides the bytecode for a security-typed application along with a policy template that can be specialized by the user for a particular operating system configuration. We have used Jif/Pol in this paper, but the concepts extend to any security-typed language. A key point is that the language must provide a policy system such that each application will have an explicit policy that can be analyzed by a system daemon to understand the security lattice and declassifiers the application uses. We discuss this further in Section 3.2.

2) Specialize application policy Although a program will be developed with some basic security goals in mind, the application policy may be customized for different users running on different systems. This is especially important because the application policy must make connections to operating system label names which may not be the same from system to system. Of course, a reference policy should always be provided by the developer which should run on a default system configuration. The reference policy also serves as a template for customization to a customized OS. We discuss this further in Section 3.3.

3) Invoke service In an MLS environment, a user may have the authority to run at various security levels, but typically only logs in at one level at a time. In our approach, when he desires to run an application with a range of levels, he must first invoke an operating system service to check the application for compliance with operating system security goals. There must be no way to subvert this, i.e. to run the application without allowing the system first to perform the necessary checks. This should be enforced in the system policy.

The operating system service performs checks based on four inputs: the system policy, the object code for the application, the application policy and the desired range of levels. We discuss this further in Sections 3.4 & 3.5.

4) Run application If all the checks succeed, the operating system service may launch the application at the requested security level range.

3.2 Programming infrastructure

To address the last two problems listed in Section 2, namely that operating system resources—both inputs and outputs—must be labeled properly in the application, operating system and language APIs are necessary. First, the operating system API must offer procedures for an application to get labels on files, sockets and other OS resources. It must also be possible for the application to set labels on resources when they are created by the application. Secondly, the security-typed language API

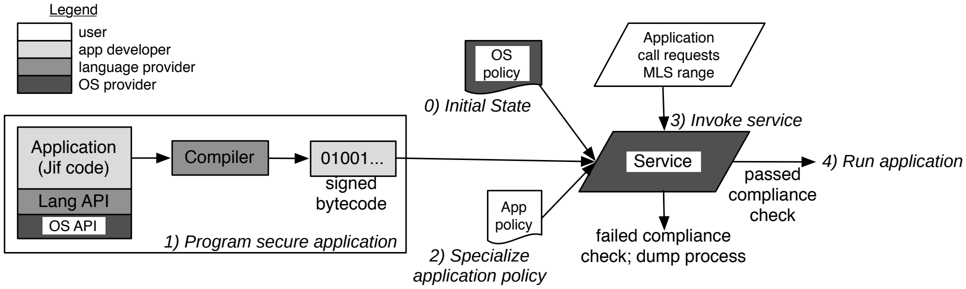


Figure 4: The process for executing an application with range of MLS privileges consists of 5 steps. The steps are performed and components are provided by different entities as shown by the different colors.

must supply procedures for getting and creating operating system resources. The primary concern is that these API abstractions provide the *only* way to access operating system resources. One solution, the one we use, is to provide a single way of creating new, or opening existing resources. With this approach, when an application's data structure is mapped onto a system resource, the internal label assigned to the data structure can be checked to correspond with the external label on the system resource. Thereafter, throughout all possible program executions, the normal security-type analysis provided by security-typed languages can ensure that that label is never violated.

3.3 Specializing application policy

Our approach assumes that a developer will construct applications that enforce some security goals. For example, a program variable into which a secret password will be read should be labeled differently from a variable that will contain public information. This must be part of the program code. The *meanings* of these labels are established in a high-level application policy external to the program code, however, and configured according to user preferences and system policy. For example, the public information could be treated just as secretly as the password if the user desires. Furthermore, the application developer will not know the names of security labels on the user's operating system; these must be configured by the user in light of the operating system policy. Another consideration is that a user may prefer not to use certain declassifiers in a particular application; this should be customizable in the application policy. Once the application policy has been specialized, the policy and application can be passed to an operating system service for compliance checking and execution.

3.4 Verifying run requests

The operating system service must be on the critical path for running any security-typed application, because it will ensure that all three requirements listed in Section 2 are met. Namely, it will ensure that 1) the labels passed into and 2) out of the application are consistent with the operating system labels and it will ensure that 3) the application will enforce the operating system's policy throughout its execution. To do this, the service must make four checks: (1) the application's information flow policy must be provably compliant with the operating system policy, (2) the code should be verified as having been compiled by a proper security-typed language compiler, (3) the declassification filters required by the application, if any, must be acceptable for the operating system and (4) from a global view, there is no suspicious behavior in running this trusted process that would appear to be covert channels (such as forking dozens of processes which might each leak a small amount of information by their existence and through declassifiers).

The key requirement here is the first, compliance testing, which is discussed in more detail below. The other three requirements are more general or more ad hoc—there are no general solutions so lots of ad hoc ones are possible. They remain ongoing areas of research. We discuss some preliminary approaches in Section 4.2.

