Towards a System-Wide and Transparent Security Mechanism using Language-Level Information Flow Control

Mohammad R. Azadmanesh
School of Computer Engineering
Iran University of Science and Technology
azadmanesh@comp.iust.ac.ir

Mohsen Sharifi
School of Computer Engineering
Iran University of Science and Technology
msharifi@iust.ac.ir

ABSTRACT
Operating systems try to provide secure platforms using appropriate security mechanisms like DAC and MAC. In spite of this, information confidentiality is not totally provided when information flows in the program memory space. Programming language level security techniques have thus been introduced to provide secure information flow inside programs. Existing works on programming language level are problematic though because their information flow policies have not been integrated into the underlying system security policies. In this paper we propose a dynamic solution for tracking and enforcing information flow policies inside the Java framework that is integrated with a trusted operating system namely SELinux. Our solution focuses on internal structure of JVM, implicating no modification to Java programming language. Experimental results have shown a bearable runtime overhead on running programs.

Categories and Subject Descriptors

General Terms
Security, Languages

Keywords
Information Flow; Mandatory Access Control; Language-Level Information Flow; system-wide security

1. INTRODUCTION
The flow of information in ever expanding networks of computers has long been recognized as a source for security breaches: “Networks, because they allow computers to share data and distribute processing, can potentially serve as a way to break into computer systems and leak information” [24].

Security experts have also quoted the necessity of operating system security to overall system security, as it has been quoted by Frank Mayer et al. [22] that “a design that tries to create security without support of the underlying operating system is a “fortress built upon the sand [4]” with no secure foundation upon which to sit”. Traditional and even recent OSs’ security mechanism called Discretionary Access Control had a big shortcoming because access control decisions were taken based on user’s discretion. So if a user trusted an application, that application could use all of that user’s privileges to access OS resources, a type of threat called Trojan horse.

Although operating systems have been able to track information flow in their domain using the MAC mechanism, there is no guarantee that a guest program respects all of system’s confidentiality policies. A program can read information from a confidential file and write them into a public file, resulting in downgrading and thereby leakage of information. On the other side, those programs running in user-space that try to control information flow, are still unaware of security policies that have been declared for their underlying operating systems and their policies may contradict with system policies.

In order to have a system-wide security mechanism, all layers of the system should control information flow and have their access decisions based on a common security policy defined for the whole system. As we said, OS was a first step toward this goal that has provided its share to this cause. Yet, having a secure platform does not necessarily imply a secure system. It is therefore reasonable to have the same security policies enforced on information inside both program and operating system while information flows from OS into program or vice versa.

Some user-space programs like X window [16], apache server and postgresql have been extended to become aware of underlying OS security features in order to track information flows inside their memory space at runtime but the generalization of program awareness to all network applications can be cumbersome and perhaps impossible. This is because doing so needs an exact analysis of program for determining types of resources and operations on these resources and then implementing mechanism for controlling access inside program. This can lead to redesign of a large program which would in turn bring many challenges ahead.

Language-level information flow techniques provide a platform in which correctness of information flow inside programs written at those languages are guaranteed. Such techniques are however independent of the information flow policies that are enforced by the underlying operating systems, resulting in incompatibilities between the two sets of policies.
A good security mechanism is one where policies can be defined declaratively in a way that is not frozen inside program code. In the latter form, any changes to policies may well require a re-compilation of the program that can be too costly for large applications. Besides, most want to keep the standard syntax of programming language intact. A developer who is expert in writing code in a programming language would prefer to write code in his familiar language syntax rather than an unfamiliar one.

With the above issues in mind, we propose a solution wherein information flows from all input channels into JVM are recognized and the necessary modifications are made to associate security context to information. Our solution also provides a framework that tracks information flow inside JVM while access control is performed. This is achieved by modifications made to the JVM execution engine; resulting in a new engine we call SEJVM.

In contrast to other dynamic information flow tracking approaches that need a static analysis of the code before execution to detect implicit information flows; our modified execution engine uses access control hooks in the body of interpreting methods of bytecodes to prevent implicit flows. The modified JVM protects the output channels from JVM to operating system too in order to prevent information from getting exposed illegally. By modifying the JVM execution engine and the body of Java API methods, we do not require any changes to the programming language syntax or to the existing Java source codes. SEJVM is implemented inside JableVM, which is an open source JVM.

