Integrating SELinux with Security-typed Languages

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,tjaeger,mcdaniel*}*@cse.psu.edu*

Abstract

Traditionally, operating systems have enforced MAC and information flow policies with minimal dependence on application programs. However, there are many cases where systems depend on user-level programs to enforce information flows. Previous approaches to handling this problem, such as privilege-separation of application components or assuming trust in application information flow enforcement, are prone to error and cumbersome to manage. On the other hand, recent advances in the area of security-typed languages have enabled the development of realistic applications with formally and automatically verified information flow controls. In this paper, we examine what it takes to integrate information flow enforcement of applications written in a security-typed extension of Java (called Jif) with SELinux. To this end, we have extended the Jif infrastructure to support interaction with SELinux security contexts, and we describe the SELinux policy and system calls which are necessary for a successful integration. We have also identified the need for further services, such as a means of formally verifying compliance between information flow policies. We have demonstrated the utility, flexibility and security of our approach by constructing a prototype multi-level secure email client.

Keywords

Information Flow Policies, Java Information Flow (Jif), security-typed languages, SELinux

1 Introduction

Mandatory access control (MAC) operating systems like SELinux often require the assistance of user-level programs. For example, the X server is a user-level service that processes information for all programs that use the graphical user interface [21]. Since multiple users of multiple security clearances may use the same X server, it is necessary to enable the X server to enforce the system's MAC policy on the resulting information flows. The X server has been extended with a reference monitor interface to enforce such MAC policies [21], and a user-level policy server architecture has been developed to provide MAC enforcement to any application with an appropriate reference monitor [28]. One problem with this approach is that the completeness of the reference monitor hooks (i.e. whether all security-sensitive operations are checked) depends on the programmer [8, 9] and is not an easy property to check. Past experience shows that this is error-prone.

An alternative approach to enforcing MAC policies in user-level programs is to use security-typed languages, such as the Java + Information Flow (Jif) language [23, 25], Flow Caml [27], and a web scripting language [22]. Jif is the most mature among these. Jif extends the Java language to enable security labels to be associated with program variables by extending their data types. Jif enforces a traditional lattice policy [7] where the data in a variable of label x may not be leaked to a variable of label y unless y is equal to or dominates x in the lattice policy. Importantly, the Jif compiler will only compile programs that meet this requirement, so an executing Jif program is *guaranteed* to enforce its information flow policy. This may involve some hooks for making runtime checks of the policy, but Jif ensures the completeness of hook placement. While these guarantees depend on trust in the Jif compiler and the JVM implementation, it is debatable whether these assumptions are more significant than those in the X server or Linux kernel itself. In this paper, we investigate the integration of Jif programs into an SELinux system.

A question is how important is it for SELinux to recognize MAC enforcement in Jif programs. First, although there are only a small number of Jif programs that have been written to date, real applications are starting to emerge [11] which is also driving the construction of development infrastructure for Jif [1, 11]. We expect that once a critical mass of infrastructure and knowledge is developed we will see many applications built in such languages.

Second, and more importantly, the problem of application-level information flow control is more prevalent than has been acknowledged. The community is well-aware that some servers (e.g., the X server, as mentioned above, as well as `sshd`) may be entrusted to enforce separation

between the information flows of different clients. The current MLS policy already recognizes the need to allow many system utilities and policy management tools to handle data with multiple levels of sensitivity (including `passwd`, `logrotate`, `klogd`, `auditd`, `tmpreaper`, `iptables` and twenty-eight others). Further, we identify that client applications, such as email clients and web browsers, must protect client data that may be of multiple secrecy levels. For example, phishing attacks involve leaking secret data to unauthorized principals. Finally, we have found that most applications need to protect their own integrity (e.g., from network data) [26], and security-typed languages, such as Jif offer a means to do this in a principled way. For these reasons, we find a significant motivation in having applications built in security-typed languages and integrating system and application information flow enforcement.

