Mandatory Access Control at the Object Level in the Java Virtual Machine

Vivek Haldar Michael Franz
University of California
Irvine, CA 92697
+1-949-824-7308
{vhaldar, franz}@uci.edu

ABSTRACT

For decades, secure operating systems have incorporated mandatory access control (MAC) techniques. Surprisingly, mobile-code platforms such as the Java Virtual Machine (JVM) and the .NET Common Language Runtime (CLR) have largely ignored these advances and have implemented a far weaker security that does not reliably track ownership and access permissions for individual data items. We have implemented a system that adds MAC to an existing JVM at the granularity of objects. Our system maintains a strict separation between mechanism and policy, thereby allowing a wide range of policies to be enforced. In preliminary benchmarks, we find that the run-time overhead of tracking MAC tags for every object is around 30%.

1.INTRODUCTION

Three important trends have been recently emerging in the field of trustworthy computing.

At the application level, more and more code is now being targeted at high-level language runtimes and virtual machines that execute some form of safe, platform-independent bytecode. The most prevalent examples of this are the Java virtual machine[1], and the more recent .NET common language runtime[2]. Such code platforms offer a number of advantages over native code. The virtual machine performs a number of static and dynamic checks to ensure a basic level of code safety – type-safety, and control flow safety. Type safety ensures that operators and functions are applied only to operands and arguments of the correct types. A special case of type safety is memory safety, which prevents reading and writing to illegal memory locations – for example, beyond the bounds of an array – and thereby also provides separation between different processes without the need for hardware-based memory management. Control flow safety prevents arbitrary jumps in code (say, into the middle of an instruction, or to an unauthorized routine). These basic properties of safe code are enforced by a combination of static (e.g. bytecode verification) and dynamic (e.g. array bounds checks) techniques. Thus, safe code does away with a major source of bugs and vulnerabilities in current systems that stem from unsafe memory operations in C – such as buffer overruns and format string attacks.

At the operating system level, there has been a dire need of mechanisms to enforce information flow, confidentiality and integrity. Mandatory access controls, thought only to be relevant to (and originally designed for military systems[3], are gradually beginning to be incorporated into commodity open operating systems such as BSD and Linux. Projects like TrustedBSD[4] and Security-enhanced Linux[5] (SELinux) add techniques and tools to specify, manage and enforce a range of mandatory access controls. As opposed to discretionary access controls that rely on users to specify a security policy, and also do not control access throughout an object's lifetime, *mandatory access controls* rely on centrally administered policies that are imposed on every data item in the system throughout its lifetime. Examples of systems that need strict information flow controls are

payment processing e-business applications, medical data applications, as well as upcoming utility computing "grids", where computational resources are remotely "rented" out.

Control of sensitive information is also becoming increasingly important on personal computers. This is highlighted by the wide spread of various spyware programs that, in addition to whatever it is they purport do to, also have backdoors and surreptitiously leak information from the computer to remote sites.

At the hardware level and system software level, the Trusted Computing initiative (headed by the Trusted Computing Group [6]) is working towards introducing a hardware "root of trust" into commodity computing devices such a PCs and PDAs. The solution depends on a combination of minimal trusted hardware and a system software stack that takes advantage of it. The trusted hardware, also called the trusted module, can store secrets and check system integrity. This is combined with a layer of trusted system software that can use the module's functionality to check and enforce the integrity and confidentiality of the rest of the system.

These three trends together highlight the increased need for security on commodity personal computers, and in particular, a shift away from discretionary user-controlled mechanisms to stricter, centrally controlled mandatory mechanisms.

The Java virtual machine is widely used as a portable and safe development target, but its current security mechanisms are geared towards discretionary controls. These are typically imposed by a user to limit access to system resources by the execution of untrusted code in a Java Virtual Machine. Currently, the JVM lacks mechanisms to either specify or enforce information flow constraints.