Note that because the service itself is trusted to handle multiple security levels of data, it should either be written in a security-typed language and bootstrapped into place, or it should be small enough to be verified by hand.

3.5 Compliance analysis

A trusted application is given some flexibility to handle information in ways that a normal application could not. Before granting such privileges, however, the operating system should check to be sure that the application will not abuse them. In other words, the application policy should *comply* with the operating system policy.

> An application is said to be *compliant* if it introduces no information flows that violate the policy of the operating system in which it is running.

For a security-typed application *all* possible information flows can be determined based solely on the high-level delegation policy, modulo some declassifications [13]. The operating system need only check the compliance of flows that are relevant to its own principals.

The compliance analysis between a security-typed application with high-level policy and a MAC-based operating system with static policy consists of three steps. (1) Convert the application policy and operating system policy into a form in which they can be compared. (2) Determine which security levels are shared between the operating system and application. For each of the security levels, collect all allowed flows for the application. (3) Compare these to the flows allowed by the operating system. If there are strictly more flows allowed by the operating system with respect to shared security levels, then the application can be declared compliant and can be safely executed.

This problem contains several challenges. One is that the OS may contain security levels not used in the application and the application may contain security levels not used in the OS. Another is that the OS and application may have a mismatch in the granularity of permissions. Also, either policy could be quite large and unwieldly, making analysis slow or even intractable. These were all problems we had to solve when implementing this analysis for Jif and SELinux. We describe our implementation in Section 4.3.

4 Implementation

For our implementation, we use the Security Enhanced Linux (SELinux) operating system [25] provided as part of the mainline Linux kernel. To build our secure applications, we used the most mature security-typed language, the Java + Information Flow (Jif) language [23], augmented with our policy system (Jif/Pol). Jif is the only security-typed language with an infrastructure that was robust enough to be expanded to handle the kinds of system calls that were necessary for interacting with SELinux. Because we were focused primarily on confidentiality, it was sufficient for us to use Jif v. 2.0.1 (v. 3.0 adds integrity to the security labels in Jif and is a target for our future work).

Our implementation consists of three major endeavors. First, we extended the Runtime infrastructure of the Jif compiler with an interface to SELinux kernel 2.6.16 for getting and setting SELinux security contexts on network sockets and files. In order to make this configurable we added some primitives to the Jif/Pol language and implemented the changes in the Jif/Pol policy compiler. Second, we constructed the Service for Inspecting and Executing Security-Typed Applications (SIESTA). This includes a system daemon along with an interface that can be run by the user; both were written in C. It also includes a policy compliance checker which was written in XSB Prolog. Thirdly, we have utilized this infrastructure to build and test two demonstrative applications: `logrotate` and JPmail.

4.1 Extensions to the Jif Runtime

The basic paradigm in Jif for labeling operating system resources is to parameterize the resource stream with a label and pass that label into the proper method of the `Runtime` class when opening or creating the resource. The `Runtime` method then checks to ensure that the label passed in by the application is acceptable (not too high, not too low) for the resource being requested. For example, the following code gets the standard output stream and attempts to leak a secret.

```
// user is a principal passed in through main(...)
Runtime[user] rt = Runtime[user].getRuntime();
final label{} lb = new label{user:};
PrintStream[lb] stdout = rt.stdout(lb);
int{high:} secret = ...;
stdout.println(secret);
```

Jif ensures (1) that the `Runtime` class, which is instantiated by `getRuntime()`, is parameterized only by the user who executed this program (for SELinux this would be the security context of the program). Jif also ensures that when creating a stream for `stdout`, that the stream is parameterized by a label which is (2) equivalent to the label passed as a parameter to `rt.stdout()` and (3) no more secure than the label on the `Runtime` class. This corresponds to the notion that we should be able to print public data or user data to the user terminal, but nothing more secret than this.

Following this paradigm, we extended the Jif `Runtime` to handle labeled IPsec sockets (both client and server sockets) and labeled files. The basic concept is straight-forward: we provided `openSocket`, `openServerSocket`, `openFileWrite` and `openFileRead` methods which ensure that the streams attached to these operating system resources are properly labeled within the Jif application. The details of this implementation required some additional work, however, in order to properly interface with the SELinux API for getting and setting security contexts on sockets and files.

The code for implementing the socket interface was particularly challenging, because of the way labeled IPsec handles SELinux security contexts and IPsec security associations (SAs). Namely, in order to provide the proper cryptographic protocols for a particular socket, it is necessary that the application first creates the socket, then assigns the proper security context and then attempts to make a connection (it must occur strictly in that order). At that point, the IPsec subsystem attempts to establish an SA for the given security context, local host and port number and remote host and port number.

The problem is that the standard Java Socket API (on which we must build for Jif) does not distinguish between creating a socket file descriptor and attempting to make a connection. Consequently, we had to extend the Socket class to implement our own SelinuxSocket and SelinuxServerSocket. The constructors for our new classes can take a security context. Then, when the socket attempts to establish a connection, a shim is inserted between socket creation and socket connection that calls into the SELinux API to change the socket's security context. After connection, the Runtime class ensures that the SA retrieved for the socket corresponds to the security context that it was set to.