In this paper, we examine how to enable SELinux to leverage the MAC enforcement abilities of programs written in security-typed languages, in particular the Jif language. Because these programs enforce their own lattice policies, there is no need for the addition of a reference monitor interface or to use the SELinux policy server. On the other hand, we must be able to configure the Jif application and SELinux systems to work together. The Jif application must be able to extract the necessary information from the SELinux system to be able to enforce its information flow policy. We describe how SELinux mechanisms and policy are used to enable a Jif email client application to use SELinux to enforce system policies. Also, the SELinux system must be able to convince itself that a Jif program is enforcing the system's MAC requirements. We outline the mechanisms that are required to achieve this goal. This paper is a shorter version of another technical report [14], but this paper focuses on the specific SELinux impact of integrating security-typed applications with MAC systems.

The rest of the paper is structured as follows. In Section 2, we motivate the need for comprehensive information flow enforcement that includes applications and operating systems. In Section 3, we examine the challenges in leveraging security-typed languages for system information flow enforcement. In Section 4, we outline the approach to building security-typed applications that enable comprehensive information flow control by examining an email client, *JPmail*. In Section 5, we detail how to implement *JPmail* on an SELinux system, such that the system information flow requirements can be enforced. In Section 6, we discuss taking the proof-of-concept to a more complete approach. In Section 7, we conclude and discuss future work.

2 Comprehensive Information Flow Enforcement

We propose integrating SELinux controls with information flow-aware applications to enforce system security guarantees on the secrecy and integrity of data. While information flow control is a traditional approach to enforcing security, even comprehensive MAC enforcement with flexible policy models, such as SELinux, are not capable of practical information flow control alone. We claim that they need help from information flow-aware applications in some cases, and we examine what is necessary to integrate security enforcement in SELinux with such applications to achieve the necessary information flow control.

A reasonable question to ask is why the SELinux security goal should be expressed in terms of information flow. SELinux uses an extended Type Enforcement policy [4] and multilevel security (MLS) models that can represent information flow security goals, least privilege, and many others. Regardless of how the policy is expressed in SELinux, we must prove that a system protects the secrecy and integrity of data. Fundamentally, secrecy is about which subjects may be able to access the data in worst case, and integrity is about which subjects can provide data that others may depend upon. In both cases, information flow is a natural representation of these requirements.

For example, we aimed to prove that SELinux policies protect the integrity of the system and certain applications [19, 16]. In order to do this, we built a policy analysis tool, called Gokyo, that identified the SELinux TE policy rules that permitted low integrity data to reach high integrity subjects [20]. The analysis is a Biba information flow integrity analysis [3] where any information flows of low integrity data to high integrity subjects are problematic. A deeper examination determined that nearly all high integrity applications must process some low integrity data (e.g., network requests), but it is possible to limit applications to receive that low integrity data only through interfaces that the applications declare are capable of processing such data according to Clark-Wilson integrity semantics (i.e., discard or upgrade immediately) [6, 26]. Note that although SELinux (with some minor extensions) can restrict which interfaces receive low integrity data, it cannot prove that the application really protects itself correctly. We claim that the problem is similar for information flow secrecy where many client and server applications may be entrusted with data of multiple access classes.

As a result, the premise of this paper is as follows. First, we find it valuable to evaluate whether SELinux policies enforce security goals for both secrecy and integrity using information flow. Second, even to achieve system information flow goals, some applications must be able to prove their ability to protect themselves from low integrity input data and prevent the leakage of high secrecy data. Third, SELinux must be able to support such applications' information flow control (e.g., via system labels and policy) and leverage such application-level information flow control to prove that the system security goals are achieved.

3 Leveraging Security-Typed Languages

The recent emergence of security-typed languages that enable the construction of applications with provable informa-

tion flow guarantees motivates us to integrate application and system information flow control. Security-typed languages extend the type system of the language to include security labels. Typically, these labels provide semantic information used to enforce information flow, either secrecy or integrity, on the variables. The compiler ensures that any legal program in the language enforces the information flow implied by the security-typed labels, so information flow guarantees, such as the $\star$*-property* and *simple security property* [2], can be enforced by the compiler.

By integrating the enforcement guarantees of security-typed languages and operating systems, SELinux can build a comprehensive guarantee of system security. If an application can guarantee to SELinux that it only allows information flows permitted in the SELinux policy, then SELinux can guarantee the secrecy and integrity implied by those flows. Security-typed applications will need support from SELinux to do this. For example, SELinux must provide the labels for input data that the application receives, such that it may enforce security properly. In this section, we explore the characteristics of SELinux and Jif that must be considered to create a comprehensive information flow enforcer.