Information flow is one of the most important security-relevant program properties. Programs can leak information in many and subtle ways. The most obvious way is to publicly display or communicate confidential information. Less direct methods use control flow that depends on confidential data. Even subtler are covert channels that convey information through indirect sideeffects of program execution, such as CPU usage, or energy and voltage fluctuations. Even highly inspected software may have subtle information leaks.

Current systems suffer from a dearth of mechanisms to ensure the confidentiality of sensitive data. In most cases, the software vendor is simply trusted to have provided well-behaved software. Due the highly complex and dynamic nature of today's system software, this is far from adequate. What is needed is an automated method to either enforce or explicitly prove the absence of information leaks.

As a solution, we have extended the JVM with functionality to do mandatory access control at the granularity of objects. Our implementation strictly separates the enforcement mechanism from the specification of polices. This allows flexible specification and enforcement of a wide range of policies. Moreover, we show that these techniques are implementable in current JVMs with minimal modifications to other JVM subsystems, while maintaining full backwards compatibility.

We have implemented this by adding an *access control tag* to each object, and modifying the virtual machine to check that tag at every data access to an object. Policies will take the form of *predicates* over these access control tags. Since mechanism and policy are strictly separated, various policies can be *plugged in* to the VM.

A key advantage of keeping mandatory controls in the virtual machine is that it is completely transparent to programs being run in it. No access to source is needed, and the bytecode format does not need to be changed. The binding of code and policy happens at runtime, when mandatory access tags are assigned to objects.

The novel contributions of this paper are twofold: to explain the need for mandatory access controls in the Java virtual machine, and present its design and implementation. We also evaluate and

discuss the impact of introducing this new access control mechanism into the JVM. Finally, we compare our scheme with existing access control techniques for Java, and discuss the advantages and disadvantages of each.

The rest of this paper is structured as follows: Section 2 gives an overview and evaluation of present techniques for access control and information flow in Java, both at the language as well as virtual machine level; Section 3 presents the design and implementation of mandatory access control in a Java virtual machine – this forms the core of the paper; Section 4 evaluates our design against a number of well-known security principles; Section 5 briefly explains the context in which this work was done – semantic remote attestation; Section 6 presents related work in access control; Section 7 discusses avenues for future work; and Section 8 concludes.

2.EXISTING APPROACHES

Early Java implementations (up to JDK 1.1) had two distinct security environments. The first environment, a complete sandbox, was meant to constrain the execution of applets downloaded from the Web. These applets were considered completely untrusted. The sandbox disallowed any access to the local filesystem, as well as any network connections to domains other than the one from which the applet originated. This sandbox policy was designed to prevent untrusted code from leaking local data, and consuming too many network resources. The second environment had no constraints at all, and was used to run local code on a machine. Code on the local disk was considered completely trusted. Thus, this early model was essentially all-or-nothing, accounting for either completely untrusted, or completely trusted code, with no gradations in between.

Later versions of Java (after JDK 1.2) added capabilities to create more graded security environments, and provide a variety of more fine-grained security permissions[7]. Instead of being trusted (local), or untrusted (remote), code was now associated with principals. A public key infrastructure and cryptographic signatures were used to bind principals to code. A security policy specified what permissions code originating from various principals would get. Permissions included filesystem read and write permissions, and network socket capabilities. Enforcement was relegated to a runtime *security manager* that regulated access to privileged resources by looking up the permissions possessed by the object that made the request. For example, a policy may specify that all code digitally signed by the domain `uci.edu` is allowed to read any local file, but to write only under `/tmp`.

However, there are many useful security polices that the current Java architecture does not address. Higher level policies that depend on program state cannot be specified. An example of such a policy is “disallow transmitting on the network *after* reading from the local filesystem”. Inlined reference monitors[8] and software fault isolation[9] have been used to enforce policies such as this. But even those techniques cannot handle stronger policies such as: “any data read from the local filesystem must not be transmitted on the network”. Also, policy can only be specified in terms of permissions exposed by the Java security API. Another critical drawback is that once a security check is done, there are no controls on the propagation of data thereafter. Data confidentiality policies cannot be expressed or enforced in the current Java scheme. This is the reason why a policy such as “any data read from the local filesystem must not be transmitted on the network” cannot be currently expressed.

