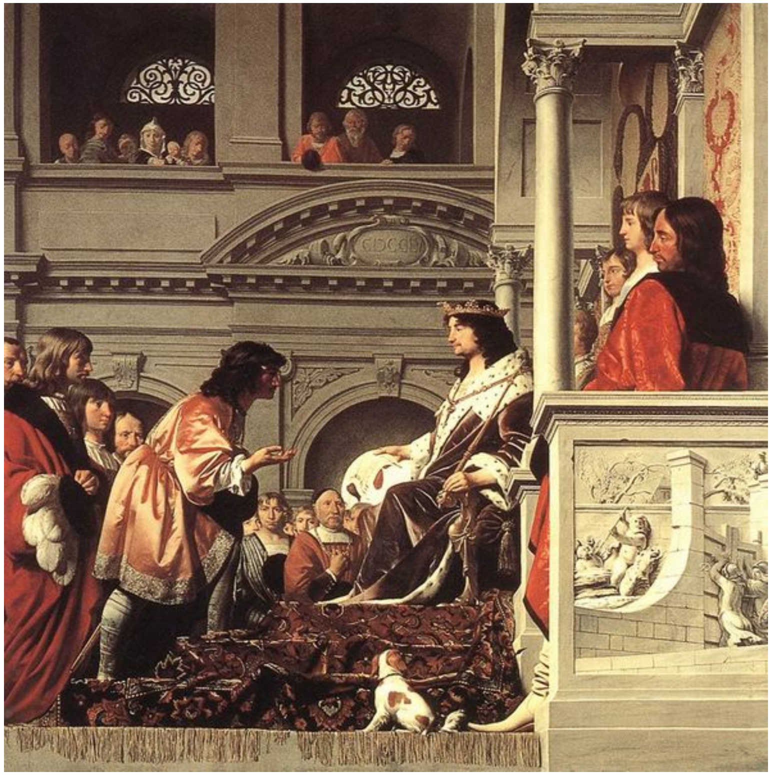
Integrating SELinux and Security-typed Languages

Sandra Rueda - ruedarod@cse.psu.edu
Boniface Hicks, Trent Jaeger, Patrick McDaniel

Systems and Internet Infrastructure Security Lab
Department of Computer Science and Engineering

The Issue

- Operating systems like SELinux enforce information flow policies at the granularity of application inputs and outputs.
- …but… some applications need privileges (access to multiple security levels):
  - Server software
  - Client software: e-mail clients, web browsers
  - High integrity programs with low integrity inputs



The Issue [2]

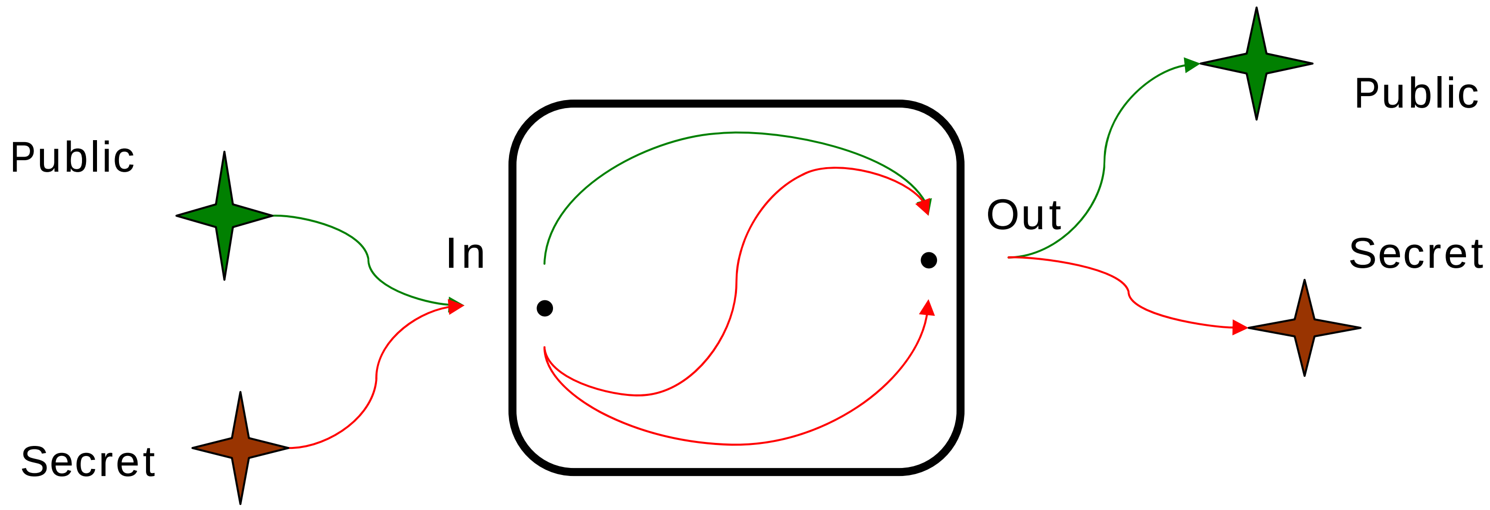
- The OS trusts that privileged applications preserve information flow policies