3.1 Java + Information Flow (Jif)

Security-typed language compilers can guarantee the $\star$-property ("no write down") and simple security property ("no read up") of the Bell-LaPadula model [2] for all data in a given application. By extending all datatype declarations with a label that indicates the security level of the data and then performing a compositional type analysis, a security-typed language compiler is able to catch all illegal explicit and implicit[1] flows in a given program. In this way, all programs which successfully compile are also assured to have the strong security property of *noninterference* [10] between high-secure data and low-secure data.

In this work, we have used the most mature security-typed language, Jif (**J**ava + **i**nformation **f**low) [23, 25], which covers the majority of the Java language and includes many advanced features specific to information flow (which we have found to be vital for developing real-world applications [11]), such as runtime principals, dynamic labels, label polymorphism, declassification[2] and labels drawn from the decentralized label model (DLM) [24]. Essentially, a Jif *label* consists of a *principal* e.g. `{alice:}`, `{bob:}`, or a conjunction of principals (`{alice:;bob:}`) where the principals are drawn from a *principal hierarchy*. The special label `{}` denotes "public" and is always at the bottom of the hierarchy, such that $\{\} \sqsubseteq \ell$ for all labels, ℓ.

Figure 1 gives an example of a Jif program. In this example, the Jif application handles two operating system objects, the stdin `InputStream` and a `Socket`. When getting the objects from the OS, the application must request them at a certain secrecy level (by passing a Jif label as an argument to the `Runtime` class). The code in `Runtime` (which is part of the trusted computing base for the Jif compiler) checks the labels against the operating system resources and throws an exception if they are not sufficiently secret. Otherwise, it returns the object at the requested secrecy level. Here is where the power of Jif becomes evident: once a resource is labeled, the Jif type system will ensure through a static analysis that the label is never violated througout the duration of its lifetime. This means that all leaks, such as the one in Figure 1, and more complicated leaks, will definitely be caught at compile-time.

```
1: Socket{} leak =
       Runtime.openSocket("spy.org",9999,{});
2: InputStream{sec} in = Runtime.stdin({sec});
3: String{sec} passwd = in.readLine();
4: leak.println(passwd); // ERROR! Illegal leakage
```

Figure 1: Example of security-typed program written in Jif pseudo-code, demonstrating a leaky program using operating system resources. Jif catches the leaks at compile-time.

As we have indicated, applications may have to handle illegal information flows, such as the input of low integrity data or the controlled release of secret data. Security-typed languages use *declassification* to relabel data in an exceptional way, contrary to the lattice policy. Jif implements *robust declassification*, allowing a data item's label to be downgraded under certain conditions. We have extended this mechanism to handle *trusted declassification* [12]. This requires that each principal specify the declassifying functions (*declassifiers*) which should be trusted as downgraders for that principal's data in a Jif program. This is specified in a policy which is dynamically checked whenever downgrading is attempted during program execution. Declassifiers allow the strong noninterference property to be violated in controlled ways. For example, encryption is often considered a safe means of information release (because the information released is vanishingly small). Also acceptable may be a declassifier which performs certain checks or audits for the declassification.

3.2 Integrating Security-Typed Applications and SELinux

Integrating security-typed applications into the enforcement of an SELinux policy requires support from both SELinux and the application. Solutions to these problems will guarantee an integrated infrastructure to enforce information flow policies across layers in a single machine and across machines in a network. [13]

- **SELinux labeling**: The application needs SELinux to provide mechanisms to identify the security label of all system channels to the application.
- **Application labeling**: SELinux needs the application to provide mechanisms to specify the security label of all application channels to the system.

[1]The latest technology only considers control flows, not timing or termination flows.

[2]The most recent version of Jif has added integrity constraints to labels [5], but we did not use this feature.

- **Policy Compliance**: Both SELinux and the application require that the information flows in the two policies comply (i.e., no new flows are created between nodes in the lattices).
- **Authorized Declassification**: Both SELinux and the application must authorize the declassifiers used to relabel data.

First, a security-typed application must be able to determine the label of any data that it receives from the system. SELinux provides a means to extract the labels of file and network data, so we verify whether these are sufficient.

