Towards Formalizing the Java Security Architecture of JDK 1.2

Lora L. Kassab[1] and Steven J. Greenwald[2]

[1] Naval Research Laboratory
Center for High Assurance Computer Systems
Washington, D.C. 20375, USA
kassab@itd.nrl.navy.mil
[2] Independent Consultant
2521 NE 135th Street
North Miami, Florida 33181, USA
sjg6@gate.net

Abstract. The Java security architecture in the Java Development Kit 1.2 expands the current Java sandbox model, allowing finer-grained, configurable access control for Java code. This new security architecture permits more precise, yet flexible, protection for both remote code (loaded across a network connection) and local code (residing on the same machine running the Java Virtual Machine) developed using the Java programming language. Our formal model and analysis is intended to: (1) allow designers and implementors to understand and correctly use the protection provided by these security controls, and (2) provide guidance to a JVM implementor wishing to support these security controls. Access control decisions in Java are made based on the current execution context using stack introspection. To model this, we employ a state-based model that uses multiple access control matrices to model the security controls in JDK 1.2. We also present a safety analysis and discuss the effects of static and dynamic security policies for a given Java Virtual Machine.

1 Introduction

The rise of Java as a programming language has important implications if employed for high assurance systems. Java can be used to implement systems that have high assurance requirements and must perform critical functions correctly. The security controls supported by the Java programming language and the Java Virtual Machine (JVM) can be a significant benefit in developing high assurance systems in Java, but only if: (1) designers and implementors understand the protection those controls can provide and develop a system that uses the controls appropriately to meet system requirements, and (2) the JVM on which the system runs implements the controls correctly, without providing loopholes or bypasses.

This paper addresses the first of these two concerns by providing a model for the security controls implemented in the Java Developer's Kit (JDK) 1.2.

The purpose of this model is to provide: (1) designers and implementors with a precise understanding of how these controls work so that they can design systems that use the controls correctly, and (2) guidance to a Java virtual machine implementor wishing to support these security controls. The need for a precise understanding of the security controls is highlighted by the fact that the new controls require complex changes to the functionality of the virtual machine. As noted before in [18], changes to the JVM can destabilize the entire system.

The issue of whether a given implementation of the JVM on a given platform implements the controls correctly is important but beyond the scope of this paper. One interesting opportunity created from having many JVMs on different platforms is that the same application could be executed on a variety of underlying hardware and software platforms and their results compared as a way of reducing vulnerability to hardware flaws, software flaws, and perhaps even malicious attacks or misuse [14].

Even with such security concerns, Java's still-growing popularity has several roots. Language features such as type-safety, automatic memory management and range checking on strings and arrays are examples of how Java reduces the chance of some common kinds of programming errors, *e.g.*, buffer overflow. (This is very important for safety-critical software as well.) Further, the availability of JVMs for a wide variety of platforms brings the possibility of "write once run anywhere" to application developers. The inclusion of applet tags in HTML together with the incorporation of the JVM into web browsers permits web servers to provide downloadable executable content (Java applets) transparently, moving appropriate computing tasks from the server to the client domain.

Java's developers recognized the concerns that users might have about permitting arbitrary applets to execute on their machines and initially provided a simple "security model" for applet execution [20]. In this model, downloaded applets would be confined by the JVM to executing within a "sandbox" that would prevent them from, for example, altering the client's file system, or communicating with any network sites other than the site from which they were downloaded. Java applications (local programs not downloaded via a web browser) would execute as "trusted", without sandbox constraints.

Predictably, some initial JVM implementations didn't implement the "sandbox" correctly [4, 6], and when they did, developers sometimes found the sandbox model too restrictive. Further, although the notion of a sandbox is simple, its detailed implications for security enforcement are not. Java applications, on the other hand, were (like typical programs run on clients) subject only to the user's constraints, and the idea that some additional constraints (perhaps less stringent than those imposed on applets) might be imposed on applications has appeal to users and developers alike.

