Practical, Dynamic Information-flow for Virtual Machines

Vivek Haldar, Deepak Chandra, and Michael Franz

University of California, Irvine, CA 92617, USA,
{vhaldar, dchandra, mfranz}@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. Moreover, our implementation is independent of any specific JVM, and will work with any JVM that supports the JVM Tools Interface.

1 Motivation

High-level language runtimes and virtual machines are becoming increasingly popular platforms for development. More and more code is now being targeted at these language runtimes that execute some form of safe, platform-independent bytecode. The most prevalent examples of this are the Java Virtual Machine (JVM) [12], and the more recent .NET Common Language Runtime (CLR) [14]. Such code platforms offer several 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 [4]. Control flow safety prevents arbitrary jumps in code (say, into the middle of a procedure, 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 errors and vulnerabilities in current systems that stem from unsafe memory operations in C—such as buffer overruns and format string attacks.

While virtual machines provide a portable and safe development target, their current security mechanisms are geared towards discretionary controls only. These are imposed to limit access to critical system resources. However, once access to a resource is gained, there are no mechanisms to track the usage and ensure that it respects the security constraints of the system. Essentially, the entity is *trusted* to handle the resource

properly. As a system gets large and complex, it becomes increasingly hard to ensure its trustworthiness. It may be breached because of an unintentional programming error or because of malicious code camouflaged as safe code.

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.

To further motivate our approach, consider the following scenarios:

Scenario 1—Downloaded program manipulating sensitive data: Consider a Java applet from an untrusted (or partially trusted) site that computes tax returns. The program has access to private information such as Social Security Numbers (SSN) and salary information, and it needs to communicate with its "home base" to consult tax tables and to charge the user's credit card for the service. How can we be sure that it isn't also leaking the user's SSN, salary, and bank account information? Clearly, in current Java systems, we cannot.

This scenario highlights a very common problem faced by users today: they use programs that manipulate various sensitive information items on their behalf, and yet, they are given no mechanisms to control how these programs handle their data. They essentially have to trust the program to behave correctly, and not leak secrets to untrusted sources.

Scenario 2—Java program handling sensitive databases: Consider a Java program connecting to two databases, one of which contains sensitive information, and another that contains public information. Even though data from the sensitive database is probably marked as such, once it has been read into a Java program, this meta-information is lost and becomes un-enforceable. Nothing prevents the program from reading rows from the sensitive database, and writing them into the public database. In general, detecting "channels" in Java programs in the current situation requires auditing of source code.

Scenario 3—Application server handling sensitive user input: Consider a web application server that presents a web form for user input. Some of the information is sensitive and hence sent over the wire using an SSL connection. But at both ends, there is no distinction between the sensitive information so secured and the rest of the data—both in the server and the client's browser, this information lives side by side and is potentially vulnerable to programming errors or malicious code. Adding mandatory access controls to the JVM can provide labeling of sensitive data, separation of such data from non-sensitive data, and for example, even enforce rules that such information must be transported using SSL connections.

Mandatory Access Control has been studied in the context of operating systems. Security-conscious environments such as the military and the government have have been using strict MAC mechanisms in secure installations [7] for decades. Recently, mandatory access controls are beginning to be incorporated into commodity open operating systems such as BSD and Linux. Projects like TrustedBSD [20] and Security-

enhanced Linux [17] (SELinux) add techniques and tools to specify, manage and enforce a range of mandatory access controls. However, while mandatory access controls are becoming increasingly common in underlying operating systems, language runtimes like the Java Virtual Machine lack mechanisms to either specify or enforce information flow constraints. This has created a *semantic gap* between the access models of the operating system and those of the language runtime.

As a solution, we have extended the Java virtual machine with functionality to perform 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, our technique is implemented in a *VM-independent manner*, We did need to make some modifications to the system libraries, but these are fully backwards compatible.

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 current techniques for access control and information flow in Java, both at the language as well as virtual machine level, discusses some of their shortcomings, and motivates the need for mandatory access control in the virtual machine; Section 3 presents the design rationale for mandatory access control in a Java virtual machine, using a couple of simple examples; Section 4 details our implementation and results; Section 5 discusses open issues and future work; Section 6 presents additional related work and Section 7 concludes.