SELinux:

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 mgmt	hald, dmidecode, udev, kudzu
Miscellaneous services	passwd, tmpreaper, insmod, getty, consoletype, pam_console

The Issue [3]

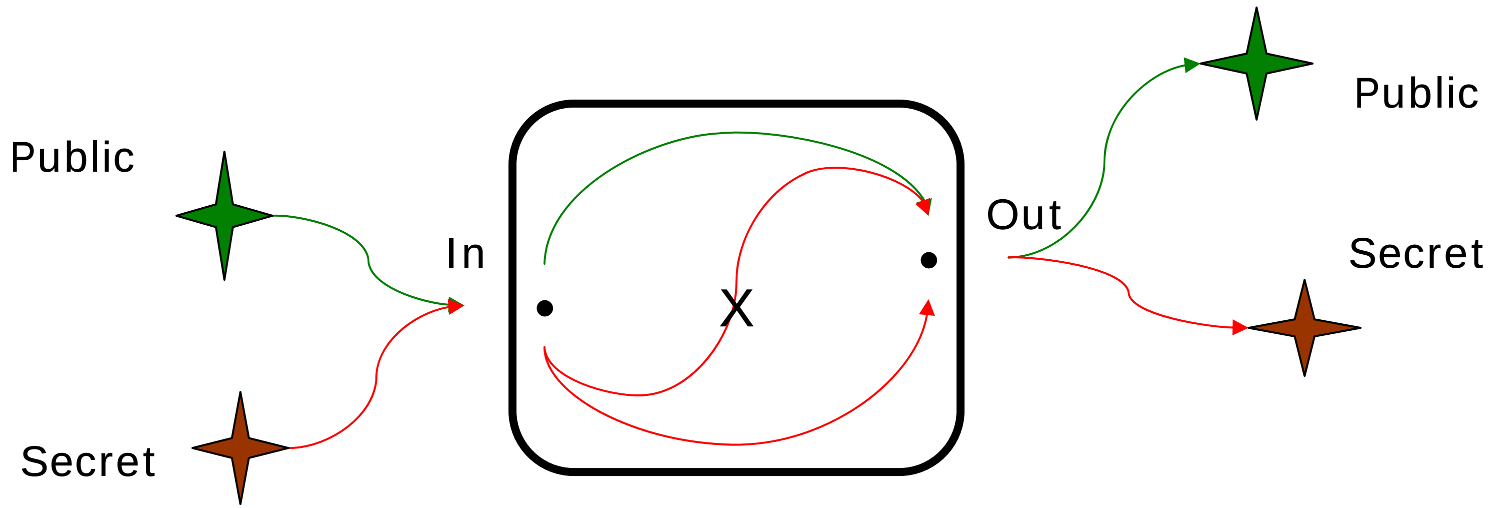
- Neither SELinux nor any other operating system have any means of tracing information flow management within an application



Information Flow Enforcement

- Can applications show they are enforcing information flow policies?

  → This is the goal of security-typed languages



Security-typed Languages

- Security-typed compilers guarantee enforcement of lattice information flow policies.