The rest of paper is organized as follows. Section 2 states the problem and our scope. Section 3 discusses works related to language-level information flow control and their tradeoffs. Section 4 gives an overview of SELinux architecture. Section 5 presents our approach to the problem and our solution. After that, we describe details of steps toward building our solution inside any virtual machine. It includes the way to derive confidentiality policies in line with SELinux, associating these policies to data and determining thread labels. We also describe a framework for tracking information flow and access control inside JVM. Section 6 reports the results of running benchmarks on SEJVM. Conclusion and further works are presented in Section 7.

### 2. PROBLEM STATEMENT

Inside a MAC-based OS kernel like SELinux, all resources are labeled with a security context that determines the policies on accessing resources. For example, processes running on behalf of a remote user such as a remote shell process are assigned a security context that determines the domain of their actions.

Program interactions with OS are controlled based on privileges granted to the process corresponding to the program. If a process is authorized, it is granted access to a resource and it can retrieve information into its memory space where OS has no control over it anymore. OS relies on the privileged process to keep information confidential.

But this assumption may not be correct, because a user-level process inside itself can host multi threads that can be running on behalf of different users who are interacting with the program through network. It is not therefore true to assume that all the remote users are at the same level of trustworthiness.

As an example, consider JBoss application server running as a single Java process (See Figure 1). Remote users ask for information through sending HTTP requests to a host where JBoss process is running. The JBoss process catches each request and allocates a thread (it can be a new one or a pre-built thread) for processing the request.

All I/O requests by threads are issued to OS through this single process containing them and its privilege would be the one set up for it based on system policies once it was created. In this case, the assigned security domain is the domain of a simple Java process that is an unconfined domain based on SELinux Targeted policy. So requests issued from different threads in different security domains are all treated the same. Yet worse, if some confidential information is read into program space by a privileged thread, another unprivileged thread residing in the same process can access that information as long as information access is done in user-level space and inside running process’s memory space. That is because the OS does not have any control over internal accesses to a process memory space done by different threads running inside the process.

Illegal information flows are not always deliberate. Consider a program that reads information from a confidential file and then writes them into a public file, violating confidentiality of information through the illegal channel provided by the running program, without any malicious minds.

### 3. RELATED WORK

In general one can divide programming language level security approaches into three different categories: static, dynamic and hybrid.

In the static analysis approach, program information flow is analyzed before running and compared with a predefined policy. Dening and Dening [6] were the pioneers to propose static information flow control in programs. They also proposed a lattice structure for label hierarchy. Though quite theoretical, their approach has been leveraged in a number of languages [12, 25].

Jflow and its successor, Jif [19, 21], have extended the Java programming language with a security-typed system in which data type declarations are extended with security label declarations and information flows are controlled at compile time using type checking. They are actually proposed as a practical language for developing secure applications. A static analysis ensures secure information flow inside programs before running. Although this language is derived from Java, it has a different syntax than the
original Java language. The policies are frozen into source code of programs such that any changes to the policy require a recompilation of programs. In our solution, policies are defined at system level and no source code modification is needed. By removing the necessity for static analysis, our solution can be applied to compiled Java programs without any changes to their bytecodes.

By introducing noninterference [9] as a strong and end-to-end security property and proving that type systems can enforce noninterference property, a research area was opened to prove noninterference for other security-typed languages [27].

While none of these systems were integrated with system policies, B. Hicks et al. [10] introduced the concept of integrating security-typed languages to SELinux. They have taken Jif as the security-typed language and added annotations to variable declarations to include SELinux security context besides Jif labels. Since system policies and program policies are quite independent, there exists a phase for testing compliance of system policy with policies defined inside program source code. Our solution inherits its policies completely from OS policies and therefore there is no need for compliance checking. Their solution is static and needs program annotating while ours is dynamic and transparent to program source code.

Purely static techniques do not consider dynamic nature of policies defined for resources. Such mechanisms assign labels to program variables and do not consider labels of data source at runtime. Therefore freezing policies in compile time cannot enforce correct information flow.

Dynamic information flow techniques are a variation of information flow techniques that computes dynamic labels for each piece of information at runtime. Haldar et al. [13, 14] provide such a mechanism for the Java virtual machine at the granularity of Java objects. Their work has nothing to do with implicit flows and is not integrated with system policies. This is in contrast to our work that handles in finer granules, prevents implicit flows and is integrated with system policies.