At the Java source level, fields and classes can be marked with *access modifiers* such a `public`, `private` and `protected` to limit their visibility to other classes and packages. While enforced offline by the Java compiler, marking a field private does not mean that it is inaccessible at runtime. Private fields can be easily accessed using Java's reflection capabilities. Thus, these modifiers should be thought of as an abstraction tool to hide implementation details, rather than for strict protection of information.

Some recent research has focused on statically enforcing information flow at the source level using language-based techniques. Various language-level techniques can be used to control information flow[10]. Type-based information flow relies on programmers inserting security label annotations into source code. Myers et al[11] used a type system to enforce information flow statically. Their Jif compiler is a source-to-source compiler that checks a Java program with information flow annotations, type-checks it, and outputs a regular Java program. These are then statically type-checked – successful type-checking implies the absence of illicit information flows.

Attempts to statically impose information flow on bytecode[12] suffer from serious shortcomings, such as the inability to handle dynamic object creation, and being forced to make overly conservative assumptions when doing inter-procedural analysis.

A fundamental shortcoming of static analysis is that it must work under a closed-world assumption. This means that the analysis must have access to the whole program, and that the program that finally gets executed must be exactly the same program that was analyzed. Any dynamic extensions to a program invalidate the assumptions used by the analysis. This runs counter to Java's model of dynamic class loading, which may occur at anytime during program execution. Another disadvantage of static methods is the *early binding* of policy and code. The policy one wants enforced must be known at compile time. This is suitable for well-known policies that rarely change. However, for policies not known a priori, or when the same program needs to be executed with difference policies, more dynamic methods are needed that allow *late binding* of policy and code.

3.MANDATORY ACCESS CONTROL ON OBJECTS

In this section we describe our design and implementation of mandatory access controls in a Java Virtual Machine.

The questions that must be asked when designing this are:

- What is the granularity and unit of data protection?
- How are access controls enforced?
- How are access controls specified?

In keeping with the principle of separation of mechanism and policy[13][14], our design keeps the third aspect disctinct from the first two. This keeps our mechanism from being biased towards a specific policy, and also allows a variety of policies to be enforced.

Both the Java language[15] and the Java Virtual Machine[1] are object-oriented. An object is both the fundamental level of abstraction at which a programmer thinks while writing Java code, as well the runtime data structure around which a Java virtual machine is built. Unlike atomic variables that contain a single data item which may be part of a larger logical collection of data, objects conveniently encapsulate one or more related data items and code into one abstraction. Thus, we consider *objects*[1] to be the unit of protection in our design.

Having fixed the unit of protection, the next question is: what enforcement mechanism should be used to protect it? To answer this, we must enumerate all the ways in which an object can be accessed, and interpose our mechanism between the access and the object. The interposed enforcer must then make a decision about whether to allow the access depending on the access control permissions of the object.

In the Java virtual machine, all computation and access to objects takes place using a set of high-level machine-independent bytecode instructions[1]. Thus, there is fairly narrow and well-defined interface through through which access to objects takes place[2]. We must now focus on the bytecodes that enable the transfer of information from one object to another. There are two classes of

[1] Sometimes also called *class instances*.

operations that do this: method calls, and reading and writing fields. Bytecodes to read and write fields directly modify data in other objects. Method calls result in indirect information flow, through parameters and return values.

Thus, to implement mandatory access control at the object level, we must interpose an enforcement mechanism between bytecodes that access objects and the objects being accessed. Bytecode instructions that read and write fields in objects are: `getfield, getstatic, putfield,` and `putstatic`. Bytecode instructions that invoke methods are: i`nvokevirtual, invokestatic,` and `invokespecial.` Our enforcement mechanism intercepts these bytecodes, and then based on the labels of the source (the one that executed the bytecode) and target (the one that is being accessed) objects, decides whether to allow the access.

Policy is encapsulated by a *decision function*, that given the source and target object, and kind of access (read or write), decided whether to allow the access. Our mechanism uses this policy decision function as a blackbox, and it is completely independent of it. This gives us considerable flexibility in using various mandatory access control policies.