Consequently, Javasoft has developed more flexible security controls in JDK 1.2 [8]. With this new architecture, the earlier constraints on applets can be relaxed (depending on the source of the applet and any digital signatures that have been used to sign it) and local code can now be subjected to the same security controls as applets. Users can configure policies that their JVM must

enforce on applications as well as applets [9]. Thus, the security controls now allow finer-grained access controls for applets as well as applications. This results in a security architecture that allows multiple sandboxes with varying access permissions to co-exist.

But with flexibility comes complexity and thus error-proneness. The reason for developing this formal security model is to answer precise, detailed questions about how the security controls are intended to behave and the implications of how each JVM implements the security controls. These questions often reveal subtle, undocumented, and sometimes unexpected behaviors that surface in an implementation of the JVM. As we describe the formal model, we will identify such aspects of JDK 1.2's current security controls and we will discuss the effects of static and dynamic security policies for a given JVM.

2 Overview of JDK 1.2 Security Controls

The security architecture in JDK 1.2 is based on protection domains that represent units of protection within the Java runtime environment. *Protection domains* are defined according to: (1) the location of where the Java code originated (the URL codebase), and (2) a set of cryptographic keys corresponding to the private keys that signed the code. Every *.class* file[1] belongs to a *single* protection domain and is granted permissions according to its domain. Thus, a protection domain is scoped by the set of classes and objects (in the object-oriented sense) that are currently directly accessible by a principal, where a principal is an entity in the computer system to which permissions are granted [8].

Security policy files are used to specify the overall system security policy. These files contain a sequence of permission entries specifying which protection domains should be created and what permissions to grant to each protection domain. For each user running a JVM, there is a *system security policy file* and optionally a *user security policy file* that can be added to it. Since there are no negative permissions, the composition of the system and user policy file is simply the union of the two files to specify the policy "in-effect" for a user's JVM.[2] If neither policy file is present, then the default security policy is the original sandbox policy.

An example of a permission entry in a security policy file is:

```
grant codeBase "http://www.itd.nrl.navy.mil" SignedBy "abcdefg" {
    permission java.io.FilePermission "/home/foo/bar", "read", "write";
};
```

The above example indicates that code originating from the URL *http://www.itd.nrl.navy.mil/* signed with the key "abcdefg" has the permission to read and write to the system resource (in this case a file) "/home/foo/bar."

[1] A *.class* file is the bytecode corresponding to the source code for a Java object-oriented class.

[2] To date, a JVM is implicitly owned by one user only.

Access control decisions are not based solely on the contents of the policy file(s). Thus, an entry in the security policy file does not necessarily warrant access to a system resource, because access control decisions are also made depending on the execution context. The execution context includes checking all protection domains and their associated permissions "involved" in the request before granting any permissions. This design prevents more restricted protection domains from acquiring permissions not indicated in the security policy files. This mechanism is known as *extended stack introspection* and has been implemented by JavaSoft, Netscape, Microsoft and other vendors. Each implementation varies [18, 16, 15, 8], but the core design is similar and is the basis of our model.

Before describing the components of our model, figure 1 illustrates how the JDK 1.2 security architecture maps into our model (defined in Sections 3 and 4). On the left-hand side of the figure, we show four protection domains that may co-exist, which have varying permission boundaries. We represent the security policy for this JVM with a VM policy matrix as indicated on the right-hand side of this figure. This VM policy matrix specifies the security policy, which is constructed from the system policy file (and possibly a user policy file based on the user starting the JVM) for the four protection domains currently executing on this JVM. Using this VM Policy Matrix, we define a domain matrix for each thread of execution on a JVM. These domain matrices correspond to the multiple sandboxes with varying permissions that may co-exist in JDK 1.2.[3]