2 Existing Approaches

Early Java implementations (up to JDK 1.1) had two distinct security environments. The first environment, a complete sandbox, was designed 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. It had no gradations between these two extremes.

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 [9]. 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 permis-

sions, 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 "do not allow transmitting on the network after reading from the local filesystem". Inlined reference monitors [8] and software fault isolation [19] have been used to enforce policies such as this. But even those techniques cannot handle stronger policies that track information within a program. An example of such a policy is: "any data read from the local filesystem must not be transmitted on the network". Note that this is a finer-grained policy than the earlier one because it permits sending on the network even after a local file has been read—it merely forbids sending information that was actually read from the file.

Another shortcoming of the standard Java security architecture is that policies 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 currently be expressed.

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 [16]. Type-based information flow relies on programmers inserting security label annotations into source code. Myers et al. [15] use a type system to enforce information flow statically. Attempts to statically impose information flow on bytecode [3] suffer from serious shortcomings, such as the inability to handle dynamic object creation, and being forced to make overly conservative assumptions when performing 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 to be 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. Static methods also need access to source code, which is only rarely the case in most installations. The more frequent case is that only binaries or compiled bytecode are present.

Adding mandatory access control to the JVM cleanly sidesteps these problems. Dynamically enforcing MAC policies in a JVM has the following advantages:

– Since enforcement is *dynamic*, policies can be *late-bound* to code, and can even change dynamically. MAC allows the tagging of specific data items for the lifetime of program execution. The binding of code and policy happens at runtime, when mandatory access tags are assigned to objects.
– The *separation of mechanism and policy* gives great freedom in expressing a variety of MAC policies.
– 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.* Thus our proposed enhancement is completely backward compatible with the large existing base of Java bytecode.
– Adding MAC to the JVM also bridges the gap in access control models between military-grade operating systems that have long had support for MAC policies, and applications written in virtual machines that still rely on discretionary controls. Applications running in a JVM cannot make full use of OS-level MAC classifications. Adding MAC to the JVM will allow a more seamless inter-operation between OS-level and program-level access control for data items.

3 Solution: Mandatory Access Control on Objects

We will illustrate the key concepts of our approach with a simple running example.

Consider a Java class, `SecretProcessor`, that reads in a sensitive local file to process it, and then attempts to write the data it has read in to a new file in a publicly-viewable folder.

The following is pseudo-code for `SecretProcessor`.

```
class SecretProcessor {
  void processSecret() {
    FileReader inSecret = new
                FileReader(secretFile);
    FileWriter outPublic = new
                FileWriter(publiclyViewableFile);
    // this should not be allowed!
    outPublic.write( inSecret.read() );
  }
}
```

The policy we want to enforce on this program is that data read from sensitive files should be prevented from being written to publicly viewable files.

To do that, we need to address the following issues:

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

To separate mechanism and policy [10], our design keeps the third aspect distinct 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 [11] and the Java Virtual Machine [13] 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 logically related data items and code into a single abstraction. Thus, we consider *objects* to be the unit of protection in our design. Hence, access control tags are associated with objects.

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 [13]. Thus, there is fairly narrow and well-defined interface through through which access to objects takes place. We now focus on the byte-codes that enable the transfer of information from one object to another. There are two classes of 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. For example, in `SecretProcessor`, the `read()` method call returns secret data.

We need to specify how an object is initially assigned a tag, and then, how tags are propagated at runtime. For our example, we deduce initial tags from a simple mapping between file locations and their sensitivity. So, for instance, a `File` object for a file in folder `Secret` is marked "sensitive". Similarly, `File` objects for files in a folder called, say `Public`, are marked "public". This policy must be specified by the user.

Once an object gets tagged "secret", any other object that reads from it (using a field access, or a method invocation) must also inherit this tag. Ultimately, all objects that have read sensitive data will get tagged "secret".

For the final step, for an output channel (folders, in this case), we need to specify what level of data it is permissible to output on that channel. For our example, data tagged "secret" cannot be written to the `Public` folder.