For the initial specification of tags, we rely on a *label function*, that, given an object, assigns it a mandatory access control label. The label function is invoked at object creation time. Currently, this label function is simply a program, but in future work we would like to use a specification language to map objects to labels.

The decision and label function together make up a specific policy.

3.1 Implementation

We have implemented a prototype of our design in the KVM, a Java virtual machine designed for small resource-constrained devices. It is a pure interpreter, written in C. We chose the KVM as our base because of its clean and small implementation.

We modified the class data structure to add a mandatory access control tag to every object. The particular layout of this tag is left unspecified. It is created by the label function, and interpreted by the decision function.

The bytecodes that can access objects are: `getfield, getstatic, putfield, putstatic, invokevirtual, invokestatic,` and `invokespecial.` We have modified the implementation of these bytecodes to call the decision function before being allowed to execute. When access is denied a runtime Java exception is raised.

To measure the overhead imposed by mandatory controls, we created a few micro-benchmarks that run the access-sensitive bytecodes mentioned above in a tight loop. We used a simple multi-level lattice policy for our experiment. The measurements were performed on an Intel Pentium-M 1.3Ghz system with 384 MB of RAM, running Linux 2.6.3. We measured an overhead of approximately 30%. Note that this is the worst-case overhead. In real applications access-sensitive bytecodes are typically less frequent, and so we expect the overhead there to be lower.

4.EVALUATION

In this section we evaluate our design. The criteria used for evaluation are based on those from the seminal paper by Saltzer and Shroeder[17][3].

Economy of mechanism: the design should be as simple and small as possible. Our mechanism involves interposing access checks before the execution of a small number of bytecodes out of the

[2] there is the potential for using some of theese mechanisms in a virtual machine as covert channels – we discusss this further in section 4.2

[3] Inspired by [16], where the authors also measure their techniques against Saltzer and Shroeder's criteria.

complete instruction set. No other JVM subsystems (garbage collection, threads etc) are affected by the introduction of this new access control mechanism.

Fail-safe defaults: The default decisions should be conservative and prevent access. This is the task of the policy specifier. Our mechanism does not make this decision. However, without mandatory access controls, we simply fall back to Java's default security mechanisms.

Complete mediation: Every access to every object must be checked. Once again, working at the level of platform independent bytecode helps us in this regard. All computation and access in a virtual machine can only be done through the execution of a well-defined set of bytecode instructions. Thus, we can easily mediate every access by interposing our protection mechanism into the execution of the relevant bytecode instructions.

Least common mechanism: Minimize the amount of mechanism common to more than one user so that potential information paths between users do not compromise security. In our design, there is no mechanism at all that is common to more than one user. The mechanism is implemented in one place – the virtual machine – and every user uses the same mechanism.

Accountability: Record accesses so that they can be referred to in case of a breach. Our system does not perform logging by itself. However, logging of access checks could easily be implemented as part of the policy decision function.

C*ompatibility: How does the security mechanism impact backward compatibility?* Our mechanism is completely transparent to programs being run in the virtual machine. Implementing MAC in the VM does not need access to application source to enforce controls, nor does it it need any changes to the Java bytecode format.

Remote calls: How does the mechanism extend to remote calls? This work was done in the context of making remote attestation, one of the core techniques of Trusted Computing, more flexible. We explain below, in Section 5, how we use our technique for remote enforcement of security policies.

4.1Taint propagation in Java

The scripting language Perl[18] provides a useful mechanism for safe execution of programs that process potentially malicious user input. The goal is to prevent user data from being used in sensitive system operations. For example, what if data input in a web form is passed on to the system shell for execution, or as parameters to system calls? To prevent such attacks, Perl's *taint mode* tracks all user input at runtime, and prevents this data from being used as parameters to a predefined set of sensitive system functions. Data must be explicitly *untainted* (by filtering it through a regular expression) before it is allowed to be passed into such functions.