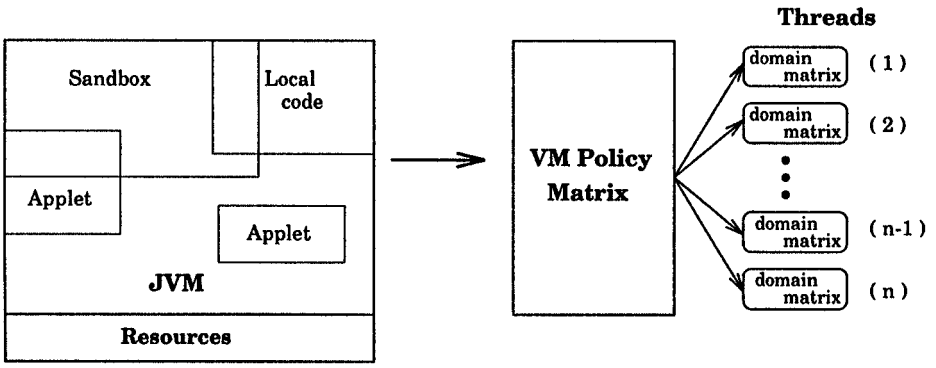

Fig. 1. JDK 1.2 Security Architecture Mapping

In the following sections, we define our Java Access Model (JAM), a state-based model that uses access control matrices to model protection states. There are a few noteworthy distinctions between JAM and other access matrix models. First, even though JAM relies heavily on the use of matrices, it does not

[3] The permissions will vary based on the protection domains (and their associated permissions) that are present in a domain matrix.

have any individual matrix cell operations *per se*, which eliminates the ability to add, remove or transfer access rights between principles. Second, access is not accomplished by a lookup of a single matrix cell, as in many of the traditional techniques [13]. Instead, a special columnar operator is used to determine access. Finally, more than one matrix (VM Policy Matrix and domain matrices) is necessary to model the security controls for a JVM. These differences rule out many of the characteristics of other access control matrix models such as Bell-LaPadula (BLP) [2], Harrison, Ruzzo, and Ullman (HRU) [11], and Sandhu's TAM model [17]. The next section defines our formal representation of the JDK 1.2 security architecture.

3 The Virtual Machine Policy Matrix

The virtual machine policy matrix is a (source by target) matrix of allowable actions. We also show how a given virtual machine policy matrix defines any possible domain matrix that may exist for a JVM. Every component in the model is finite. The definitions of the virtual machine policy matrix follow.

Definition 1. *R is a set of resources. Elements of R model anything available through the operating system, e.g., files, devices or network connections.*

Definition 2. *A is a set of actions. A includes the permissions provided by the underlying operating system (e.g., read, write, execute).*

Definition 3. *S is a set of sources. A source, is a pair (n, k), where n is a URL that names the location of a collection of classes and k is a set of public keys that are associated with the signers of the classes in n. If n is the wildcard *, this represents any URL. Likewise, if k is the string "*", then this means the corresponding collection of classes are not signed. Similar to subjects in traditional access control matrix models, sources denote the rows in an access control matrix. The elements of S are not necessarily disjoint. That is, there may be sources in S, where the collection of classes and the corresponding keys (if any) overlap. When this happens we say, one source is a prefix of another. Source $s_1 = (n_1, k_1)$ is a prefix of source $s_2 = (n_2, k_2)$ if $n_1 \preceq n_2$ and $k_1 = k_2$, where $\preceq$ denotes the URL prefix operator. For example, the source (http://www.itd.nrl.navy.mil/, "abcdefg") is a prefix of source, (http://www.itd.nrl.navy.mil/people/, "abcdefg"), because the keys match and the URL that names a collection of classes in the first source is a prefix of the URL http://www.itd.nrl.navy.mil/people/.*

Definition 4. *T is a set of targets. Targets are strings that name sets of resources. Similar to targets in traditional access control matrix models, targets denote the columns in an access control matrix. However, in a VM policy matrix, it is not always the case that every source is also a target as in other access control matrix models. In a VM policy matrix, it is possible that more than one target refers to the same set of resources. For example, the use of symbolic links in UNIX causes more that one path name to refer to the same physical file.*

Definition 5. *The virtual machine policy matrix PM is an $|S| \times |T|$ matrix, where $PM[s,t] \subseteq A$ gives the allowable actions for target t, using the collection of classes loaded from source s. The policy matrix PM is a static representation of the system security policy file (with the possible addition of a user security policy file), where each row represents a source and every column is a target. The virtual machine policy matrix is defined at Java runtime start-up, thus defining the security policy for a given JVM until the JVM terminates.*