The rest of the code in Runtime.openSocket(...) follows the model of stdout(...), checking to ensure that the label which is attached to the Jif Socket object corresponds to the security context attached to the operating system resource. The difference is that because sockets are always two-way in Java, the label must be equivalent (neither higher security nor lower security) to the SELinux security context.

Extending the Jif policy system Because Jif and SELinux use different kinds of labels and principals, we needed to make some connection between them. We handled this by extending the Jif/Pol policy language with an operator, [·], to signify an operating system label and also wild cards to match a series of labels.

Consider the lattice for logrotate in Figure 3. The SELinux principal logrotate_var_lib_t:s0[3] is at a lower secrecy level than the Jif principal configP. With our new policy syntax, we can express this relationship as [.*:.*:logrotate_var_lib_t:s0] -> configP. The Jif Runtime does not create principals in advance to correspond with every operating system principal; it only creates them as needed (e.g. when a file labeled with that principal is opened). Furthermore, when they are created they are assumed to be unrelated (incomparable) to any other principals in the lattice. Thus, the effect of the policy statement given above is that a hook is inserted into the Runtime to watch for any principals matching this wildcard and when one is created, it will be properly placed in the lattice.

When the relationship is reversed and a Jif principal must be lower in the security lattice than an SELinux principal, the relationship is stored in the Jif principal at the time of its creation (the beginning of the program). For example, if we have the policy statement logP -> [.*:.*:.*:.*], placing the Jif principal logP lower in the lattice than any SELinux principal. Note that this policy statement does not presume anything about the relationship between different SELinux principals that match the same wildcard—only that each of them is higher than logP.

To summarize, the lattice policy we used for logrotate is small and concise (it is the lattice that was illustrated in Figure 3):

```
[.*:.*:logrotate_var_lib_t:s0] -> configP;
configP -> logP;
logP -> [.*:.*:.*:.*];
```

4.2 SIESTA

The Service for Inspecting and Executing Security-Typed Applications (SIESTA) consists of two parts: a service interface and a system daemon, as shown in Figure 5. The service interface takes two inputs from the user—the security-typed application to be executed and the desired MLS range at which it should be executed. The service interface calls a long-running system daemon to carry out the checks described in Section 3.4. If everything is acceptable, SIESTA proceeds to execute the Jif application with the special MLS privileges. In the following, we describe SIESTA's operation in more detail.

The SIESTA service interface starts running with the same MLS level as the process that called it. Running the interface also causes an SELinux transition into the siesta_t domain which limits the process's functionality to communicating with the daemon and forking a new trusted Jif application. The communication between the interface and the daemon is supported by IPC mechanisms plus security functions, creating what we called a SIESTA channel. Furthermore, supported by OS policy we make the siesta_t domain the sole gateway for executing Jif applications in a domain with special MLS privileges. This guarantees that the user cannot directly run a Jif application with special privileges unless it has first been checked by SIESTA. The logic in the service interface is quite simple, deferring the complex considerations to the system daemon.

First, the SIESTA system daemon is responsible for ensuring that the Jif application it has been given is trustworthy. It must first ensure the "Jif-ness" of the application bytecode archive (jar). Doing this in a general way is really an orthogonal issue and a research topic in itself, so here we take a straight-forward approach and just validate a jar signature against a potential list of third-party, trusted Jif compiler signatures.

Once the Jif-ness of the application has been established, the policy jar that has been passed to the daemon must be checked for compliance against the system policy. The jar contains the Jif-compiled policy files (principals and policy store) that will govern the application while it executes. It also contains a manifest of the policy from which it was built. The policy compliance check is described in more detail in Section 4.3.

Thirdly, the SIESTA daemon ensures that the declassifiers used by the application are acceptable to the operating system. Although there are no general solutions to

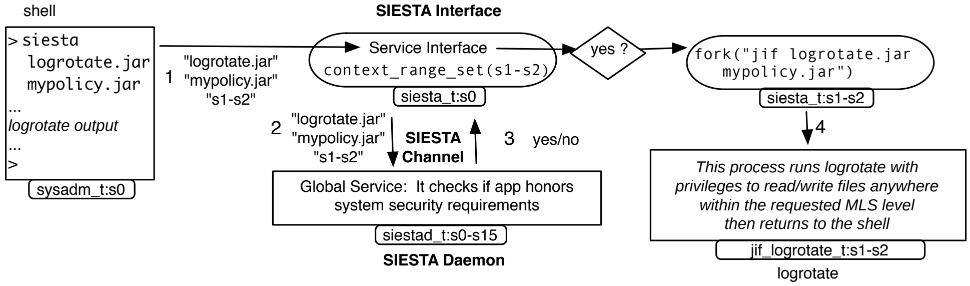


Figure 5: SIESTA: a service which securely validates and executes trustworthy security-typed applications.

this problem, we provide a preliminary approach. Before running an application, a compliance algorithm should check the declassifiers used by the application against a white-list or black-list of declassifiers. For example, standard military procedure prohibits the use of DES for protecting secret data. This can be easily checked in the application policy and applications that violate such requirements can be prevented from executing. As more security-typed applications are used, a list of trusted declassifiers can be established and become a more natural part of the operating system policy. Also, some applications, like `logrotate`, don't need any declassifiers at all. Other approaches could also be feasible here, taking advantage of ongoing research in quantifying the information leaked through declassification [28]. We discuss this further in Section 5.1.