Second, a security-typed application must be able to convey the labels of its output data to SELinux. This turns out to be more complex. Suppose an application processes both secret and public data. If the application sends the data to a remote computer that can receive both secret and public data, then it must identify the data to the operating system, such that SELinux can protect the data correctly. For example, SELinux should not send secrets in the clear or send secrets to a remote computer that is not trusted. We investigate the systems support that SELinux should provide for applications.

Third, it is necessary for correct enforcement of system and application information flows, that the two policies are compliant. Compliance in this case means that there is no information flow in one system that is not allowed in the other, and there is no integrity dependence in one system that is not allowed in the other. This must be stated in terms of the SELinux and Jif application policies.

Namely, consider a program with an input flow I from the operating system, labeled with SELinux security context `s1` and Jif principal `p1`. I eventually flows to an output O labeled with the Jif principal `p2`, and the program would like to output O to an OS resource with security context `s2`. Jif will ensure that `p1` $\leq$ `p2` (i.e. `p2` is more secret than `p1`) throughout the application. Before allowing data labeled `p2` to be output to an OS resource labeled `s2`, however, we must be sure that `s1` $\leq$ `s2` in the SELinux policy. Fortunately, these mappings can be vetted before the application is executed, because the Jif policy (with statements such as `p1` $\leq$ `p2`) and the Jif/SELinux mappings (such as `s1 -> p1` and `p2 -> s2`) are made available to a compliance checking service prior to execution. The compliance algorithm and policy mappings are slightly more general than what we just described, but they follow the same idea. Details can be found in a recent technical report [15].

Fourth, any cases that involve declassification require that both parties authorize the declassification. For example, from an application's perspective, all input and output channels are public, thus any secret data must be encrypted to be sent out on the network. However, SELinux is able to use Labeled IPsec [18] to encrypt and label such channels, so the application need not do its own encryption if it authorizes SELinux to do this. Of course, SELinux must also authorize the use of IPsec.

4 Case Study: An MLS Email Client

In this investigation, we have focused on integrating the SELinux MAC security with Jif application-level information flow control for a secure email client. Specifically, we re-worked a secure email client written in Jif, *JPmail*. The client originally presumed that the operating system and network could not be trusted to maintain the confidentiality of emails as they were sent outside the client to remote email servers. In this work, we have investigated how to remove this assumption when the client is running in SELinux and examined what impact this makes on both application and operating system.

4.1 Information Flow Requirements

A challenging problem with multi-level secure email is that a single person often has multiple security clearances. The prevalent approach to this problem is to run multiple email clients in multiple security contexts. This approach is not easy to manage. A good alternative is to run a single email client that is aware of information flow security and can provably prevent leakage of secret emails to public recipients.

To avoid information leakage, multi-level secure email clients must support specific requirements:

1. Secret emails should be encrypted before being sent out over the Internet.
2. For easier usability, all of the user's email should be readable within a single interface (i.e. merged into a single listing), regardless of security classification.
3. To prevent leakage (write-down), a reply to an email should only be sent out at the same level as the original incoming message (or possibly at a higher level).
4. The clients should be able to utilize existing email (SMTP and POP3) servers.

4.2 JPmail Email Client

JPmail is an information flow aware application developed with Jif, that enables users to send, receive and reply to emails while also guaranteeing preservation of privacy (i.e., information is not leaked to unauthorized parties). The *JPmail system* is composed of four parts: (1) the *JPmail client*; (2) the operating system on which it is running; (3) the Internet; and (4) remote, public mail (SMTP and POP3) servers. The original JPmail client presumed that the operating system made no effort to protect the secrecy of data being sent out on network sockets. The client also presumed that remote mail servers could not be trusted to keep emails secret (to satisfy requirement #4). Consequently, to meet requirements #1 and #4, the original client incorporated a proprietary public key infrastructure (PKI) in order to encrypt all outgoing secret emails and decrypt all incoming secret

emails. Unfortunately, the complexity of handling the encryption and the corresponding declassifications (since releasing encrypted data is technically still leaking information, however small, about the unencrypted secret data) retarded the efforts of including features such as handling multiple security levels within a single client. Thus, we were not able to meet all the above-listed requirements for a secure email client (namely, #2 was left incomplete).