*Figure 2 is a simple example of a policy matrix. Recall that sources are composed of a URL (or * for any URL) and the set of associated public keys (or the wildcard string, "*", if not signed). An empty cell in the matrix implies that the source has no permissions for the corresponding target. In figure 2, the last entry has a wildcard, *, to specify code loaded by the JVM from any URL receives the following permissions. Thus, all code may read files in the directory "/tmp", so this expands the boundary of the original sandbox.*

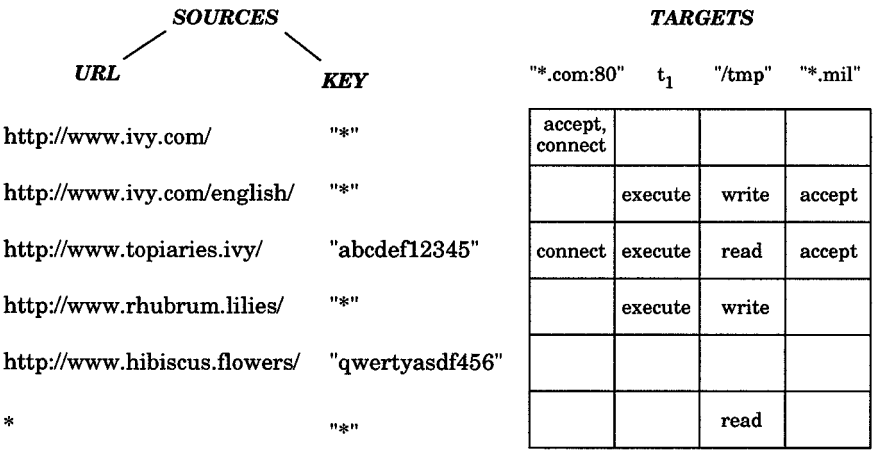

Fig. 2. Virtual Machine Policy Matrix

Definition 6. *DM is a set of domain matrices. Each element $dm \in DM$ is an $|S_{dm}| \times |T|$ matrix corresponding to a thread of execution on a JVM, where $S_{dm} \subseteq S$ and T is the set of targets. Each row in the domain matrix is similar to a capabilities list, which for Java is called a "protection domain". The actions in $dm[s,t]$ are the union of all actions in virtual machine matrix element $PM[s',t]$ for every source $s' \in S$ that is a prefix of s. Thus, for every source $s' \in S$ that is a prefix of $s \in S_{dm}$, the permissions granted to s' for each target $t \in T$ are granted to s for t.*

Figure 3 is an example of a domain matrix (with a single row) which is based on the policy matrix in Figure 2. To illustrate how the domain matrix was created, suppose the class http://www.ivy.com/english/green/foo.class were loaded by a JVM with the policy depicted in Figure 2 in effect. Based on the protection domain of this class, its set of permissions would be the union of the two sources in the policy matrix that are prefixes of this class, namely (http://www.ivy.com/, ""), (http://www.ivy.com/english/, "*"), and the last entry in the matrix which applies to any URL. Note that the code loaded from this source was not required to be signed (as indicated by "*").*

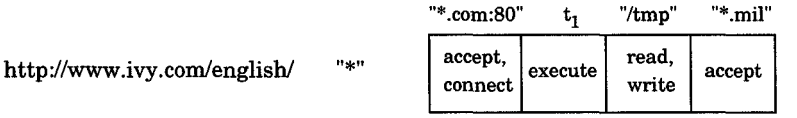

Fig. 3. Domain Matrix

A domain matrix (with multiple rows) can be viewed as stack (LIFO order), where the last row added is at the bottom of the matrix. We decided, however, to use a matrix representation for consistency with the PM representation. With the matrix form we do not lose any generality as we can manipulate it in a stack-like way when necessary. Also, the matrix allows us to determine permissions (defined later) with greater ease than a stack.