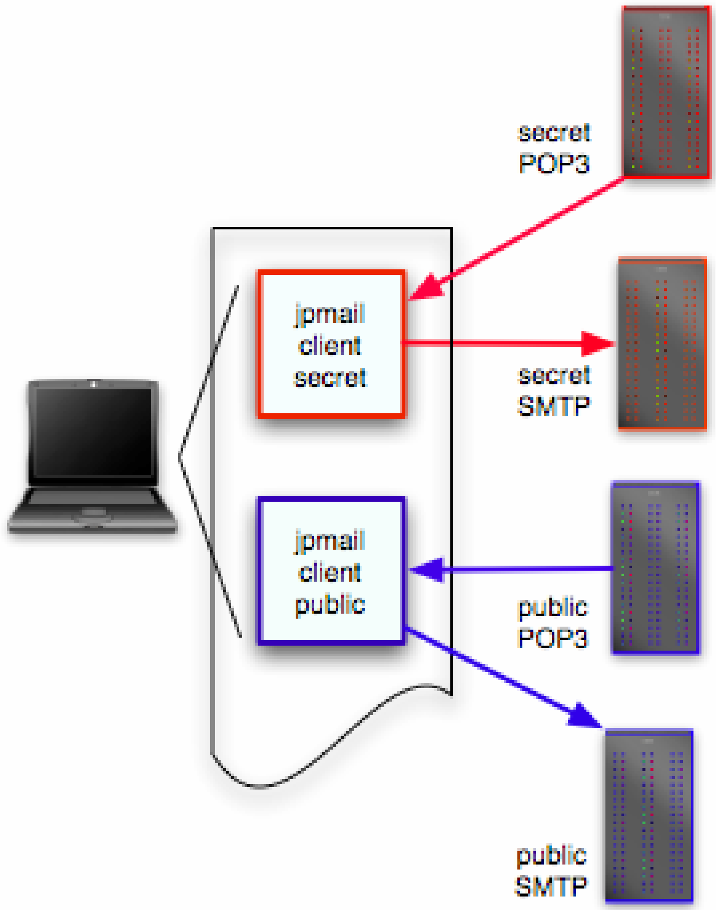
If a program does not meet the policy
$\rightarrow$ it does not compile

Security-typed Languages [2]

- Variables are augmented with annotations that define a policy
- Policies are enforced by compile-time type checking

```
Pseudo-code:

InputStream {sec} in = stdin(…);
String {sec} passwd = in.readLine(…);
Socket {pub} leak = openSocket(…);
leak.print(passwd); // Compiler message:
                    // Illegal Flow!
```

Analysis: Client Application



- *JPmail*: information flow aware email client
  - Single interface to read all levels of emails. It must preserve noninterference!
  - Secret e-mails must be encrypted before sending them out
  - Any reply should be sent out at the same level as the original message

Analysis: System Application

- *logrotate*
  - It is a service that rotates logs
  - Logs may span various security levels on a system
  - It works based on a configuration file
  - It is required to have separation among:
    - log files of different programs
    - log files and configuration files for a single domain and among domains

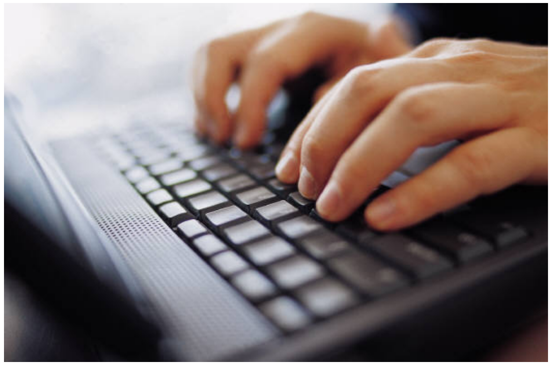
Related Work

- Options to handle applications that require access to multiple security levels:
  - Separation of privilege (virtual or physical gap)
    - Require additional resources, more complex management
  - Manual Inspection
    - Prone to error
  - User level policy server
    - No guarantee of completeness
    - We are still subject to manual inspection

Our Solution

- Develop applications that enforce system information flow policies and are able to prove it to the operating system

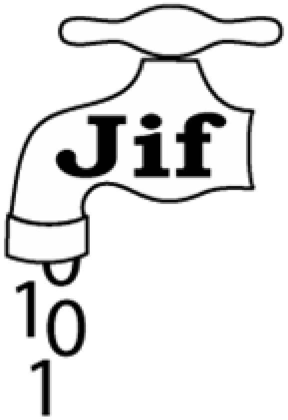
How ?