Lastly, although we do not attempt to implement any particular policy for eliminating the covert channels which could be created through the execution of hundreds of these security-typed applications, we provide hooks that could be used as such policies are developed.

The SIESTA daemon must be executed by a system administrator prior to executing any security-typed applications. It must run with a full range of MLS privileges so that it can handle security-typed applications of all sensitivities. At the same time, it can be limited to a fairly constrained functionality, because it only needa to read from files and communicate with the SIESTA service interface via IPC.

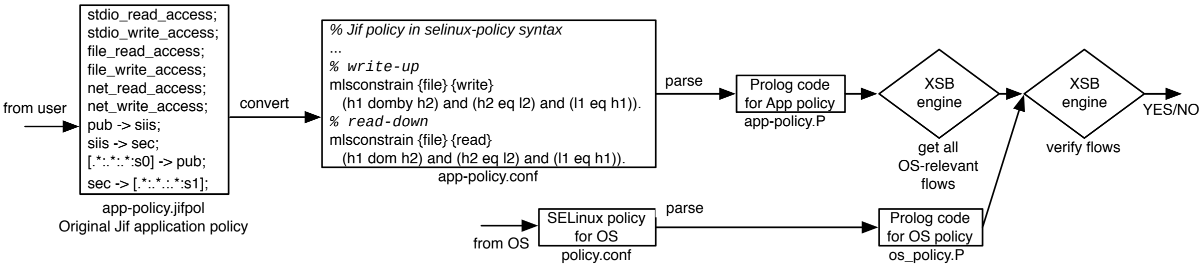
4.3 Compliance analysis

There are a few key challenges in attempting to determine compliance between Jif policy and SELinux policy. The foremost challenge is in the semantic difference between Jif's information flow lattice and SELinux's MLS constraints. Although SELinux claims to have an MLS policy (which normally means a "no read-up", no write-down" lattice-based policy), the so-called "MLS" extension is really a richer policy language which can be used to implement MLS, but can also implement more general policy goals. The second challenge lies in the difference in granularity between Jif policy and SELinux policy. While SELinux policy distinguishes various operations and resource types (the policy for setting the attribute on a file could be different from writing to a socket, for example), Jif policy gives a more comprehensive view of all information flows in an application. The third challenge lies in the size of the SELinux policy for a whole operating system. The standard, complete operating system policy consists of well over 20,000 policy statements.

For the first challenge, analyzing policy compliance would have been a straight-forward lattice comparison if not for the generality of the SELinux MLS policy. Thus, some policy analysis tools are needed to determine what information flow goals are implemented by the operating system and whether they are compatible with the information flows in the application we are seeking to execute. Although several SELinux policy analyzers exist, none were suitable for our purposes, because none handles the new SELinux MLS extensions which were our primary concern. Consequently, we developed our own policy analysis tools for SELinux MLS policy, inspired by the policy analysis engine, PAL [30]. Our tool determines what information flows are allowed between MLS levels. We describe this analysis in more detail elsewhere, including a formal consideration of correctness [14]. For this work, we extended and utilized this tool to compare the flows allowed in a Jif application to the flows allowed in the host operating system.

The second challenge is that in order to compare the operating system policy and the application policy, they must be in a comparable form. Since SELinux policy is more general than Jif policy, we translate our Jif policy into an SELinux policy. This also allows us to reuse our policy analyzer for both policies.

For example, consider the Jif policy in Figure 6 in the box labeled `app-policy.jifpol`. This policy says that the Jif program has access to operating system files and network sockets. Also, it allows data to flow from `pub` to `siis` to `sec`. Furthermore, the policy states that `pub` is equivalent to the security level `s0` and `sec` is equivalent to the security level `s1`, while `siis` has no corresponding



```
stdio_read_access;
stdio_write_access;
file_read_access;
file_write_access;
net_read_access;
net_write_access;
pub -> siis;
siis -> sec;
[.*:.*:.*:s0] -> pub;
sec -> [.*:.*.:.*:s1];
```

```
% Jif policy in selinux-policy syntax
...
% write-up
mlsconstrain {file} {write}
  (h1 domby h2) and (h2 eq l2) and (l1 eq h1)).
% read-down
mlsconstrain {file} {read}
  (h1 dom h2) and (h2 eq l2) and (l1 eq h1)).
```

Figure 6: The compliance testing process.

identity in the operating system.