4.3 Integrating JPmail with SELinux

For this work, we updated the JPmail client to take advantage of SELinux security enforcement. This enabled a significant simplification of the client application's code and allowed us also to extend the client to handle multiple information flows within a single execution.

By integrating with SELinux security enforcement, we were able to simplify JPmail in the following ways:

- Utilizing secure communications (Labeled IPsec) allows the application to forego encryption while still satisfying requirement #1 from Section 4.1.
- Integrating a secure operating system changed our assumption about remote mail servers: knowing that the servers are confined at a certain security level (by SELinux) allows us to entrust our emails to the servers without encryption (except the encryption provided in transit by IPsec) without fear of leakage. Thus we still meet requirement #4.
- By introducing a mechanism for trusting the operating systems' secure sockets, problems related to declassification no longer need to be handled in the application. While not directly enabling us to meet requirement #2, this simplification of program logic and code complexity opened the door for us to include this feature, rather than requiring independent incarnations of the email client for different security levels.

The logic and information flow framework for requirement #3 was developed largely in previous work [11, 13]. With these changes, we are able to meet all the above-listed requirements. More details about how these requirements are met are given in the next section.

5 Implementation

The implementation described below implements the following email operations: reading, sending, and replying to emails (shown in Figure 2).

Reading email: Suppose Alice has secret clearance, so she is allowed to read emails at that security level and lower levels. This means that she can contact POP servers running at these levels to retrieve such emails. When she receives an email at a lower level (e.g., confidential), it is reclassified to secret. This fulfills criteria #2 and #4 from the information flow requirements in Section 4.1.

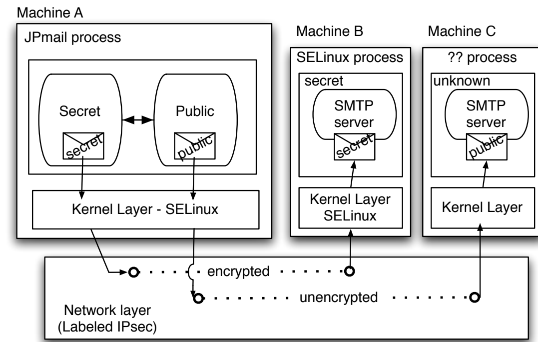

Figure 2: We configure SELinux to allow a trusted, Jif application to select the security levels required for a specific I/O channel. Jif is extended to allow applications to output unencrypted, secret data, because it is able to trust the OS to securely handle its secret data through Labeled IPsec sockets.

Sending email: Since Alice has secret clearance, her keyboard input enters at that security level. Declassification (e.g., encryption) is necessary to send the email onto the Internet. Such declassification must be authorized for secret email data. The output is labeled as secret and *JPmail* has to ensure that it is only sent to an SMTP server with a secret security level. Using Labeled IPsec fulfills criterion #1, ensuring that secret data is properly protected when it is sent out onto the Internet. In this case, the operating system is authorized to perform the proper declassification via encryption . This implementation also fulfills criterion #4 because the remote SMTP server need not be Jif-aware so long as it is sand-boxed by SELinux in the appropriate security context (guaranteed by the fact that a properly labeled IPsec socket could be established).

Replying to email: When replying, *JPmail* has to declassify the response (including the text of the original email) from the upgraded secrecy level (Alice's highest level) to the original level of the email. This must be done carefully to ensure that no secure information is leaked. The reply is sent to an SMTP server with the declassified security level. The algorithm and framework for this is not impacted by integration with SELinux and is discussed further in previous work [11, 13]. This part of the implementation fulfills criterion #3 in the information flow requirements.