Now consider a slightly modified version of the same `SecretProcessor` class. This time, the `processSecret` method first writes public data to another public file, and then later writes secret data to a publicly-viewable file. The first write should be allowed, while the second should be blocked.

```
class SecretProcessor {
  void processSecret() {
    FileReader inSecret = new
                FileReader(secretFile);
    FileReader inPublic = new
                FileReader(publicFile);
```

```
    FileWriter outSecret = new
                FileWriter(anotherSecretFile);
    FileWriter outPublic = new
                FileWriter(publiclyViewableFile);
    // this should be allowed
    outPublic.write( inPublic.read() );
    // this should also be allowed
    outSecret.write( inSecret.read() );
    // this should NOT be allowed!
    outPublic.write( inSecret.read() );
  }
}
```

Such a policy is enforceable using runtime MAC tags associated with objects. In this case, an instance of `SecretProcessor` is not marked "secret" until it actually reads secret data (using the call to `inSecret.read()`). Hence, the first write is allowed, since the class is not tagged "secret" yet. However, after reading secret data (using the call to `inSecret.read()`, the object gets tagged "secret", and is henceforth forbidden from writing to public channels.

This example demonstrates how using runtime MAC tags on objects can support fine-grained policies. This is in stark contrast to Java's existing security mechanisms, based on *permissions*. An entire Java program runs under a policy, which is a list of permissions. For example, a `FilePermission` grants read or write permissions to certain sets of files. Similarly, a `SocketPermission` allows the program to connect to a certain host on a certain socket. But note that the permissions are imposed on the *entire program*. So, for example, a program can either be allowed to read secret data, or not at all. Policies that explicitly track data cannot be expressed. The policy of the last example, which was "allow reading both secret and public data, but do not allow secrets to be written to public files" cannot be expressed using Java's existing permission mechanisms.

4 Implementation and Results

We have implemented our scheme as a plug-in that will work with any Java 1.5 compliant virtual machine. We make extensive use of the JVM Tools Interface API (JVMTI) that "provides both a way to inspect the state and to control the execution of applications running in the Java virtual machine (JVM). JVM TI supports the full breadth of tools that need access to JVM state" [1]. Note that our implementation is *independent of any specific JVM*, and will work with any JVM that supports the JVMTI API.

Our JVMTI MAC plug-in does the following:

- intercepts field accesses and writes.
- instruments class files at load time to intercept entries and exits of certain methods.
- instruments the constructor of `java.lang.Object` so that we can intercept object creation.

Field accesses and writes, and method entries and exits need to be intercepted because those are the two means through which data passes between objects. We also need to intercept object creation to assign objects their initial tags. JVMTI provides event-based notifications for field accesses and writes. However, bytecode rewriting is a more efficient way to intercept method entries and exits. It allows us to selectively instrument methods of relevant classes.

An access control tag is associated with every object in the virtual machine. A tag is a 64-bit long value.

The policy is specified by the following:

- When to allow field reads
- When to allow field writes
- When to allow method calls
- How to propagate tags when any of the above three happens

Each of these is a predicate over the tags of the two objects involved. For example, whether an object `a` can access a field of another object `b` will be decided by evaluating `canReadField(tag(a), tag(b))`. Currently, this policy is specified by writing code that evaluates these predicates. These policy "callbacks" are invoked by our mechanism to enforce a concrete policy. This clean separation between mechanism and policy gives us great freedom to use a great variety of policies. Note that this dynamic mechanism allows us to change policies between separate runs of the same program—something not allowed by static type-checking mechanisms.

To get an estimate of the overhead of adding MAC-tags to objects, we measured its overhead for a some simple microbenchmarks. The microbenchmark is essentially the example described in section 2, that does file I/O (reads from one file, and writes data to another) in a tight loop. We measured the overhead for both buffered (with 4KB buffers) and non-buffered (a byte at a time) reads and writes. All measurements were done on a Pentium IV 1.7 GHz machine with 1 GB of RAM, running Windows XP, and JDK 1.5 from Sun. The measured the slowdown compared to running the same program on a JVM without MAC-support. The slowdown for the non-buffered case was a factor of 176, and for the buffered case was 121.2. Currently, this slowdown renders our system unusable for real programs. However, our goal was to first explore the idea of using MAC at the object level rather than optimize for performance. We are exploring alternative implementation methods, such as bytecode instrumentation, to make this overhead lower.