Given this policy, the contents of an operating system file at level s0 could be read into the application at level sec (through a read-down) and then written out to a file at level s1. This constitutes a flow from s0 to s1. This flow must then be checked against the operating system's policy to determine if it is an allowed flow. Note that although siis has no corresponding principal in the operating system, it cannot simply be ignored, because it could be involved (as in this case) in a flow between two operating system principals. At the same time, the flow from pub to siis need not be checked against the operating system policy, because the siis principal does not correspond to an OS principal. Only when both endpoints of a flow have corresponding OS principals does the flow need to be checked against the OS policy.

Next, if the Jif application asks for file_read_access and file_write_access, we then add, respectively, read-down rules and write-up rules for file access, giving an SELinux-style policy as shown in the box, app-policy.conf. We add similar rules for user I/O if stdio_read_access and/or stdio_write_access are set and for sockets if net_read_access and/or net_write_access are set in the Jif policy.[4]

The third challenge we faced was the magnitude of the operating system policy, which threatened to make the analysis intractable. We are able to manage this in several ways. Firstly, once the policy has been compiled into Prolog, it need not be compiled again. Furthermore, XSB Prolog has some efficient methods of storing the policy, using tabling, which improve performance for analysis. Most importantly, however, we are able to radically reduce the analysis of the operating system policy by first analyzing the application policy. Because we are only interested in verifying that the flows allowed by the application are also acceptable to the operating system, we don't need to check all operating system flows—just the ones that intersect with the application.

4.4 Sample applications

We have implemented two sample applications in order to demonstrate the range of functionality provided by our architecture. The first is logrotate which demonstrates proper labeling of files and tracking of information flows from multi-leveled files handled within the same application. This is an example of a secure implementation of an operating system tool and demonstrates features that would be common to many of the utilities listed in Figure 1. The second application is larger and more complex—a modification of the JPmail email client [12]. For this work, we migrate this client from using a PKI for achieving end-to-end confidentiality to using the SIESTA infrastructure with labeled IPsec.

4.4.1 logrotate

The logrotate utility is regularly executed by cron to gradually phase out old log files. The utility rotates each set of log files based on some configuration. For example, the messages log is renamed to messages.1, messages.1 to messages.2, etc. The configuration specifies which logs to rotate and each log has a rotate attribute indicating how many back logs to save. The full version of this utility can also run scripts, compress logs and send emails. We did not implement these additional features, but chose to focus on the essential functionality of log rotation.

The logrotate program handles a variety of sensitive information flows (an example lattice is shown in Figure 3 and the lattice policy is given in Section 4.1). It handles two files which are (typically) publicly readable: a configuration file and a state file. It handles various other log files at various security levels, creating and modifying the files as needed in order to rotate and delete logs according to their particular configurations. The data in the log files is more or less secret depending on the nature of the log. For example, the logs for X Windows and wtmp[5] are usually publicly readable. Other logs such as secure or maillog are more secret due to their contents. On the other hand, the attributes of the log files (e.g. seeing that they exist, getting their names, reading their last date of modification, etc.) are public.

In order to rotate logs, the program needs to read configuration information and state information and based on that, the logs themselves are renamed. This effectively passes information from the configuration files into the log files (it is clear from looking at the directory, for example, what the `rotate` attribute is for each log—usually it is the highest filename extension, like the `4` in `messages.4`). Thus, in order for our application to function properly, the level of the configuration data must be lower or equal in the lattice, i.e. less secret, compared to the level of the log data. On the other hand, we do not want to leak log data into the configuration file (since it is often publicly readable) or into other log files. In fact, our Jif application verifies that this policy can be upheld: not even small bits of information released by control flows are leaked from the log files into the configuration files and not even a single declassifier is needed to implement this system utility.

4.4.2 Email client

The JPmail application [12] is an email client built in Jif 2.0.1, using our Jif/Pol policy framework [13], which enforces information flow control on emails according to a given Jif policy. When we built JPmail, it was the first real-world application built in Jif. Previously, in order to maintain information flow control, JPmail utilized encrypting declassifiers to send out email on public networks. By utilizing labeled IPsec sockets and trusting the operating system to handle distributed security (i.e., the MAC OS security ensures that emails are not leaked from intermediate or destination servers), we were able to remove the cryptographic infrastructure from JPmail and significantly simplify the code. Furthermore, we were able to extend our mail client to handle communication with mail servers at multiple security levels within a single process.

While this application serves to demonstrate the usage of client sockets, the real significance of this application is mainly in its size and complexity. It is the largest existing Jif application and so it gives us some insight on the difficulty of augmenting a realistic application to work with SIESTA. In this vein, we were gratified to discover how much cleaner and simpler the code became when it could trust the operating system to handle security concerns over its resources.