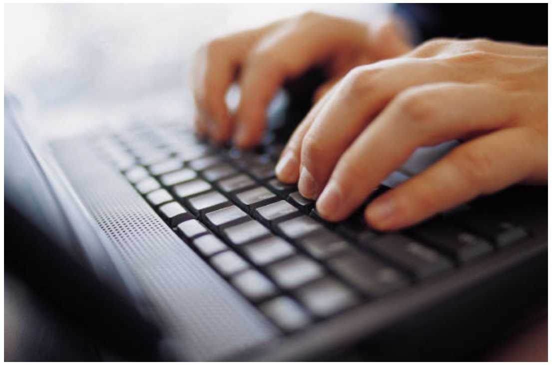
- Two main tasks:
  - Develop applications aware of security goals and with means to prove information flow enforcement → T1. Develop with Security Typed Languages
  - Integrate these applications with SELinux → T2. Integration Framework

T1. Application Development

- We use *Jif*
  - Jif = Java + Information Flow
  - Currently, Jif is the most mature security-typed language
  - Where are the real Jif applications ?
    - JPmail [Understanding practical application development in security-typed languages. ACSAC 2006]
      - High level configurable policy
      - Connected with existing system



T2. Integration Framework



- We identify three main tasks:
  - Implement mechanisms for the application to determine the label of its input channels

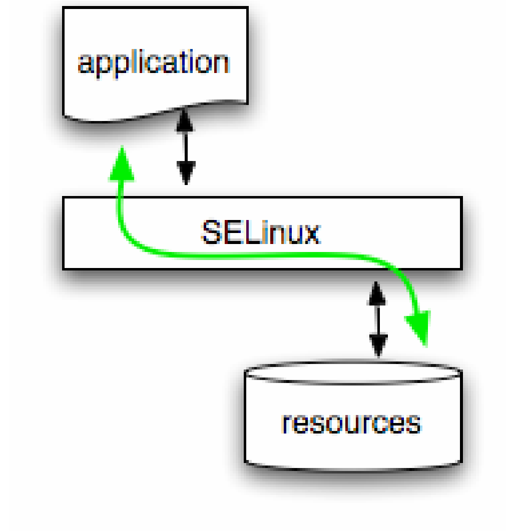
    → a) Label Exchange (⇒)
  - Implement mechanisms for the application to communicate to the operating system the label of the outputs

    → b) Label Exchange (⇐)
  - Implement mechanisms by which an application can prove its information flow enforcement is consistent with the system policy

    → c) Policy Compliance Testing

a,b) Label Exchange

- Tasks a,b:
  - to get labels for inputs
  - to assign labels to outputs
- To do so we need:
  - A mapping between SELinux and application labels
  - Be able to exchange labels at runtime (application inputs and outputs)




```
FileInputStream {sec} in = FileInputStream(…);
String {sec} data = in.readLine(…);
Socket {pub} leak = openSocket(…);
leak.print(passwd);
```

SELinux file label

SELinux socket label

Label Exchange [2]

- Mapping between SELinux and application labels

Label Mapping:

SELinux *Jif*

user_u:object_r:jpmail_t:s3 → {.*:.*:.*:s3}

- Exchange of labels at runtime (application inputs and outputs)