5 Discussion and Future Work

There are several avenues for future work. The most immediate need in our current system is for a *policy specification language*. Currently, the policy is simply written out as code that is called-back from our implementation. In the long run, this is error-prone and unportable. We would like to design a policy specification language that can succinctly capture a wide range of policies for MAC at the object level.

Another area we would like to investigate is whether the unit of protection can be meaningfully made finer than an object. The disadvantage of having objects as the

unit of protection is that we lose precision when an object mixes data of two levels, e.g. “secret” and “public”. In that case, an object that has is tagged “secret” cannot release “public” data. This can be addressed by not treating an object as a single unit, but rather, performing more fine-grained access control on it’s fields and variables as well. It is an open question whether this finer granularity will be worth the overhead for real programs, or whether objects, even though coarser-grained, are a sufficient level of granularity.

Another area of future work is interfacing our MAC-enabled JVM with operating systems that support MAC. 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 [20] and Security-Enhanced Linux [17].

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 performed by Bell and LaPadula [2], who pioneered the idea of information being classified at multiple sensitivity levels. Denning extended the Bell-LaPaulda model to use a lattice for sensitivity labels [5]. Denning was also one of the first to use static analysis on source code to enforce information flow properties with very little runtime overhead [6]. Denis Valpano was the first to formalize the soundness of the analysis that Denning proposed [18]. Andrew Myers et al [15] were the first to use 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.

7 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. They cannot enforce policies that explicitly track data through the virtual machine. Static approaches to controlling information flow do not handle dynamic policies very well, and force a very early binding of code and policy. While operating systems have supported mandatory access controls for a long time, virtual machines currently do not have any support for it.

As a solution, 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. This is the natural level of abstraction at which a programmer thinks while writing Java code, as well the core implementation structure around which a JVM is built, and seems to be the natural abstraction for reasoning about information flow in a JVM. The enforcement mechanism and specification of policy are kept strictly separate from each other. This allows us to use our enforcement mechanism with a wide variety of policies.

8 Acknowledgments

This material is based on research sponsored by the Air Force Research Laboratory under agreement number FA8750-05-2-0216. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Java virtual machine tools interface. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/.
2. D. Bell and L. LaPadula. Secure computer systems: mathematical foundations. *Report MTR 2547 v2, MITRE*, November 1973.
3. C. Bernardeschi, N. D. Francesco, and G. Lettieri. Using standard verifier to check secure information flow in java bytecode. In *Computer Software and Applications Conference*, 2002.
4. B. N. Bershad, S. Savage, P. Pardyak, D. Becker, M. Fiuczynski, and E. G. Sirer. Protection is a software issue. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 62–65, Orcas Island, WA, May 1995.
5. D. E. Denning. The lattice model of secure information flow. *Commun. ACM*, 19(5):236–243.
6. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
7. Department of Defense. *Trusted Computer System Evaluation Criteria, DOD standard 5200.28-STD*. 1985.
8. Ú. Erlingsson and F. B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *New Security Paradigms Workshop*, pages 87–95, Ontario, Canada, 22–24 1999. ACM SIGSAC, ACM Press.
9. L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. The Java Series. Addison-Wesley, Reading, MA, USA, 1999.
10. R. Grimm and B. N. Bershad. Separating access control policy, enforcement, and functionality in extensible systems. *ACM Transactions on Computer Systems*, Feb 2001.
11. B. Joy, G. Steele, J. Gosling, and G. Bracha. *Java Language Specification*. Addison-Wesley, 2000.
12. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, 1999.
13. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.
14. Microsoft Corporation. Microsoft .NET, 2003.
15. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.
16. A. Sabelfeld and A. Myers. Language-based information-flow security. 21(1), 2003.

17. S. Smalley, C. Vance, , and W. Salamon. Implementing selinux as a linux security module. Technical report, May 2002.
18. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
19. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, 1993.
20. R. Watson, W. Morrison, C. Vance, and B. Feldman. The trustedbsd mac framework: Extensible kernel access control for freebsd 5.0. In *USENIX Annual Technical Conference*, June 2003.