The generic MAC framework that we have described here can be used to perform taint checking in the JVM. By using a multi-level security lattice, and then defining a policy that tags all sensitive system classes (such a `java.lang.System`) as "low", and all other classes as "high", we can ensure that no unchecked data is passed to system classes as parameters. Such a feature would make it easier to quickly deploy untrusted bytecode on a Java server. This could also be used as a program debugging tool to see the flow of information in a Java program, much like "electric fences" are used to detect memory leaks in C.

4.2Covert channels in a virtual machine

Explicit channels for the transfer of information, such as as assignments or method calls, can be controlled by changing or monitoring the mechanisms that implement them. However, information can also be transmitted through *covert channels* that do not depend on explicit mechanisms, but the side-effects of computation[19]. Examples are:

- Timing channels: measuring how long a computation took can reveal something about the data it was operating on[20]. A subset of timing channels are termination channels, where the termination of a program reveals information[21].
- Power channels: measuring the power consumption of a CPU (or a peripheral, such as a smart card) can be used to infer the bits being computed.
- Resource channels: information could be leaked by monitoring the consumption (or exhaustion) or various resources, such as a CPU or memory.

A full-fledged Java virtual machine has many potential covert channels. Examples include: how often, when and how long the garbage collector runs. For example, code could be crafted to purposely trigger the garbage collector. Measuring the latency of garbage collection could reveal information about the size and number of objects. This is is an instance of using resource consumption as a covert channel.

While mandatory access controls can control the overt flow of information in a virtual machine, stemming the flow through covert channels remains an open question.

5.TRUSTED COMPUTING

This work was done in the context of a larger project to explore how language-based security techniques can be used to enhance Trusted Computing. In this section we give a brief overview of Trusted Computing, and our efforts to mitigate some of its shortcomings using language-based techniques, and how the work presented here fits into this broader effort.

Trusted Computing[6] is an effort to bring some of the properties of closed, proprietary systems to open, commodity systems. Trusted computing introduces mechanisms and components in both hardware and software that check and enforce the integrity of a system, and allow it to authenticate itself to remote systems. For example, a secure booting procedure makes sure that the operating system has not been tampered with. Using a chain of reasoning that starts from a trusted hardware module, we can arrive at a conclusion about the state of a system after boot-up. Similarly, we can deduce for sure what particular program is running on a system.

Remote attestation, one of the core features of a trusted computing infrastructure, is the process by which software authenticates itself to remote parties. This allows the remote party to make certain assumptions about the behavior of the software.

Before trusted computing can reach its full potential, questions such as the following need to be addressed:

- How do programs running on trusted platforms authenticate each other in a manner that ensures that each party satisfies some security criteria, while leaving room for various differing implementations?
- The current client-server network computing model assumes a trusted server, and untrusted (even malicious) clients. Thus, even though a significant fraction of work is done at the clients, all the trust resides at the server. How can we design new network protocols (or adapt existing ones) to work in an environment that allows a more flexible partitioning of trust?
- Moving away from the model of having a fully trusted server, and a fully untrusted client, how do we design models and applications that use them, that can broker trust in more flexible and dynamic ways than is possible today?

Standard ways of doing remote attestation are based purely on cryptography, and suffer from many critical shortcomings – they are static, inexpressive, inflexible and do not scale. Most importantly, they do not speak about program behavior – they can only attest to the presence of a particular bin-

ary. It is possible for an attested binary to have bugs and not obey the security policy a server was expecting it to. Remote attestation is hard to scale up to a flood of software patches and upgrades. It also does not accommodate a varied, homogeneous computing environment very well.

5.1Semantic Remote Attestation

The shortcomings of traditional ways of remote attestation can be traced back to one root cause – *what is desired is attestation of the behavior of software running on a remote machine, but what actually gets attested is the fact that a particular binary is being run.*

We are working on a technique called *semantic remote attestation* [22] that attempts to alleviate these shortcomings of standard remote attestation. The core idea behind our technique is to use a language-based virtual machine (a trusted virtual machine) that executes a form of platform-independent code. Software up to and including this virtual machine is trusted. However, the virtual machine can certify to remote parties various properties of code running under it by explicitly deriving or enforcing them. This can be done in many ways, such as observing the execution of programs running in a VM, or analyzing the code before execution. This is particularly easy to do with high-level platform-independent code that has a lot of information about the structure and properties of code.