Jif runtime environment was extended

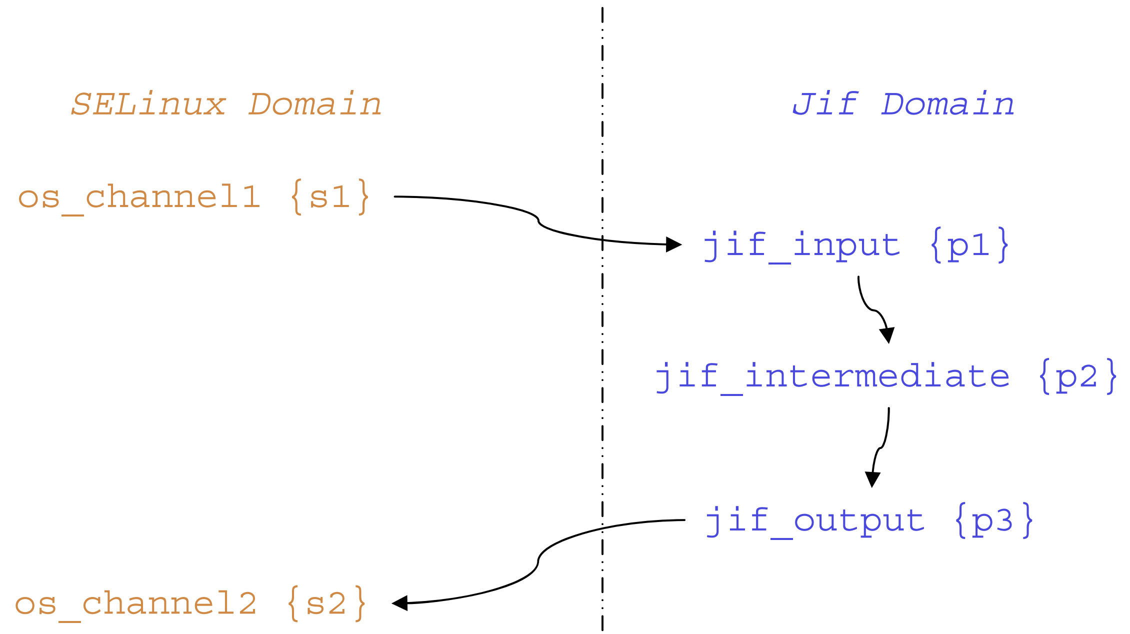
OS functions supporting the extension

```
Socket [{s3:}] stream =
  openSocket(host,port,new label(
           "user_u:object_r:jpmail_t:s3"));
```

```
getfd()
fsetfilecon()
getsockopt()
```

c) Policy Compliance

- We want to prove that the application enforces a policy that is consistent with the SELinux policy → it does not add flows that are not allowed in the operating system

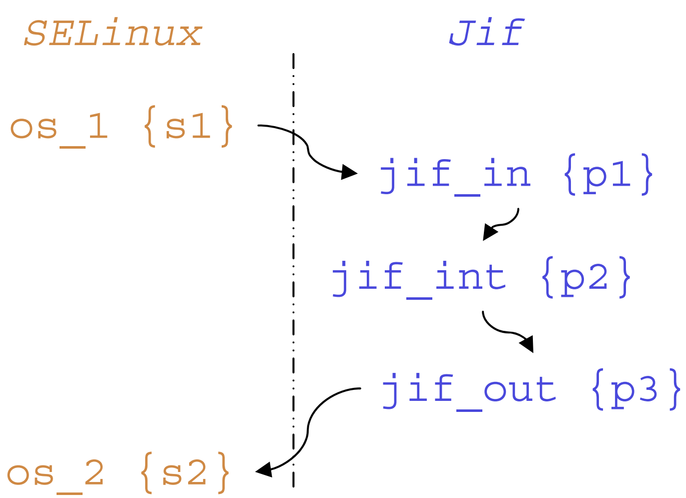


s1 ≤ s2 ?

Jif guarantees p1 ≤ p2
and p2 ≤ p3
(p3 is at least as secret as p2
and p2 is at least as secret as p1)

Policy Compliance [2]

- Implementation:
  1. Detect information flows allowed in the OS and the App
  2. Determine the security levels that are shared between the OS and the App
  3. Flows allowed in App $\subseteq$ Flows allowed in the OS



1. OS flows: {s1 -> s2, s4 -> s5}
   Application flows: {p1 -> p2, p2 -> p3, p1 -> p3}
2.

Label Mapping:

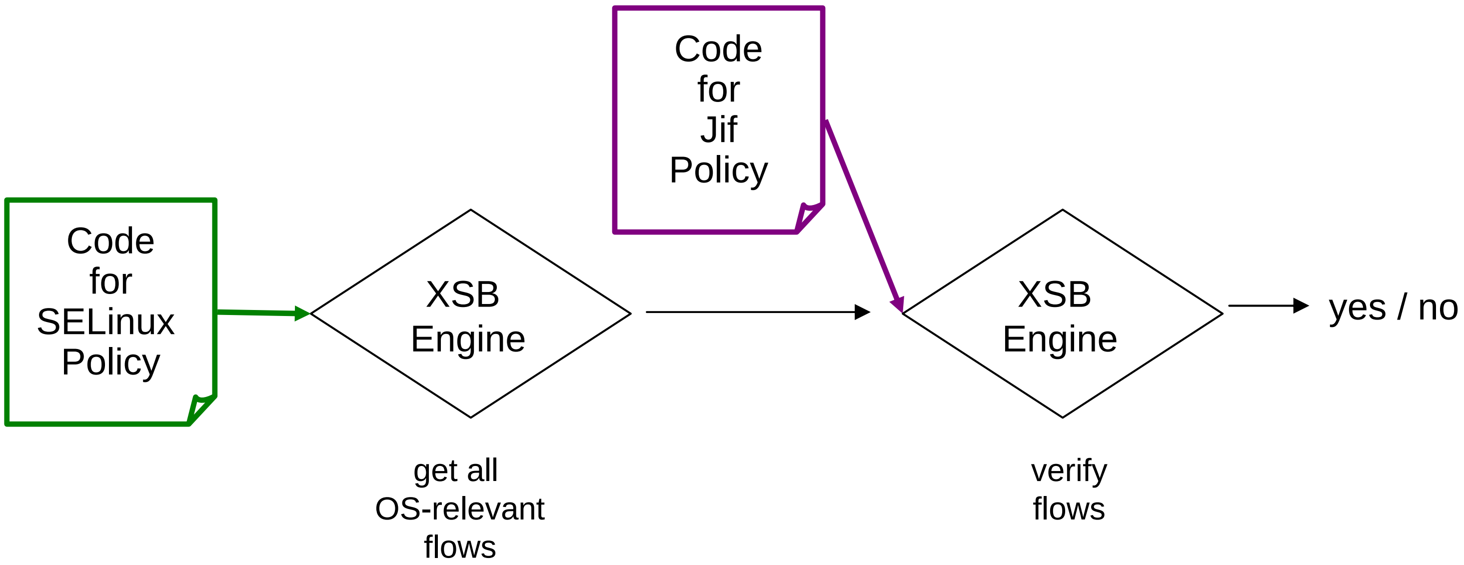
SELinux		*Jif*
user_u:object_r:jpmail_t:s1	$\longleftrightarrow$	{.*:.*:.*:p1}
user_u:object_r:jpmail_t:s2	$\longleftrightarrow$	{.*:.*:.*:p3}

renaming shared levels:
p2 is not shared
p1 -> p3 becomes s1 -> s2

3. {s1 -> s2} $\subseteq$ {s1 -> s2, s4 -> s5}

Policy Compliance [3]

- Implementation [NAS-TR-0058-2007. CSE SIIS Lab 07]
  - Translation of policy rules to Prolog statements
  - XSB Prolog engine
    - Tracing of flows allowed by the OS
    - Tracing of flows allowed by the application



Implementation Example

- We integrated JPmail and JPlogrotate with SELinux
- SELinux rules for JPmail:
  - We assigned MLS-related attributes to our application
  - We allowed our application to set up the level of its output resources (at run-time those levels depend on the level of the input)
  - We used Labeled IPsec to create appropriate network connections