Due to the shortages of pure dynamic approaches for detecting implicit information flows through control flow, hybrid techniques have been proposed too. RIFLE [26] is a hybrid approach that uses a combination of software and hardware for information flow control. At load time, binaries are rewritten to a new one that appends security labels to instructions. Information flow is tracked by modifying hardware architecture to track labels on words. A dataflow analysis is also performed on binary before executing to convert implicit flows to explicit ones to be tracked by the hardware.

In [5], a static analysis is performed on Java bytecodes before running to detect implicit information flows and necessary annotation is made into code to assist runtime environment for computing dynamic labels. This solution is not integrated with system policies either. It has not got any policy language, resulting in insertion of policies inside code like other solutions.

Having got familiar with shortcomings of different approaches, we defined our goal as a transparent and system-wide security solution in which information flown from OS into program space is kept confidential. We take Java framework and show how we can integrate it with SELinux in order to provide a secure platform thereby fulfilling system-wide security.

4. SELinux

SELinux is an implementation of Flåsk security architecture inside the Linux kernel, done by National Security Agency in December 2000 [22]. The Flask architecture cleanly separates the definition of the policy logic from the enforcement mechanism. SELinux was first released as a set of kernel patches for the 2.2.x kernel. This has resulted in a new version of Linux with flexible mandatory access control over Linux abstractions. The Linux kernel acts as an object manager. A security server and an AVC have been incorporated into the kernel. The kernel has been instrumented to intercept all access to kernel resources and labeling decisions and make the necessary access and labeling decision requests to AVC. Interception is done through hooks put inside kernel. The security server implements a security model that is a combination of Type Enforcement (TE), Role Based Access Control (RBAC), and Multi Level Security (MLS). SELinux is capable of supporting many types of policies enabling it to meet various system goals. Policies can be defined to provide strong process isolation and to support least privilege. They can be used to meet the integrity goals of the system. The flexibility of SELinux allows policies to be tailored to meet the specific security goals of applications.

5. OUR APPROACH

In order to keep the syntax of programming language intact and have integrated security policies with the underlying system’s policies, we propose our solution at the level of programming language pragmatics, which is the JVM. Since SELinux employs Mandatory Access Control (MAC) mechanism for controlling information flow, we should also propose a MAC framework for the JVM to be integrated with the underlying MAC mechanism. A MAC mechanism is an information flow control mechanism in which each piece of information is labeled with a security level and information flow is controlled by augmenting the ordinary computation of data within a running program with a simultaneous computation of the corresponding label that controls its future dissemination. In order to have a MAC mechanism inside JVM, we should determine types of information and their security context. After that, we need to provide facilities to associate this label with data during its lifespan. As data is going to be accessed or written to output channels, we need to provide necessary access controls before access can be taken. All of these steps will be described in detail in following sections.

5.1 Determining data confidentiality policies

As we stated before, we are going to track flow of incoming information from OS into programs. Therefore the confidential data that need to be protected is all those which are coming from OS. Now, we need to determine security labels such that it is compatible with the related information inside OS. Suppose a request for a file is issued by a program. After reading the file, part of its contents are conveyed into program space and delivered to the program in the form of a program data object. This data object contains part of information belonging to the file inside OS and therefore should be labeled as the file. Most existing works suffer from the absence of a flexible and declarative way for specifying security policies. They either freeze policies inside program source code or try to develop a new language for specifying policies. Our solution has an interesting property in facing with this challenge. Remind that, in SELinux, all resources are labeled with a security context. Since all input
into program are mediated by OS and because of dynamic nature of our solution that results in labels to be attached to data not to variables, we can determine data labels through asking OS for input data’s labels besides data itself. This can happen at the channels defined between OS and the programming language runtime environment.

In Java, access to OS resources is made through Java API. This API provides the connection through native methods that are implemented in a low-level programming language like C using JNI (Java Native Interface) technology. Therefore, we can manipulate input channels such that besides moving data, security context of information is also brought into JVM memory space and attached to the data. Figure 2 shows the architecture of the modified channel.

To do so, we need to determine the types of channels provided by Java framework and their location among API packages. In Java, there are two types of channels for communicating with OS: file channels and socket channels. Each of these channels, based on their ability to support non-blocking I/O is split into channel-based and stream-based I/O.

Stream-based I/O for files is implemented by the classes FileInputStream/FileOutputStream in the package java.io. All other streams whose data is provided through OS file system contain an instance of these classes as their main internal stream and filter information into/from these channels.