4 Java Access Model

The virtual machine policy matrix defined above applies to a single JVM. In this section, we define the Java Access Model (JAM), which is the state-based model for a single virtual machine policy matrix.[4] Our Java Access Model is defined as $JAM = (PS, C, f, U)$, where PS is the set of protection states, C is the set of commands (inputs) that cause JAM to transition from one protection state to another, $f : PS \times C \rightarrow PS$ is the transition function, and U is a set of principals. We use the sequence of input commands $c \in C^*$ to distinguish the various entities that are found in a protection state. Each entity is given the subscript of the command that created it. For example, if command i in the input sequence creates a domain matrix, then the domain matrix is labeled dm_i. The following four definitions define the components of JAM.

[4] JAM can be modified to allow more than one JVM to concurrently execute on a single computer. However, the focus of this paper is on modeling the security controls of a single JVM.

Definition 7. *U is a set of principals. It models the entities to which authorizations are granted (and as a result, accountability). Today, a JVM is implicitly owned by a single user causing U to be a singleton set.*[5]

Definition 8. *A protection state ps is a sextet (V, Π, S, R, T, A), where*

- *V is a triple u, PM, DM, where u is a principal, PM is a virtual machine policy matrix, and DM is a set of domain matrices.*
- *$\Pi : u \rightarrow PM$ is the policy function that represents the effect of Java policy files; it maps a principal u to a virtual machine access matrix PM. Thus, this function returns the matrix based on the system policy file. (If a user policy file exists, the union of system and user policy file is used. This is easily accomplished since there are no negative permission entries in either file.)*
- *S, R, T, A are the sets that are used to define the policy and domain matrices in V.*

Definition 9. *The protection state $ps \in PS$ of JAM changes by one of the following commands in C.*

- **init**(u): *The command init(u) creates a new triple, $V = (u, PM, DM_i)$, where u is a principal in U, $PM = \Pi(u)$ and DM_i is an empty set of domain matrices (no threads of execution). This command models a principal starting a JVM.*
- **destroy**(PM): *Remove the triple $V = (u, PM, DM_i)$ from the current protection state ps. This command models the termination of a JVM.*
- **start**(dm, s): *Add a new domain matrix dm to the set of domain matrices DM_j for the triple (u, PM, DM_j). This command models the creation of a new thread executing the class(es) from source s, where the domain matrix dm consists of all targets in T and the set of sources S_{dm} for this domain matrix is the singleton set s. If start(dm, s) is the i^{th} command in the input sequence c, then the new domain matrix is dm_i, where $i > j$. The actions in $dm[s, t]$ for each $t \in T$ are obtained by the union of $PM[s', t]$ for every $s' \in S$ that is a prefix of s.*
- **stop**(dm): *Remove the domain matrix dm from the set of domain matrices DM. This command is less interesting as it simply removes permissions for a thread. This command, however, must also remove any privileges that may have been associated with dm (the "privileged" mechanism is discussed in Section 4.1).*
- **enter**(s, dm): *Add the source $s \in S$ to the set S_{dm} for domain matrix $dm \in DM$. This command adds a new row to the domain matrix dm. This models a thread of execution entering the protection domain s. As in the command start, the actions in $dm[s, t]$ for each $t \in T$ are obtained by the union of $PM[s', t]$ for every $s' \in S$ that is a prefix of s.*

[5] This set would not be a singleton if multiple JVMs are modeled or if protection domains are extended to include the notion of "running-on-behalf" of a principal as presented in [1, 8]. Thus, we maintain this set for clarity and extensibility.

- **exit**(s, dm): *Remove source s from the set S_{dm} of domain matrix dm. This command removes a row from the domain matrix dm. This command models a thread of execution exiting the protection domain s. This command is most interesting as the removal of a row in a domain matrix has the potential to increase the permissions granted to a thread, which is atypical of commands in traditional access control models. That is, the removal of a row will never decrease the permissions granted to a thread of execution on a JVM.*

Definition 10. *$PERMIT : dm, t \rightarrow a$ is the permit function which returns the allowable set of actions $a \in A$ that a thread may perform on a target t. For every $s \in S_{dm}$, the permit function takes the intersection of $dm[s, t]$ for target t as follows.*

$$PERMIT(dm, t) = \bigcap_{s \in S_{dm}} dm[s, t]$$

The resulting set is the effective set of actions that a thread may perform. It is possible that a given s does not have any actions for a target t_i, which is denoted by an empty cell in the matrix. In this situation, the intersection would return the empty set, since there exists at least one source $s \in S_{dm}$ where $dm[s, t_i]$ is an empty cell.