Also significant about this application is its use of declassifiers. This is due to the fact that it gets user input for all levels of email accounts from the same terminal window. The application logic handles the proper downgrading of input when responding to a public as opposed to a secret email. Another, minimal source of leakage is through an implicit flow caused by handling both public and secret email accounts in the same user interface loop. This small flow that normally occurs when a single user interface is used for multi-level inputs is handled with a declassifier. The declassifier and its use in the code must be determined to be safe for the email client and then it is included among the white-list of declassifiers in the operating system policy.

5 Discussion and evaluation

5.1 Declassification

Strict information-flow policies are too strict for some applications. This necessitates slight relaxations of the policy through controlled “escape hatches”. There has been a great deal of consideration about declassification in the language-based security community [28]. We have added our own modest work, called *trusted declassification*, to this collection [13] with the greatest advantage being its practicality and the way it exposes declassifiers through a high-level policy. This exposure of declassifiers is key for our compliance analysis. The policies it allows are similar to recent work on integrity policies for generating Clark-Wilson Lite models [31] of security.

The key is that all declassifying filters (aka declassifiers) must be declared in the high-level policy, indicating what information flows they can be used for. For example, in order for an application to expose secret data (labeled `{sec:}`) to the public (labeled `{pub:}`) after encrypting it with AES, the application’s policy must contain the statement, `sec allows crypto.AES(pub);`. Otherwise, when the application tries to use the declassifier at runtime, the policy check fails and an exception is thrown. Thus, an application’s code may contain potentially many declassifiers, but only those which are explicitly allowed in the policy can be used at runtime (the compiler ensures that all necessary runtime checks are present in the application before it generates the object code).

5.2 Performance

In this section we consider performance costs associated with the approach outlined in the preceding sections. We stress the preliminary nature of the implementation, experiments, and test-bed. Because of the unique and cross-cutting nature of these experiments, it is highly difficult to isolate performance cost (simultaneously at the application, OS and network layers). Experimental error is caused by interactions between the OS (process scheduling, interrupts), network delays, Java (garbage collection and dynamic class loading), and other system services and applications (process interference). Hence, we focused our initial experimentation toward obtaining a broad performance characterization of the design, leaving more precise evaluation and the invention of apparatus to achieve it to future work.

We study the overheads associated with information flow controls at the application (Jif) level. We compile the Jif programs using Ahead-of-Time (AOT) compilation with the `gcj` compiler v. 4.1.1 with `classpath` 0.92 [29]. We examine two operations, *a*) *logrotate* which renames up to four log files per set and as many

Operation	Configuration	Median	Mean	σ
logrotate	C	7.923501	7.943820	0.133496
logrotate	Jif	13.949643	13.925600	0.122477
send	C	17.825400	21.834692	12.163714
send	Jif	12.522900	15.620364	10.705158
SIESTA	compliance	241.060289	252.830086	25.025038
SIESTA	cached	31.639957	32.368633	3.353408

Table 1: Time (ms) to send a 10KB email in both Jif and C, time (ms) to perform one rotatation of 50MB of log files and time (ms) to start up the Jif process using SIESTA (includes Jif-ness validation and compliance checking).

sets as requested, and *b*) *send* which sends a single email from the client to the server. For `logrotate`, we compare our Jif application with the latest C version 2.7.1, using only minimal features of the applications. For sending mail, we compare a custom C-based MTA application with JPmail with IPsec enabled. A 3DES ESP with MD5-integrity policy was used in all IPsec-enabled tests. The tests were run between two identical 3GHz Intel hosts running SELinux Kernel version 2.6.16 with 1GB memory on a lightly loaded 100MBit network. All experiments were repeated 100 times.

Table 1 provides macrobenchmarks for the different operations and configurations when sending a single 10KB email and when rotating forty log files totalling 50MB. For sending email, the overheads of the system/approach are relatively small: in all cases the average execution time is less than 25 milliseconds, and in many cases significantly less. In general, costs are in line with unprotected systems, which indicates operations such as these are likely to be unaffected by the additional security infrastructure.

5.2.1 Sample applications

In the case of `logrotate`, the Jif application consistently runs 2x slower than the C version. We tested the two programs with various log files of different total sizes. Since there is no inspection of the contents of the files, the size is the determining factor. The displayed result is for a standard complement of log files totalling 50MB. For this utility, the decrease in speed is inconsequential. The `logrotate` application is generally a cron job that only runs daily or even weekly. Additionally, our Jif code is entirely unoptimized and could be improved.

A further comparison of the Jif and C applications shows, interestingly, that Jif is faster, on average, for email sends, although there is a significant variance for both C and Jif. We found the Jif functions represented a vanishingly small amount of overhead in this case. Further investigation revealed that a significant portion of the additional costs observed in the Jif application are due to delays in the use of Java network APIs. The C program is slower for sends because of an implementation artifact: JPmail does a less graceful exit with the server, whereas the C program waits for the server finalization.