These classes isolate programmers from details of underlying OS. Every instance of FileInputStream/FileOutputStream contains an instance of FileChannel that provides channel-based I/O for files. This channel is implemented by FileChannel at the package java.nio.

The interface defined for these classes contain methods for reading and writing to a file. Calling these methods results in a hierarchy of method calls that ultimately ends in calling a method prefixed with “native”. The body of this method is where we need to manipulate for fetching security context of information besides information itself. Note that, since just the body of method is modified, its signature and compatibility with Java standards are kept. On the other hand, these methods are limited and can be easily determined for modification.

Stream-based socket channels are provided by the classes SocketInputStream/SocketOutputStream in the package java.net. These classes provide network I/O for Java programs. Channel-based I/O for sockets is implemented by SocketChannel in the package java.nio.channels.

Reading data from a socket is done through an instance of class SocketInputStream. This class is an extension of class FileInputStream whose read method is overridden to read from a socket instead of a file. The ultimate method call that is responsible for native reading from socket and is prefixed with “native” is where we need to manipulate.

Note that standard input, output and error devices such as console are covered with this scenario since they are considered as files with file descriptors.

For expressions, there are intermediate results that need to be labeled. Since the security model in SELinux is a combination of TE, RBAC and MLS, and Least Upper Bound (LUB) cannot be determined for TE and RBAC attributes, we assign running thread’s security context to the intermediate values.

Now that we have recognized appropriate locations in API, we can add suitable system calls at those places to retrieve security context of information. This can be done by calling the method fgetecon(3) from libselinux with appropriate parameters.

Having manipulated input channels, we ensure that all information passed from OS into program are labeled with their corresponding labels inside OS transparently and without any interference of developers. Meanwhile, standard API structure is kept intact since we have just modified the body of methods not their signature. At runtime, program data objects containing a part of an OS resource such as a file, would respect policies defined for its container inside OS. This is the first step taken towards tracking information flow inside JVM.

### 5.2 Attaching labels to program data objects

So far, we were able to track information flow from OS into program gateways. But question arises that how labels are attached to pieces of data read from OS. To answer this question, we divide different types of data objects inside JVM to non-array objects, array objects and primitive values. Non-array objects are ones that are neither array typed nor primitive typed. Single instances of different classes belong to this category. Array object is a vector of other objects and the primitive value is the one whose data type is among built-in data types in the language.

As long as we consider OS resources as the only source of information for programs, we will deal with array objects and primitive values. That is because information read from OS, reside in the program space in the form of one of these two categories. In further works we show that the security label of non-array objects can be used to protect special resources in the context of Java framework like Reflection API and Class loaders.

The Java language specification has not determined any special representation for objects residing in the heap. Each JVM implementer is thus responsible to a representation for objects at runtime. For any specific implementation, one can extend the array representation to have an extra array field for maintaining the security context of information it contains. We attach an individual security context to each of the elements inside an array for performance reasons because it would not need to compute a
label for the whole array in each assignment; instead we assign the label of the new value to assignee without any extra computations for the new label. Of course, this results in fine-granularity of access control.

Primitive data values can be declared either as an instance field or a class field or a local variable inside a method. For attaching security context, the instance fields of a class, we add an array of security context to object representation that will hold security context of corresponding field in the object. For class fields, we can declare an array of security contexts in the structure representing that class. For local variables, it is different. Local variables are stored in an array created on each method call. The structure of array elements devoted to primitive type local variables can be extended to have a combination of value and security context.

Neutrality of object specification from any specific representation makes it possible to assign labels to values transparently.

5.3 Determining domain for running threads

Java programming language supports multithreading at the language level. In distributed applications, these threads can be running on behalf of a remote user. Access control decisions are taken based on the clearance of these threads.

In SELinux terminology, security labels attached to dynamic entities such as processes are called domain. So, we should determine thread’s domain for different threads. One simple approach for determining domain is to let developer assign a domain to each thread. This domain can be provided by adding a new constructor to the Thread class in Java API to have an extra parameter that is the domain of new thread. But we claim that our solution is transparent and does not require any changes to the standard Java language. So we should look for another solution to maintain transparency and Java standard.