5.1 Jif Infrastructure Extensions

In Jif, the only way to request resources, such as files, standard input and standard output, from the operating system is through a special `Runtime` class. We have extended the `Runtime` class to handle labeled sockets and files. (We consider the details here only for sockets—files were more straightforward, applying similar concepts.) Jif applications have a policy, external to the application, that establishes acceptable mappings between SELinux security contexts and Jif principals. For our example, we pre-

```
// Example Jif code tries to leak secret data
InputStream[{s3:}] = Runtime
  .openSocket(host, port, new label{s3:},
            "user_u:object_r:jpmail_t:s3")
  .getInputStream();
PrintStream[{}] pubFile = new PrintStream(Runtime
  .openFileWrite("leak.txt",false,new label{}));
char ch = in.read();
pubFile.print(ch); // ERR: illegal information flow
```

Figure 3: This Jif code demonstrates how Jif controls information flow through an application. This code opens a secret socket, reads data from the socket and attempts to write it to a public file. A compiler error would occur in this case because the code violates the noninterference property for confidentiality (no write-down).

sume our policy establishes that access is only allowed to a `user_u:object_r:jpmail_t:s3` (SELinux label) socket through a stream labeled `{s3:}` (Jif label). Furthermore, since Jif also ensures that a stream connected to a public file must be labeled `{}`, and Jif prevents data labeled `{s3:}` from flowing to channels labeled `{}`, a Jif program can be trusted to keep the secret socket input from being leaked to the public file. An example of this is given in Figure 3. (Note, that this code is simplified for illustration purposes. In order to compile in Jif, it must also handle all runtime exceptions.)

Our extension to Jif adds a call to the SELinux C library in the `Runtime.openSocket()` method (such as the first line in Figure 3). `openSocket` takes four arguments: the host IP address and port number we want to connect to, as well as the Jif label and the SELinux security context we would like to label the socket with. The `Runtime` then performs the proper security checks and socket initialization. The `Runtime` first creates an unconnected socket with the Java constructor. Then we call two C functions: (1) `getfd` to get the file descriptor on the socket and (2) `fsetfilecon` to attempt to change the context on the socket to `openSocket`'s fourth input parameter (in this case it would be `"user_u:object_r:jpmail_t:s3"`). If this succeeds (i.e., it doesn't throw a `SecurityException`), we attempt to connect the socket using the Java method, `sock.connect(...)`. This will automatically engage the IPsec subsystem, attempt to establish a security association for the socket and make the connection with the remote server socket. After making a connection, the security association (SA) is checked by making the kernel call to `getsockopt` with the `SO_PEERSEC` attribute in our C function `getSecurityContext`. If the SA does not have the security level requested by the Jif application, a `SecurityException` is thrown. If all this succeeds, the `Runtime.openSocket()` method returns the newly created, properly labeled socket. Otherwise, it throws a `SecurityException`.

Another important consideration regarding the protection of secure information for an email client is what label should be on data entered via the keyboard. Currently, the Jif `Runtime` class requires that the standard input must have the same or a higher security level than the process running the program. As we extend the Jif infrastructure for tighter integration with SELinux, this label could also be drawn from the X Window that the process is running in. A similar approach could be taken for standard output. We leave this as future work.

5.2 SELinux Implementation

This section presents an overview of a proof-of-concept system that implements comprehensive information flow enforcement using SELinux. This proof-of-concept system uses the SELinux Type Enforcement (TE) and MLS policies to enforce information flow secrecy. The support for MLS in Labeled IPsec was being developed as this work was done, so we built this system using the initial Labeled IPsec system. We discuss the implications of moving to the MLS Labeled IPsec in Section 6, and plan to port to that system.

Enabling *JPmail* to send information over the network at different security levels requires allowing the application to change the security context of their resources (sockets), so they can be associated to appropriate IPsec security associations. The following list presents the general requirements:

- *JPmail* must run in a context that enables it to access sockets with different MLS levels, for instance secret and public
- *JPmail* needs to relabel a socket's context to secret or public
- *JPmail*'s secret and public sockets need access to security associations of those respective labels

We created a separate domain for Jif applications since we are authorizing them to determine the security level of their resources. We do not want to give such control to other applications. Such privilege is given to Jif applications because they guarantee information flow enforcement over their own data. The following rules configure the environment to accept our new domain:

```
type jpmail_exec_t;
type jpmail_t;
domain_type(jpmail_t);
domain_entry_file(jpmail_t,jpmail_exec_t)
domain_auto_trans(user_t, jpmail_exec_t, jpmail_t);
```