The fact that trusted computing, and its core technique, standard remote attestation, can lock consumers into a particular program or platform has been a very widely expressed fear[23]. A key advantage of our approach is that reasoning about the behavior of a program is not tied to a particular binary. Semantic remote attestation checks for program properties, and works with different implementations of the same program as long as they satisfy the properties required of them.

Two techniques that a trusted virtual machine can use to certify properties of code running on it are: installation of a runtime monitor; and running various test suites. See [22] for details.

5.2Remote specification of information flow properties

Another technique that can be used by a TrustedVM is using mandatory access control on objects in a trusted virtual machine. The goal is to certify to remote parties communicating with a TrustedVM that the information they provide is being handled according to a policy also specified by them.

Consider a network exchange between some remote party and a TrustedVM that involves the exchange of some sensitive data. In such a scenario, the remote party would like to have some means of constraining how the information is handled by the TrustedVM. Taking advantage of MAC support in a VM, the remote party could specify a MAC policy for the TrustedVM to enforce.

6.RELATED WORK

In section 2 we reviewed existing access control approaches for Java and the Java virtual machine. Here we briefly survey broader related work in mandatory access control.

Early work in information flow and mandatory access control(MAC) was done by Bell and LaPadula[24], who pioneered the idea of information being classified at multiple sensitivity levels. Denning [25] extended the Bell-LaPaulda model to use a lattice for sensitivity labels. Denning was also one of the first to use static analysis on source code to enforce information flow properties with very little runtime overhead[26]. Volpano formalized the soundness of the analysis that Denning proposed[27].

7.FUTURE WORK

There are many avenues for future work that we would like to explore. Currently the task of assigning MAC labels to objects is delegated to a label function, which could use arbitrary logic to

do its task. However, what we would like is to have a *high-level policy specification language* that can express how to assign labels to objects.

In an operating system that supports MAC at the filesystem level, we could use MAC labels from the filesystem to imply MAC labels of objects. For example, a Java `File` object that read from a file with a particular label should automatically get the same label. This would also mitigate the privilege escalation problem, where a program that uses files of various classification levels must run at a level at least as high as the highest level among those objects. When mandatory access controls are extended into the application manipulating those objects, such as ours, then the same controls also apply inside the execution environment of the program. To start with, we would like to interface our virtual machine with mandatory access controls in operating systems such as TrustedBSD[4] and Security-enhanced Linux[5].

For conceptual simplicity and ease of implementation, we current system has been implemented in an interpreter. In the future, we would like to extend our implementation to a full-fledged just-in-time compiler for Java bytecode, such as the Jikes Research Virtual Machine. We are also currently working on implementing suitable macro-benchmarks to measure the performance impact of MAC in a JVM for real applications.

Trusted computing systems use *trusted paths* between input devices and applications or device drivers to prevent spoofing as well as eavesdropping. For example, a fully encrypted an authenticated channel is used between a password-prompt dialog and the application asking for it. We would like to implement corresponding functionality inside a virtual machine. Currently, the dynamic nature of the Java virtual machine makes is easy to do things like modify the class hierarchy, or use reflection to interpose wrappers around method calls – both at runtime. For example, dynamic method wrappers (also known as dynamic proxies) are frequently used to add a layer of logging around method calls. Such techniques could also be used to eavesdrop on the transfer of confidential data between objects. Implementing a trusted path mechanism for object communication would be a step towards solving this problem.

8.CONCLUSION

Current access control mechanisms for Java lack support for mandatory access controls, which are needed when strict information separation is needed, or when sensitive data is handled. Static approaches to controlling information flow cannot handle dynamic policies very well. While operating systems have supported mandatory access controls for a long time, virtual machines currently do not have any support for it.