		"*.com:80"	t_1	"/tmp"	"*.mil"
http://www.topiaries.ivy/	"abcdef12345"	connect	execute	read	accept
http://www.rhubrum.lilies/	"*"		execute	read, write	
http://www.ivy.com/english/	"*"	accept, connect	execute	read, write	accept

Fig. 4. Domain Matrix

Given the domain matrix illustrated in Figure 4 and a target t_i, the $PERMIT$ function would take the intersection of the permissions for that target for every source in the domain matrix. In Figure 5, the intersection of the permissions for all targets has been calculated. For example, using the *dm* in Figure 4 and the target "/tmp", the $PERMIT$ function would return the permission *read*.

4.1 The "Privileged"

As stated previously, Java access control decisions are made based on the current execution context via stack introspection. This is not always a "complete" introspection, as blocks of Java code may declare that they are "privileged" by using the *beginPrivileged()* and *endPrivileged()* methods of the JDK class *java.security.AccessController*. The term privileged is not equated with the usual security notion of trusted code. Rather, "privileged" in this context means that

	("*.com:80"	connect)	(t_1 execute)	("/tmp"	read)	("*.mil"	accept)
∩	("*.com:80"	)	(t_1 execute)	("/tmp"	read, write)	("*.mil"	)
	("*.com:80"	accept, connect)	(t_1 execute)	("/tmp"	read, write)	("*.mil"	accept)
	("*.com:80"	)	(t_1 execute)	("/tmp"	read)	("*.mil"	)

Fig. 5. *PERMIT* function example

the block of code is responsible for requesting access to its resources, but the block of code can never gain more permissions than indicated in the system policy file (and user policy file if one exists). For example, a block of code may use this privileged construct when accessing system files, such as fonts or libraries, when other less permissive are likely to be on the execution stack.

We model the privileged construct through the commands $enter(s, dm)$ and $exit(s, dm)$, where the trace of these commands imposes an execution ordering of the protection domains traversed by a thread of execution. That is, the *enter* command appends rows to a domain matrix and *exit* removes rows in LIFO order.[6] Therefore the top-most source in the domain matrix is older than the sources below it in the matrix, which is reflected in the sequence of commands on the domain matrix. The use of "privileged" circumvents the need to check all sources in the domain matrix by only checking the sources that were entered after the "privileged" source (the entries below the privileged entry in the domain matrix).

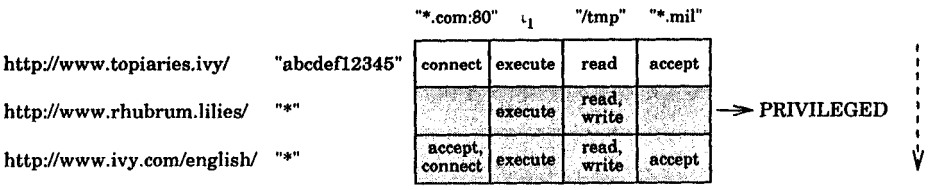

Fig. 6. Domain matrix with a privileged protection domain

In Figure 6, code obtained from the source (*http://www.rhubrum.lilies/*, "*") executed the method *beginPrivileged()*. Now, only the more recent protection domains need to be checked as indicated by the shaded portion of the figure before granting permission to any resources. We define the *PERMIT* function under "privileged" as the intersection of $dm[s, t]$ over every source s added by a command input that is greater than or equal to the command input that added

[6] Recall that a domain matrix can be viewed as stack, where the most recently added row is at the bottom of the matrix (as indicated by the arrow in Figure 6).

the privileged source as follows:

$$PERMIT(dm, t) = \bigcap_{c_i(s) \geq c_j(privileged)} dm[s, t]$$

Since the code is executing as privileged, the results of $PERMIT$ may not be the same. As indicated in Figure 7, the permissions allowed are calculated given the context of a thread of execution. Now, if the $PERMIT$ function is applied to the target "/tmp" and the domain matrix in Figure 3, the permissions granted are both *read* and *write* (as opposed to just *read* as in Figure 5).