We also need to assign special MLS attributes to our new domain: `mlsnetreadtoclr` and `mlsnetwritetoclr`. These attributes allow the application to read data from and write data to network resources (sockets in this case) if the MLS level of the resource lies within the application's MLS range.

```
typeattribute jpmail_t mlsnetreadtoclr, mlsnetwriteto-
clr;
```

The following rule indirectly defines the users that are allowed to run a Jif application (Since SELinux users have a role associated, any subject with this role is allowed to execute our Jif application):

```
role user_r types jpmail_t;
```

Since *JPmail* connects to POP and SMTP servers to receive and send messages, we need to give it access to sockets and security associations:

```
allow jpmail_t self:tcp_socket { create bind listen
    getopt getattr relabelfrom relabelto read write };
allow jpmail_t self:association { recvfrom
    sendto};
```

Previous statements give *JPmail* permissions to relabel its resources and use them.

The sockets' access to network communications is controlled by Labeled IPsec. Labeled IPsec [17] enables the information flow guarantees to reach other machines, by authorizing network communication only if the sockets on each machine have access to the label of the resulting security associations. *JPmail* relies on Labeled IPsec to guarantee confidentiality for data transmission.

IPsec rules must be specified that describe the secure communication requirements and labels of the connection between the two machines. These statements define that secret communication uses ESP in transport mode.

```
spdadd addr2 addr3 any
       -ctx 1 1 "user_u:object_r:jpmail_t:s1"
       -P out ipsec esp/transport//require ;
spdadd addr3 addr2 any
       -ctx 1 1 "user_u:object_r:jpmail_t:s1"
       -P in ipsec esp/transport//require ;
spdadd addr2 addr3 any
       -ctx 1 1 "user_u:object_r:jpmail_t:s1"
       -P in ipsec esp/transport//require ;
spdadd addr3 addr2 any
       -ctx 1 1 "user_u:object_r:jpmail_t:s1"
       -P out ipsec esp/transport//require ;
```

With the old Labeled IPsec implementation we need rules for every one of the possible MLS levels (s0,s1,...). A most recent version allows to specify a range, thus the level of the security association is assigned when it is created according to the actual MLS level of the involved socket and the range allowed by the IPsec rule.

An additional issue is whether it is acceptable to move the responsibility of encryption operations from the application to the operating system and labeled IPsec; encryption is usually required when downgrading information to send it through the network. Consistency between what is expected by the application (encryption and hash algorithms and authentication method) and what is actually available at the IPsec layer has to be checked by hand.

5.3 Results

Contrary to the usual case that operating systems do not trust applications, SELinux may trust the application to collaborate in enforcing the system's information flow policies. In Section 3.2, we identify that SELinux needs to depend on the application for: (1) labeling outputs to the system; (2) compliant information flow policies; and (3) authorized declassifiers. In this paper, we show that (1) is possible in SELinux using process and socket relabeling. We built the modified *JPmail* email client, and have a performance analysis of the impact of using IPsec for encryption [13].

To achieve (2), we need a service that verifies the compliance between the information flow policies. An architecture for this service [14] and an algorithm [15] are proposed in prior work. Unfortunately, identifying authorized declassifiers (3) is a manual task. Therefore, eliminating unnecessary declassifiers, such as removing the *JPmail* encryption, is beneficial.

In addition to removal of a declassifier, the integration of information flow control reduces the amount of secret data specific to the application. For example, private keys are no longer required to decrypt messages within the application. This reduces the problem that SELinux has to depend on the correct labeling of data that originates within the application.

6 Discussion

This work is a proof-of-concept to show what is necessary in general for SELinux to leverage security-typed applications to enforce information flow comprehensively. The proof-of-concept system is limited in two key ways: (1) it does not leverage MLS labels for secrecy and (2) it does not enforce integrity information flow controls. We discuss the efficacy of addressing these limitations in practice below.

First, using MLS labels to enforce information flow secrecy is a natural extension. With the inclusion of the MLS extensions for Labeled IPsec in the near future, we can port *JPmail* to use MLS labels in a more consistent way. Of the SELinux tasks, listed at the beginning of Section 5.2, all are straightforward once the system starts the *JPmail* process in the appropriate MLS range.