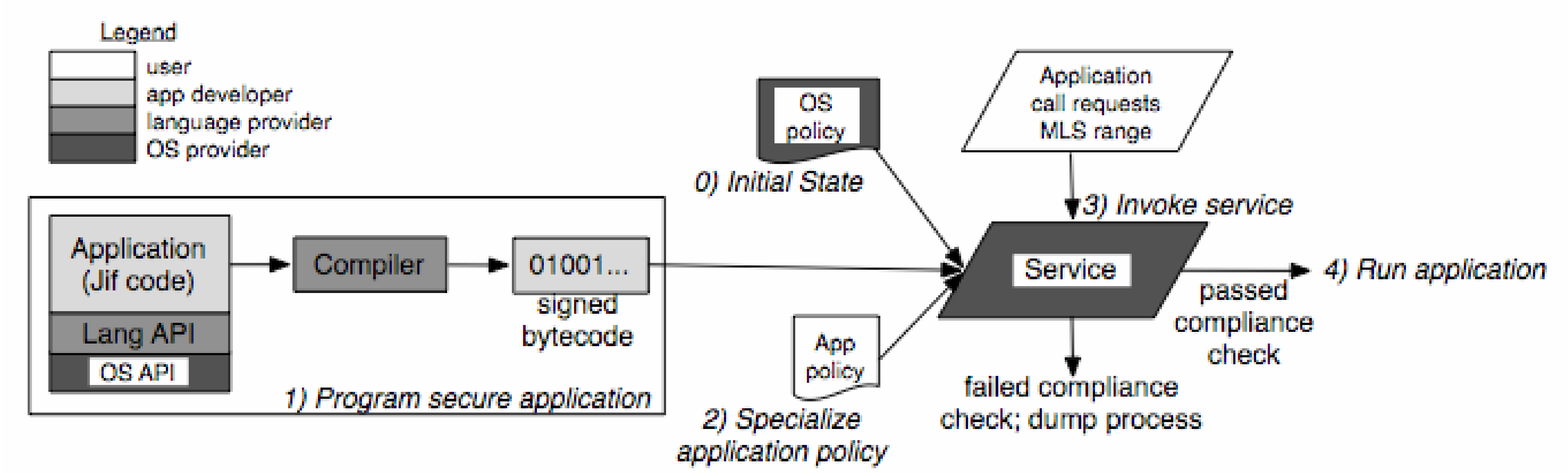
```
type jpmail_t

typeattribute jpmail_t mlsnetreadtoclr
typeattribute jpmail_t mlsnetwritetoclr

allow jpmail_t self:tcp_socket relabelfrom relabelto
allow jpmail_t self:association recvfrom sendto
```

```
spdadd addr1 addr2 any -ctx 1 1
         “user_u:object_r:jpmail_t:s1”
         -P in|out ipsec esp/transport//require;
```

Summary

- Overview of the system:

  1. Application is developed in security-typed language
  2. Developer defines high-level policy for the application
  3. Application is invoked
  4. The operating system checks policy compliance
  5. Application is initiated



Our Contribution

- We developed a model to build applications that enforce system information flow policies and are able to prove it to the OS
  – Jif for application information flow management
  – SELinux for system information flow management
  – Service to run the applications that meet our requirements
- We implemented and tested the model!

- *We designed and implemented a comprehensive framework that enables the integration of security-typed applications and SELinux to enforce end-to-end information flow policies.*

Future Work

- Integrity Management
  - Our current implementation focus on confidentiality
- Analysis of SELinux policy
  - Considering previous work in the area
  - Analysis of SELinux/Application policies to determine whether they meet specific security properties or not
- Compliance across multiple systems
  - Mechanisms to check compliance among policies that rule different systems

Questions

Secure languages at PSU SIIS Lab http://siis.cse.psu.edu

- Understanding Practical Application Development in Security-typed Languages. [ACSAC 06].
- A Logical Specification and Analysis for SELinux MLS Policy. Technical Report [NAS-TR-0058-2007,CSE SIIS Lab 07].
- From Trusted to Secure: Building and Executing Applications that Enforce System Security. [NAS-TR-0061-2007,CSE SIIS Lab 07], [USENIX Annual 07 - to appear].

Declassifiers

- Noninterference property is too strict
- Declassifiers allow relabeling under specific circumstances
- Real applications require declassifiers. For example to send encrypted messages
- Our Jif extension enables a developer to define the set of declassifiers an application may use
- Consistency application declassifiers vs. operating system declassifiers is currently done manually. Improving this process is part of our future work
- *Trusted Declassification: High level policy for a security typed language [Hicks et al. ACM SIGPLAN06]*