∩	("*.com:80"	)	(t_1 execute)	("/tmp"	read, write)	("*.mil"	)
	("*.com:80"	accept, connect)	(t_1 execute)	("/tmp"	read, write)	("*.mil"	accept)
	("*.com:80"	)	(t_1 execute)	("/tmp"	read, write)	("*.mil"	)

Fig. 7. Permissions granted with a privileged protection domain

It is important to note that the privileged mechanism is scoped within a single thread. Thus, privileges cannot leak to "untrusted" code. Although this approach allows straightforward auditing of system code to verify that privileges are enabled only for a limited time, it is still possible for a developer to "forget" to call *endPrivileged*. The potential danger in this scenario depends on whether the implementation automatically removes the privilege once the code that called *beginPrivileged* exits. Although we do not provide this comparison in this paper, we will mention that Netscape, Microsoft, and JavaSoft vary in their implementation of the privileged mechanism [18, 9].

5 Safety Analysis of JAM

We previously stated that users are not limited in what is entered into policy files. Consequently, policy matrices (PM's) are populated based on the "arbitrary" construction of policy files, which precludes our defining a secure initial state. These policy matrices are then used to form domain matrices (dm's), which are used to grant or deny permissions (via the $PERMIT$ function). Therefore our safety analysis focuses on the problem of whether a particular dm can be generated given a particular PM. We term this safety problem the *Arbitrary Policy Matrix* problem (APM) and formulate it as a decision problem to facilitate our analysis.

The two theorems that follow show that APM is in NP and that APM is actually an NP-complete problem. This fact is significant for the following reasons. It demonstrates how a very interesting result can be obtained from a simple manipulation of the model, thus demonstrating the expressiveness of JAM. Also,

it should be of great importance to developers working with JDK 1.2, since the class of NP-complete problems are generally considered to be "intractable."

5.1 The Arbitrary Policy Matrix Problem

APM can be formulated quite simply: given an initial protection state ps_0 and a final protection state ps_f, is it possible for a particular domain matrix dm to appear in any future state ps_i such that $1 \leq i \leq f$? We would like to be able to answer this question because there might be dm's that are undesirable and it would be beneficial to be able to determine if such dm's would be possible for an arbitrary policy matrix PM.

Theorem 1. *(1.) APM $\in$ NP.*

Proof. For each row in the dm, the worst case requires checking all possible combinations of the unions of all the rows in PM. There is an exponential number of possible rows, bounded by $2^S - \phi$, where S is a set of sources as defined in Section 3 and 4. Therefore, the problem can be posed as a decision problem, where an instance of a dm is guessed by a nondeterministic algorithm, and then compared to the dm in question. Note that the actual comparison is done in polynomial time mn where m is the number of rows and n is the number of columns, being a simple comparison of two matrices.[7]

In order to solve this problem, a nondeterministic algorithm need only guess a particular dm from the exponential number of possible dm's that may be created (from any particular PM) in the states of the machine. We then need to compare the guessed dm with the particular dm of interest. As mentioned above, the comparison of the two matrices takes place in polynomial time, and the number of states is linear (and therefore also polynomial). Therefore, the problem is in NP. □

Theorem 2. *(2.) APM is NP-complete.*

Proof. We polynomially reduce the well-known NP-complete Satisfiability (SAT) problem [3] used in Cook's theorem to APM.

We briefly review SAT. We have a set B of Boolean variables, and a collection C of clauses over B. The decision problem being: is there a satisfying truth assignment for C? The following example is paraphrased from Garey and Johnson [7] (we have changed some of the set names to conform to our notation).