We go back to where we talked about socket channels. Remote user commands are received from the socket devoted for it. So we can assume every thread that tries to read from a socket, is the agent that is running on behalf of that user, so it must get the domain of the remote user. To do so, we use libselinux in associated with IPsec [15]. If clients and servers run on platforms that support IPsec protocol and an association has been set up, the server side can have the security context of the client side by calling getpeercon(3) from libselinux. In this way, we can get the domain of remote user and attach that domain to the thread that reads from the socket created for that user. Since now, each action done by this thread is assumed to be done on behalf of that remote user and access control decisions are taken based on that.

5.4 Information flow tracking

In order to track information flow inside JVM, we first need to have a vision of its runtime data structures. At runtime, JVM creates a stack for each thread. Each method call by a thread results in a frame pushed onto the stack. Each frame has three parts including: local variables array, operand stack and frame data. Local variables array is an array of words used for storing method parameters and local variables. Operand stack is used for storing instruction’s operands and intermediate results. Data types stored in operand stack and local variables array are the same. Execution engine is responsible for moving data between operand stack and the local variable array based on running bytecode.

To track information flow, the behavior of execution engine should be modified to carry the security context of information besides information themselves. Each slot of the operand stack and the local variable array should be modified to contain the security context of information residing at that slot.

Execution engine takes each bytecode of the loaded classes and interprets it. Bytecodes, based on their functionality, can be divided into ten groups: stack and local variable operations, type conversions, integer arithmetic, logic, floating point arithmetic, object and arrays, control flow, exceptions, finally clauses and method invocation. Each category contains bytecodes that most of them involve stack in their interpretation. Interpretation of each of these instructions by the modified execution engine is changed to reflect information flow.

For operand stack and local variable operations, since there is just a simple movement of information, we simply move their security contexts from stack to local variable and vice versa. This can be generalized to unary instructions in all categories. For binary operations, the scenario of label computation is the same as computing the labels for expressions described before. Having modified the interpretation of all bytecodes in the execution engine, we can assure information flow control inside JVM.

5.5 Access Control

Up to this point, information was entered into JVM memory space with their security contexts propagated inside programs, but no policies were enforced. To do so, we use the SELinux reference monitor as the source of decision makings and insert hooks in appropriate locations inside JVM to consult with SELinux security server about access decision. This is done by calling avc_has_perm(3) in libselinux with appropriate parameters.

There are two types of threats that should be avoided. Consider the case when confidential information is read, now the program may mistakenly try to send this information to a channel where it is observable to unauthorized users or an unauthorized user tries to read this information and send it to its privileged channel.

In the former, we have to control the flow of information from JVM to output channels. Given that we have had determined different types of channels, we insert hooks at output channels just before writing, to control the flow of information to these channels. In the latter case, we have to control access to resources inside JVM. We define access as a read operation and let the write operation to take place without any control. That is because after writing a new value, the old value is lost forever and the variable containing it gets the label of the new value. Two types of read accesses need to be intercepted: reading an array element, reading
a field from an object. This is because these two actions result in accessing the shared memory space among all threads, that is the heap memory space. For each one of these read operations, we insert a hook in the body of its interpreting code for the execution engine to control access before access can take place.

5.6 Implicit information flow
After proposing our MAC framework, we show how our approach prevents from implicit information flows. Information flows inside a program either explicitly or implicitly. An assignment of the form \( \text{LHS} = \text{expression} \) results in an explicit flow of information from expression into \( \text{LHS} \). Implicit information flow results from control flow of the program. Consider the following code snippet:

\[
\begin{align*}
\text{pub1} &= \text{true} \\
\text{pub2} &= \text{true} \\
\text{if} (\text{secret}) \\
\text{else} \\
\text{pub1} &= \text{false} \\
\text{pub2} &= \text{false}
\end{align*}
\]

Here, while there is not any explicit assignment, information can still flow implicitly in two ways: by following the branch and by observing the non-execution of a branch. In previous code, if we assume \(\text{secret} \) is true, one can find out the value of secret after the if condition statement completes by reading the value of \(\text{pub1}\). Also, by observing non-execution of a branch, that is the value of \(\text{pub2}\), the secret value can be deduced as well.

The inherent problem with dynamic analysis is in its inability to recognize information flows done in non-execution of branches. Therefore hybrid techniques have been suggested that perform a static analysis on code for recognizing all execution paths of a program, doing necessary annotations that runtime system may use for tracking information flow.