Jif can represent a range of secrecy labels using *meets*, but this is awkward to use. Fortunately, most SELinux objects will have a single access class. We have only seen MLS ranges for directory objects, thusfar. The *JPmail* process would have an MLS range for its label that covers the emails that it can send (i.e., lowest secrecy send) and receive (i.e., highest secrecy receive). This would enable the user to use the *JPmail* client to send an email with any secrecy class within this secrecy range. Clearly, the breadth of the range depends on the declassifiers for message secrecy. Sockets would be created for each label of message that could be sent, but this may result in a large number of sockets being used. Further investigation is necessary to determine this impact.

Second, handling information flow integrity is limited in two ways in our proof-of-concept: (1) SELinux does not have an integrity lattice policy and (2) we only use the secrecy lattice in Jif at present. Our view is that the TE policy represents integrity protections whereas the MLS policy is for secrecy. In the past, we have associated integrity labels with subject types [19]. Since we have important processes (i.e., trusted) and others, a simple two-level integrity hierarchy has sufficed. Further, a recent extension of the Jif environment supports an integrity lattice policy [5]. Thus, we envision that integrity information can be enforced using the SELinux TE policy to identify low-to-high integrity

flows and the Jif integrity lattice and declassifiers to ensure that the low integrity is immediately upgraded or discarded according to the Clark-Wilson integrity model [6].

7 Conclusions and Future Work

We have described and implemented a model for integrating OS MAC security with application-level information flow controls. We demonstrate the feasibility of this model by modifying a multi-level secure email client, developed in the security-typed language Jif, to utilize SELinux security mechanisms such as Type Enforcement and Labeled IPsec. We extended the Jif infrastructure to support this interaction and we describe the policy and system calls necessary in SELinux for a successful integration. Furthermore, we note that this modification led to several improvements to the application in accord with good practices for security environments: it simplifies the code of the application, and it distributes various security tasks to the best mechanism (OS, network or application) who can perform each task.

By developing this work we also uncovered some areas for further research. It is important to formalize a theory of SELinux behavior such that it can generate provable guarantees similar to Jif's. Also, we identified the need for a formal definition of compliance between the operating system and application information flow policies.

References

[1] K. Ahmadizadeh. Integrating jif into eclipse. Honors Thesis. Pennsylvania State University., May 2006.

[2] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as NTIS AD-A023 588.

[3] 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.).

[4] W.E. Boebert and R.Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Conference on Computer Security*, pages 18–27, 1985.

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

[6] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–195, 1987.

[7] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[8] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the linux security modules framework. In *CCS'05: Proceedings of 12th ACM Conference on Computer and Communications Security*, page To Appear, New York, NY, USA, November 2005. ACM Press.

[9] V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, 2007. To appear.

[10] J. A. Goguen and J. Meseguer. Security policies and security models. pages 11–20, April 1982.

[11] 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.

[12] 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.

[13] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. Breaking Down the Walls of Mutual Distrust: Security-typed Email Using Labeled IPsec. Technical Report NAS-TR-0049-2006, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, September 2006.

[14] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. From trusted to secure: Building and executing applications that enforce system security. Technical Report NAS-TR-0061-2007, Networking and Security Research Center, Department of Computer Science, Pennsylvania State University, 2007.

[15] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A logical specification and analysis for SELinux MLS policy. Technical Report NAS-TR-0058-2007, Networking and Security Research Center, Department of Computer Science, Pennsylvania State University, 2007.

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

[17] T. Jaeger, S. Hallyn, and J. Latten. Leveraging IPsec for mandatory access control of linux network communications. In *Proceedings of the 21st Annual Computer Security Applications Conference (AC-SAC'05)*, Tucson, Arizona, Dec. 5-9 2005.

[18] T. Jaeger, D. King, K. Butler, S. Hallyn, J. Latten, and X. Zhang. Leveraging IPsec for mandatory access control across systems. In *Proceedings of the Second International Conference on Security and Privacy in Communication Networks (SecureComm 2006)*, Baltimore, MD, USA, August 2006.

[19] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, pages 59–74, August 2003.

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

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

[22] 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.

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

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

[25] 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.

[26] 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.

[27] V. Simonet. The Flow Caml System: Documentation and User's Manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003. ©INRIA.

[28] Tresys Technology. SE Linux Policy Server. Available at `http://www.tresys.com/selinux/selinux_policy_server`.