In this paper, we have presented the design and implementation of mandatory access controls in a Java virtual machine. We chose an object to be the basic unit of protection. To guard access to objects, we interpose enforcement mechanisms into the interpretation of those bytecodes that can access objects by reading and writing fields, or invoking methods. The enforcement mechanism and specification of policy are kept strictly separate from each other.

Acknowledgments

The authors would like to thank Deepak Chandra for many discussions, and suggestions.

9.REFERENCES

[1] Tim Lindholm, Frank Yellin , The Java Virtual Machine Specification, 1999, Addison-Wesley

[2] David S. Platt, Introducing Microsoft .Net, 2003, Microsoft Press

[3] Department of Defense, Trusted Computer System Evaluation Criteria, DOD standard 5200.28-STD, December 1985

[4] Robert Watson, Wayne Morrison, Chris Vance, Brian Feldman, The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0, USENIX Annual Technical Conference, June 2003

[5] Stephen Smalley, Chris Vance, and Wayne Salamon, Implementing SELinux as a Linux Security Module, Technical Report, May 2002

[6] Trusted Computing Platform Alliance, TCPA PC Specific Implementation Specification, , September 2001

[7] Li Gong, Marianne Mueller, Hemma Prafullchandra, Roland Schemers, Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2, USENIX Symposium on Internet Technologies and Systems, December 1997

[8] Ulfar Erlingsson and Fred B. Schneider, SASI enforcement of security policies: A retrospective, New Security Paradigms Workshop, 2000

[9] Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham, Efficient software-based fault isolation, ACM Symposium on Operating Systems Principles, 1994

[10] Andrei Sabelfeld, Andrew C. Myers., Language-Based Information-Flow Security, IEEE Journal on Selected Areas in Communications, Special issue on Formal Methods for Security, 21 (5), January2003

[11] Andrew C. Myers, JFlow: Practical mostly-static information flow control, Principles of Programming Languages, 1999

[12] C. Bernardeschi, N. D. Francesco, and G. Lettieri, Using standard verifier to check secure information flow in java bytecode, Computer Software and Applications Conference, 2002

[13] R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf, Policy/mechanism separation in Hydra, Fifth ACM Symposium on Operating systems principles, 1975

[14] Robert Grimm , Brian N. Bershad, Separating access control policy, enforcement, and functionality in extensible systems, ACM Transactions on Computer Systems, (), February2001

[15] Bill Joy, Guy Steele, James Gosling, Gilad Bracha, Java Language Specification, 2000, Addison-Wesley

[16] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten, Extensible Security Architectures for Java, 16th Symposium on Operating Systems Principles, October 1997

[17] Jerome H. Saltzer and Michael D. Schroeder, The Protection of Information in Computer Systems, Proceedings of the IEEE, 63(9), 1975

[18] Larry Wall, Tom Christiansen, Jon Orwant , Programming Perl, 2000, O'Reilly & Associates

[19] B. W. Lampson, A note on the confinement problem, Communications of the ACM, 16(10), October1973

[20] J. Agat, Transforming out timing leaks, ACM Symposium on Principles of Programming Languages, January 2000

[21] D. Volpano and G. Smith, Eliminating covert flows with minimum typings, IEEE Computer Security Foundations Workshop, June 1997

[22] Vivek Haldar, Deepak Chandra, and Michael Franz, Semantic Remote attestation: A Virtual Machine Directed Approach to Trusted Computing, USENIX Virtual Machine Research and Technology Symposium, May 2004

[23] Ross Anderson, Cryptography and Competition Policy - Issues with Trusted Computing, 2nd Annual Workshop on Economics and Information Security, May 2003

[24] D. Bell and L. LaPadula, Secure computer systems: mathematical foundations, Report MTR 2547 v2, MITRE, November 1973

[25] D. E. Denning, The lattice model of secure information flow, Communications of the ACM, 19(5), 1976

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

[27] D. Volpano, G. Smith, and C. Irvine, A sound type system for secure flow analysis, Journal of Computer Security, 4(3), 1996