Practical, Dynamic Information-flow for Virtual Machines

Vivek Haldar
Deepak Chandra
Michael Franz

University of California
Irvine, CA 92697
+1-949-824-7308
{vhaldar,dchandra,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. Moreover, our implementation is independent of any specific JVM, and will work with any JVM that supports the JVM Tools Interface.

1. INTRODUCTION

Language runtimes such as the Java Virtual Machine [14] and the .NET Common Language Runtime [15] have become popular targets for application deployment due to their portability and safety. They provide a number of mechanisms for secure access control. A common technique is for the code producer to digitally sign code. When executing the code, the code consumer looks up a policy mapping code sources to the resources they are allowed to access on the local site. However, current language runtimes can neither specify nor enforce fine-grained policies that track individual resources throughout the execution of a program. For example, one could specify whether or not a program is allowed to access the network at all, but not whether it is allowed to send only the contents of only certain files over the network, and not others.

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.

An example of a system that highlights the need for mandatory access control policies is an extensible application server. An application server has following two objectives: first, it should be flexible and extensible. As the needs of deployment change, one should be able to add functionality to the system. This is usually done by having an architecture for installing plugins to extend the functionality of the system. Second, it should be able to ensure the confidentiality of the secret data that it may be handling. Confidentiality of information is defined through access control policies.

Presently, using discretionary policies one can define policies that either restricts the system to install only trusted plugins, or else denies the entire program access to secret data. This all-or-nothing, coarse-grained policy greatly reduces the flexibility of the system. Instead, we need a way to specify fine-grained, application-specific policies – for example, policies that say "do not write sensitive data to a world-readable folder", or "do not send sensitive data over the network".

Mandatory Access Control (MAC) has been extensively 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 [8] for decades. Recently, mandatory access controls are beginning to appear in commodity open operating systems such as BSD and Linux. Projects like TrustedBSD [25] and Security-enhanced Linux [19] (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, in the form of a plugin that will work with any Java 1.5 compliant virtual machine. We did need to make some modifications to the system libraries, but these are fully backward 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 SOLUTIONS

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 [10]. 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 "do not allow transmitting on the network after reading from the local filesystem". Inlined reference monitors [9] and software fault isolation [24] 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.

At the Java source level, fields and classes can be marked with access modifiers such as `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 easily be accessed using Java's reflection capabilities. Thus, these modifiers should be thought of as an abstraction tool to hide implementation details, rather than as tools 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 [18]. Type-based information flow relies on programmers inserting security label annotations into source code. Myers et al. [17] 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. These are then statically type-checked: successful type-checking implies the absence of illicit information flows.

Attempts to statically impose information flow on bytecode [4] 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 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. Most 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, and some policy needs to be enforced on their execution.

Most static methods for enforcing information flow (such as a Jflow [17]) require the programmer to annotate source with special annotations relevant to information flow. This early binding of policy to code forces the programmer to predict policies under which the code may run. In most real world scenarios, this is unrealistic. The policy the code consumer wants enforced may very well be different than the policy that the programmer encoded.