5.2.2 Compliance testing

For SIESTA, the overheads are constant and small. The policy compliance check requires a call into the XSB prolog interpreter but executes relatively efficiently, requiring only 15.512256 ms on average. 5.577328 ms of this time is spent in loading the policy (both for the application and the OS) while 8.902952 ms is spent doing the flow checks. XSB prolog is highly optimized and the prolog source files can be compiled for greater efficiency. The majority of time is spent in signature validation for the jar file that is being loaded.

Fortunately, this validation process (checking Jif-ness and checking compliance) is a one-time cost when first verifying the compliance of the application and its policy. This process only needs to be repeated if the application changes (for a new version), if the application policy changes or if the OS system policy changes, all of which should be rare events. Otherwise, the service may compile the jar file together with its policy into a binary executable and the hash of the binary can be cached for future executions. Checking the hash is almost an order of magnitude faster than validating the jar signature and checking compliance (32.368633 ms vs. 252.830086 ms averaged over 100 runs).

5.3 Practicality/usability

What we have implemented is a prototype using Jif as a basis for trust when constructing trusted applications for a secure operating system. The guarantees that we are capitalizing on are not specific to Jif, but are part of the static type analysis that Jif implements for security types. Jif has some problems. For example, Jif is built on Java and requires that the JVM be loaded. This may not be desirable for some applications. Furthermore, the JVM introduces a large amount of code into the trusted computing base. Fortunately, this is not an inherent limitation to our approach because the security-type analysis we depend on is orthogonal to the JVM (and any security goals it may implement).

Another limitation is that Jif is not easy to program in. Some of this is inherent in the fact that it must be thorough about checking all information flows. For example, it requires all exceptions to be handled, including `NullPointerExceptions`, and it tracks implicit, control flows, which can be difficult for a programmer to follow. On the other hand, we believe that much of the burden can be alleviated through some semi-automated labeling and through the development of other tools and programming patterns [1, 12]. We are currently investigating these avenues. In the meantime, it is also important that the development of these tools be guided by practical experience and some knowledge of how they can be deployed. That is what we have described in this paper.

6 Related work

This research considers the intersection of two areas of related work: 1) secure systems development, particularly mandatory access controls and 2) language-based security, particularly security-typed languages.

Mandatory Access Controls The foundation of our OS work comes from the Flask architecture [34], which has been integrated into Linux through the Linux Security Module (LSM), giving Security Enhanced (SE) Linux [32]. This is now being shipped as part of the mainstream kernel in the 2.6 series and turned on by default in Redhat distributions since Fedora Core 5. Other work in operating systems with MAC security include Trusted Solaris [21], Solaris Trusted Extensions [20], TrustedBSD [9] and SEDarwin [35].

MAC Operating Systems require all subjects and objects are labeled and all security-sensitive operations are hooked with runtime checks. These checks query a previously configured security policy to determine whether the operation is allowed, based on the subject and object labels.

These policies have been used to implement various high-level information flow properties across a distributed system. Recent research has shown that this basic mediation (called Type Enforcement or TE) can be used to enforce integrity constraints on data [15]. More recently added multi-level security (MLS) labels can be used to enforce confidentiality [11]. A major limitation in all this work is that it only observes security-sensitive operations from outside of applications; it cannot peer into application code to catch data leaks.

Security-typed languages Security-typed languages have their heritage in the information flow policies of Bell & LaPadula [3] with extensions to lattices as described in [7]. This led to the concept of *noninterference* [10], in which modifications of high security data cannot be observed in any way through low security data. Thus, two execution traces with different high security inputs yield the same low security outputs. In their seminal work, Volpano, Smith and Irvine [36], showed how these information flow policies could be encoded into types and noninterference could be determined through a type analysis. The first language to implement this was a variant of Java, called Java + Information Flow (Jif) [22], which uses labels based on the Decentralized Label Model (DLM) [24].

Jif remains the most mature security-typed language, although there is much activity in the field (see a recent survey for more details [27]). Other security-typed language projects include functional, [26], assembly [5] and web scripting [17] languages, as well as language features for multiple threads of execution [33] and distributed systems [18]. Other recent work has studied *integrity* information flow [4, 6] in the context of replication and partitioning [37]. Much of the work in security-typed languages has been theoretical, but some recent work demonstrated that these languages can also produce real-world applications [12].

A recent project, called GIFT [16], implements a more general, but less rigorous approach to tracking data flows within C applications. This language framework may be another useful target for our architecture.

7 Conclusion

In this paper, we have described an important problem in secure systems development, namely the inability of an OS-level reference monitor to look inside a multi-level application. We have provided an architecture to solve this problem by using security-typed languages to track secure data flows within applications. We implemented this architecture for the security-typed language, Jif, and the MAC operating system, SELinux. Through the applications, `logrotate` and JPmail, we showed that our approach is secure, flexible and efficient.

Acknowledgements

A special thanks to Steve Chong for his tireless help in providing insight about the inner workings of Jif and to Mike Hicks for his numerous editorial comments. This work was supported in part by NSF grant CCF-0524132, "Flexible, Decentralized Information-flow Control for Dynamic Environments" and NSF grant CNS-0627551, "CT-IS: Shamon: Systems Approaches for Constructing Distributed Trust".