We suggest leveraging access control for contributing to information flow tracking. Each access to a program variable is done by a thread on behalf of a user. We showed that each thread has got a security domain corresponding to the remote user on whose behalf is running. By having each thread’s domain, we can control every access to the program data objects. In this way, no implicit flow and no label creep can occur, because every access is controlled and if the running thread is not privileged to read data contents, it is prevented from continuing execution. In previous code, if running thread is not privileged to read the secret value, the if condition expression is not evaluated and the normal control flow is stopped and the value of secret cannot be deduced through reading \(\text{pub1}\) or \(\text{pub2}\). In other words, we prevent each user from trying to access confidential information that is not allowed to read.

6. EVALUATION
We have implemented our solution in an open source JVM called SableVM under license of GNU GPL resulting in a secure JVM, we call SEJVM. To evaluate performance overhead of our solution, resulting from information flow tracking, computing dynamic labels and access control, we used the JavaGrande version 2.0 Benchmark Suite. We ran the benchmarks on a Core 2 Duo processor at a clock frequency of 2 GHz with 3 Gigabytes of memory, running a Linux 2.6.18 kernel. Results show relative slowdowns of SEJVM against normal JVM.

It is notable that among benchmarks, those which don’t have any interaction with OS, skip access control phase and therefore do not entail any access control overhead. This is what happens for Section 1 benchmarks. This is because, as far as there is no I/O operation with OS, no confidential data or labeled data exists inside JVM. At the access points, as the hooks intercept access in order to control it, they find data with no label and let access to be performed without control. It is assumed in our implementation that data with no label is not confidential and need no access control.

On the other hand, for confidential data that is passed from OS into JVM, most of them share a common security context and access decisions taken for them are the same. As an example, consider a file’s content, read in different steps by a single thread and manipulated inside JVM. Each access performed by this thread to manipulate this data, has a single access decision. To reduce overhead, we need to keep a table of access decisions in userspace to prevent from system calls per access decisions, because each system call results in a context switch that is too costly during program execution.

JavaGrande suite benchmarks consist of three Sections. Section 1 consists of basic operations of the Java programming language such as arithmetic, assignment, cast, method call, object creation, math functions and exception handling. Section 2 consists of computations frequently used in scientific applications. Section 3 consists of large scale applications that use large amounts of processing and I/O.

Figure 4 shows the results of Section 1 benchmarks. In this figure, column of assignments has a factor of about 2 relative to its normal form. This is because in every assignment, each right hand side expression to be evaluated requires access to each of the variables engaged in computing the right hand side of the expression. Each of these accesses is intercepted by hooks put in access points and controlled by consulting with SELinux.

Arithmetic column also shows an overhead due to the computation of the right hand side expressions. Evaluating an arithmetic expression requires access to variables for computing intermediate results incurring overhead for access control. While, in this case no access control occurs because of what we said about non-confidential data, our hooks intercept access to variables and control their labels which leads to an overhead.

![Figure 4. Section 1 of JavaGrande, relative slowdowns of SEJVM to unmodified JVM](image)
Other operations containing implicit assignments have low overhead. For example, method calls has almost no overhead. This is because we let information flow from variables into each other without any control over them as far as their labels are associated with them.

Section 2 represent overhead for routine applied in scientific application. Section 3 contains applications which are a mix of computational and I/O bound operations. Figures 5 and 6 report the results of slowdowns computed for them relative to their normal execution.

Overhead computed for them is a penalty paid for having a secure platform and thereby a system-wide security guarantee. Therefore, it can be tolerable in situations where these criteria are of higher priority than efficiency of running programs.

7. CONCLUSION
A system-wide security mechanism is not provided unless each part of the system ensures security during its lifespan. While operating systems as the basic building block of a system have provided secure platforms for users, information confidentiality is not assured as information is passed to user-space through programs running on this platform. We suggested a new approach for transparent tracking of information flow at programming language level thereby providing a system-wide and end-to-end security model.

Our solution is integrated with the underlying system security mechanism resulting in consistent information flow in kernel and user spaces.

Our solution did not require any changes to Java syntax. SEJVM is the result of implementing this solution. In SEJVM, input channels are modified to associate confidentiality polices with information as information flows from OS into JVM. Execution engine is manipulated such that it tracks information flow at runtime.

To prevent implicit flows, we have inserted hooks at appropriate locations that intercept each access request to information and controls them based on confidentiality policies attached to data. Through SEJVM, we can ensure SELinux about keeping confidentiality of its information at user space.

On the other side, it relieves application developers from controlling access to confidential information in development phase.

8. REFERENCES