Adding mandatory access control to the JVM cleanly side-steps 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`, which 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 [11], 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 [12] and the Java Virtual Machine [15] 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 [15]. Thus, there is fairly narrow and well-defined interface 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
```

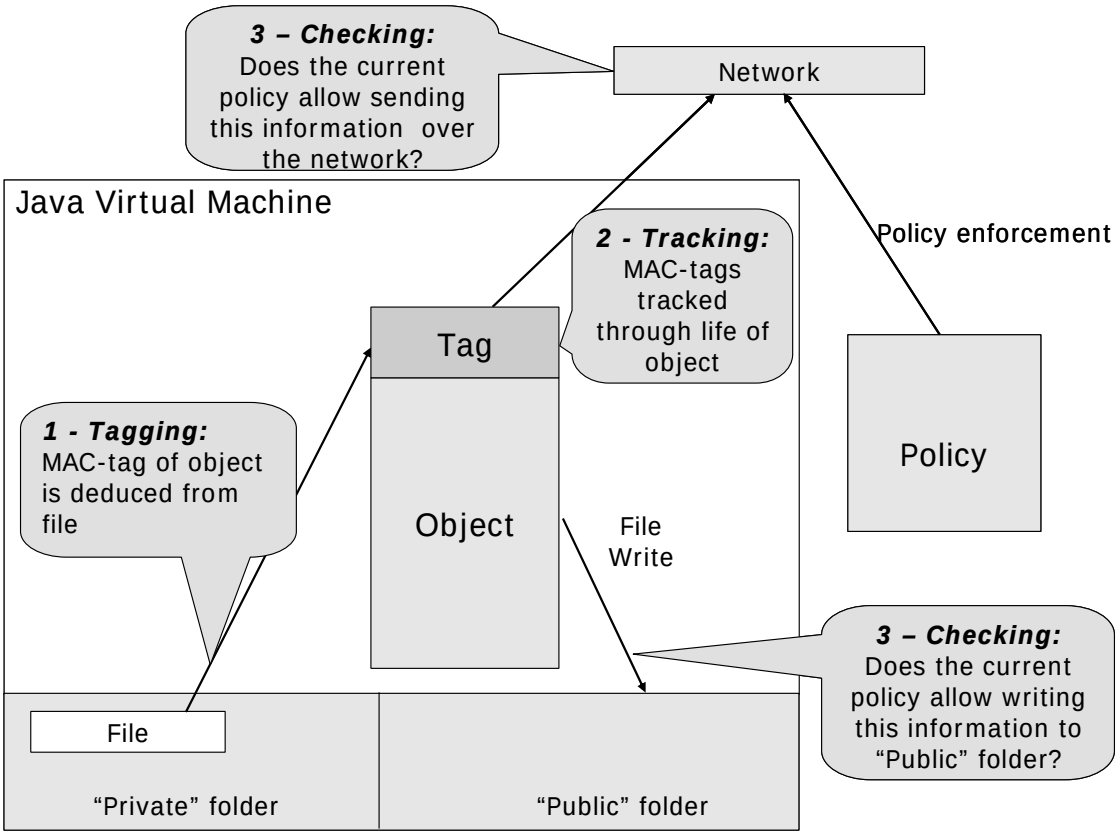

Figure 1: Overview of tracking mandatory access control tags through the lifetime of an object in our MAC-enabled JVM

```
            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 built a proof-of-concept implementation of 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.

For example, to implement strict compartmentalization of data of different tags, we could specify a policy that only allows field reads and writes, and method calls between objects having exactly the same tag, and never changes the tag of an already tagged object. Such a policy would never allow data of different tags to mix.

For another policy, consider the example from section 3. There we allow the mixing of secret and public data, but do not allow secret data to be written to public files. We enforce such a policy by allowing reads and writes between objects with different tags, but at the same time propagating the higher tag. So if a class tagged "public" reads secret data, it too get tagged "secret". Until then, that class is allowed to write to public files, but not thereafter. This dynamic changing of tags allows us to enforce fine-grained policies such as "allow reading of both public and secret data, but do not allow writing secret data to public files".

To support policies such as that used in the examples of Section 2, we need to associate access rules with *channels*. A channel is simply any input/output stream, such as a network sockets or a file handle. An access rule associated with a channel specifies whether data with a particular tag can be output to a channel, as well as what tag data input from the channel has. For our example, a simple access rule specifies that data coming from the `Secret` folder is tagged "secret", and that only data marked "public" can be written to the `Public` folder.

Such policies also require some support from the Java system libraries. For example, the `java.io.File` class needs to tag data according to the folder it is coming from. A number of other input classes need to be similarly modified. We have changed such system classes to support simple channel-based policies. All these changes are fully backward compatible. No new methods have been added to the system classes, and their externally exposed API remains the same as before.

To illustrate these implementation details, consider again the second example from section 3 (line 6 is split into two statements for clarity):

```
class SecretProcessor {
  void processSecret() {
1.   FileReader inSecret = new
                FileReader(secretFile);
2.   FileReader inPublic = new
                FileReader(publicFile);
3.   FileWriter outSecret = new
          FileWriter(anotherSecretFile);
4.   FileWriter outPublic = new
        FileWriter(publiclyViewableFile);
     // this should be allowed
5.   outPublic.write( inPublic.read() );
     // this should also be allowed
6-1. secretChar = inSecret.read();
6-2. outSecret.write( secretChar );
     // this should NOT be allowed!
7.   outPublic.write( inSecret.read() );
  }
}
```

Say an instance of `SecretProcessor`, `sp`, invokes the `processSecret` method. The flow of execution proceeds as follows: to start with, `sp` gets the default tag of "public"; on line 1, `inSecret` is tagged "secret" since it is opening a secret file (the `java.io.FileReader` class was modified to do this); on line 6-1, `sp`'s tag changes to "secret" since it has now read secret data (the instrumentation of method exits does this); since its tag is now "secret", `sp` is now stopped from performing the write on line 7. Figure 1 gives an overview of this.