Notes

[1] To be complete, in addition to these five sensitivity levels there are also category sets, but we leave them out here for simplicity of presentation, as they do not add to the technical complexity.

[2] In SELinux, "MLS" has a broad meaning with the names and semantics being drawn from a general policy. We consider the implications of this more in Section 4.3. The meanings of all type and MLS level names are also defined in the policy, but typically s0 is most public and s15 is most secret.

[3] Recall that logrotate_var_lib_t:s0 is an abbreviation for system_u:object_r:logrotate_var_lib_t:s0.

[4] For brevity and clarity of presentation, we have given a truncated version of the policy, but to be complete, our implementation includes

all the write-like and read-like operations necessary. For the same reason, although our implementation also handles category sets, we forego a discussion of that here.

[5] wtmp is queried by the UNIX command last.

References

[1] A. Askarov and A. Sabelfeld. Secure Implementation of Cryptographic Protocols: A Case Study of Mutual Distrust. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS '05)*, Milan, Italy, September 2005.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2003.

[3] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, March 1976.

[4] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[5] E. Bonelli, A. Compagnoni, and R. Medel. Non-interference for a typed assembly language. In *Proceedings of the LICS'05 Affiliated Workshop on Foundations of Computer Security (FCS)*. IEEE Computer Society Press, 2005.

[6] S. Chong and A. Myers. Decentralized robustness. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*, Venice, Italy, July 2006.

[7] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.

[8] S.W. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. VMWare, Inc., October 1998. US Patent No. 6397242.

[9] FreeBSD Foundation. SEBSD: Port of SELinux FLASK and type enforcement to TrustedBSD. `http://www.trustedbsd.org/sebsd.html`.

[10] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[11] Chad Hanson. Selinux and mls: Putting the pieces together. In *Proceedings of the 2nd Annual SELinux Symposium*, 2006.

[12] B. Hicks, K. Ahmadizadeh, and P. McDaniel. From Languages to Systems: Understanding Practical Application Development in Security-typed Languages. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC 2006)*, Miami, FL, December 11-15 2006.

[13] B. Hicks, D. King, P. McDaniel, and M. Hicks. 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.

[14] Boniface Hicks, Sandra Rueda, Luke St. Clair, Trent Jaeger, and Patrick McDaniel. 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.

[15] T. Jaeger, A. Edwards, and X. Zhang. Policy management using access control spaces. *ACM Trans. Inf. Syst. Secur.*, 6(3):327–364, 2003.

[16] L. Lam and T. Chiuch. A general dynamic information flow tracking framework for security applications. In *Applied Computer Security Associates ACSAC*, 2006.

[17] P. Li and S. Zdancewic. Practical Information-flow Control in Web-based Information Systems. In *Proceedings of 18th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2005.

[18] H. Mantel and A. Sabelfeld. A Unifying Approach to the Security of Distributed and Multi-Threaded Programs. *Journal of Computer Security*, 2002.

[19] R. Meushaw and D. Simard. Nettop - commercial technology in high assurance applications, 2000. `http://www.vmware.com/pdf/TechTrendNotes.pdf`.

[20] Sun Microsystems. Solaris trusted extensions. `http://www.sun.com`.

[21] Sun Microsystems. Trusted solaris operating environment - a technical overview. `http://www.sun.com`.

[22] A. C. Myers. Mostly-Static Decentralized Information Flow Control. Technical Report MIT/LCS/TR-783, Massachussetts Institute of Technology, January 1999. Ph.D. thesis.

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

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

[25] Security-enhanced Linux. `http://www.nsa.gov/selinux`.

[26] F. Pottier and V. Simonet. Information Flow Inference for ML. In *Proceedings ACM Symposium on Principles of Programming Languages*, pages 319–330, January 2002.

[27] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[28] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the IEEE Computer Security Foundations Workshop*, Aix-en-Provence, France, June 2005.

[29] G. Sally. Embedded Java with GCJ. *Linux Journal*, (145), May 2006.

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

[31] U. Shankar, T. Jaeger, and R. Sailer. 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, February 2006.

[32] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a linux security module. Technical Report 01-043, NAI Labs, 2001.

[33] G. Smith and D. Volpano. Secure Information Flow in a Multi-Threaded Imperative Language. In *Proceedings ACM Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, January 1998.

[34] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lapreau. The Flask architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123–139, August 1999.

[35] Christopher Vance, Todd Miller, and Rob Dekelbaum. Security-Enhanced Darwin: Porting SELinux to Mac OS X. In *Proceedings of the Third Annual Security Enhanced Linux Symposium*, Baltimore, MD, USA, March 2007.

[36] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. 4(3):167–187, 1996.

[37] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using Replication and Partitioning to Build Secure Distributed Systems. In *Proceedings of the IEEE Symposium on Security and Privacy, 2003*, pages 236–250, 2003.