To get an estimate of the overhead of adding MAC-tags to objects, we measured its overhead for 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. We 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 121.2.

We have not optimized our current implementation for performance. Our first goal was to get a quick prototype that would enable us to explore the ideas presented here. We believe that other techniques, such as load-time bytecode re-writing, will be much more competitive in terms of performance. We are currently exploring alternative ways to implement our scheme.

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 non-portable. 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 its 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 [25] and Security-Enhanced Linux [19].

Trusted computing [20] 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 manipulate various aspects of an object at runtime, such as 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.

Explicit channels for the transfer of information, such 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 [13]. 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 in-formation about the size and number of objects. This 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.

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 [3], 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 [6]. Denning was also one of the first to use static analysis on source code to enforce information flow properties with very little runtime overhead [7]. Dennis Volpano formalized the soundness of the analysis that Denning proposed [23]. Andrew Myers et al [17] 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.

RIFLE [21] is a system that tracks information flow dynamically. This is accomplished by using a combination of hardware and software. The underlying hardware architecture is modified to explicitly track information-flow labels on words. At load time, binaries are rewritten from the standard instruction set to a new one that also appends security labels to instructions. This translation also does a data-flow and reachability analysis on the binary. This converts implicit flows to explicit flows that can then be tracked by the architecture.

The major difference between RIFLE and our system is that our solution is software-only and does not require modifications to the underlying hardware architecture. However, since RIFLE analyses native binaries, it can enforce its constraints on a much wider range of programs, whereas our solution only works for Java bytecode.

7. CONCLUSIONS

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. Our prototype implementation is independent of any specific JVM, and will work with any Java 1.5 compliant virtual machine. We have implemented and tested various examples and policies that demonstrate how MAC-enabled VMs can enforce fine-grained information-flow policies that current Java security mechanisms are unable to neither specify nor enforce.

8. ACKNOWLEDGEMENTS

Parts of this effort are sponsored by the National Science Foundation under grants CCR-TC-0209163 and CCR-ITR-0205712, and by a generous gift from Sun Microsystems Labs.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the National Science foundation (NSF), any other agency of the U.S. Government, or those of Sun Microsystems, Inc.

9. REFERENCES

[1] Java virtual machine tools interface. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/.

[2] J. Agat. Transforming out timing leaks. In ACM Symposium on Principles of Programming Languages, pages 40–53, 2000.

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

[4] 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.

[5] 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.

[6] D. E. Denning. The lattice model of secure information flow. Commun. ACM, 19(5):236–243.

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

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

[9] 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.

[10] L. Gong. Inside Java 2 Platform Security: Architecture, API Design, and Implementation. The Java Series. Addison-Wesley, Reading, MA, USA, 1999.

[11] R. Grimm and B. N. Bershad. Separating access control policy, enforcement, and functionality in extensible systems. ACM Transactions on Computer Systems, Feb 2001.

[12] B. Joy, G. Steele, J. Gosling, and G. Bracha. Java Language Specification. Addison-Wesley, 2000.

[13] B. W. Lampson. A note on the confinement problem. Communications of the ACM, 16(10):613–615, 1973.

[14] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. The Java Series. Addison Wesley Longman, Inc., second edition, 1999.

[15] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, 1999.

[16] Microsoft Corporation. Microsoft .NET, 2003.

[17] A. C. Myers. JFlow: Practical mostly-static information flow control. In Symposium on Principles of Programming Languages, pages 228–241, 1999.

[18] A. Sabelfeld and A. Myers. Language-based information-flow security. 21(1), 2003.

[19] S. Smalley, C. Vance, , and W. Salamon. Implementing selinux as a linux security module. Technical report, May 2002.

[20] Trusted Computing Platform Alliance. TCPA pc-specific implementation specification (http://www.trustedcomputing.org), May 2001.

[21] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In 37th International Symposium on Microarchitecture, December 2004.

[22] Volpano and Smith. Eliminating covert flows with minimum typings. In The 10th Computer Security Foundations Workshop. IEEE Computer Society Press, 1997.

[23] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. Journal of Computer Security, 4(3):167–187, 1996.

[24] 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.

[25] 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.