> Let B be a set of Boolean *variables.* A *truth assignment* for B is a function $t : B \rightarrow \{T, F\}$. If b is a variable in B, then b and $\bar{b}$ are *literals* over B. A *clause* is a set of literals over B representing the disjunction of those literals and is *satisfied* by a truth assignment iff at least one

[7] If multiple source prefixes exist in the rows of the policy matrix, then we can collapse the policy matrix by replacing the prefix sources with a single row that is the union of the prefixes. This will yield the most permissive set of possible dm's without loss of generality.

of its members is true under that assignment. A collection C of clauses over B is *satisfiable* iff there exists some truth assignment for B that simultaneously satisfies all the clauses in C.
For example, $B = \{b_1, b_2\}$ and $C = \{\{b_1, \bar{b}_2\}, \{\bar{b}_1, b_2\}\}$ provide an instance of SAT for which the answer is "yes." A satisfying truth assignment is given by $t(b_1) = t(b_2) = T$. Alternatively, replacing C by $C' = \{\{b_1, b_2\}, \{b_1, \bar{b}_2\}, \{\bar{b}_1\}\}$ yields an instance for which the answer is "no." C' is not satisfiable.

We need to reduce, in polynomial time, an arbitrary instance of a SAT problem to APM.

We restrict ourselves to two actions in JAM, such that $A = \{T, F\}$ corresponding to Boolean TRUE and FALSE respectively. Let $m = |C|$ and $n = |B|$. We then construct a PM that has $m' = 2^n$ rows, and n columns such that each column corresponds to a unique variable in B. This gives us a PM of size $m'n$. The cells of the PM are constructed in such a way that we can generate dm's with m rows and n columns where each dm cell contains only one action, T or F (this means that the PM will contain every possible combination of unique rows with cells composed of T's and F's).

We are now ready to begin the reduction of the problem instance. The particular dm that we want to compare against is termed dm'. dm' has m rows and n columns. Each row in dm' corresponds to a particular clause in our instance of SAT, and each column corresponds to a particular variable in B. We use the function $g : C \rightarrow dm'$ to fill in some of the cells of dm'. Function g works the following way. If a particular literal in a clause in C is not negated (*i.e.,* there is no bar over the literal), then we change the corresponding cell in dm' to T. If a particular literal in a clause in C is negated (*i.e.,* there is a bar over the literal), then we change the corresponding cell in dm' to F (note that it is likely that some of the cells of dm' will be empty). It takes polynomial time mn to generate dm'. At this point the reduction of the problem instance is finished.

We now use a nondeterministic algorithm to generate possible solutions to APM by generating one dm out of a possible 2^{mn} unique dm's that are the same size as dm', where each cell contains either the action T or the action F (henceforth a dm guessed by our nondeterministic algorithm will be referred to as just "dm"). At this point, each row in the constructed dm is analogous to applying the SAT truth function t to the Boolean variables in B, and placing the results in each cell, where T is equivalent to a Boolean true and F is equivalent to a Boolean false. For example if $B = \{b_1, b_2, b_3, b_4, b_5, b_6\}$ then we would have a dm that might contain T, T, F, F, T, F respectively in each of the 6 cells in one of its rows (this being a particular guess we are checking). It takes time mn to "guess" a solution dm giving a total time of $2mn$ at this point.

We now do the actual comparison of dm against dm' to check for a satisfying truth assignment. We decompose the guessed dm into m one-row matrices dm_i^1, $1 \leq i \leq m$, taking time mn for a total time of $3mn$ at this point. We then construct a temporary matrix, dm_h that is the same size as dm' by applying a new function $h : dm'[s, t]$, $dm_i^1[t] \mapsto dm_h[s, t]$ to each cell in dm_h. Function h

works as follows. If cell $dm'[s,\ t]$ is empty, then h inserts F into $dm_h[s,\ t]$. If the cell contains T, h copies the action from $dm_i^1[t]$ to $dm_h[s,\ t]$. If the cell contains F, then h places the compliment of $dm_i^1[t]$ into $dm_h[s,\ t]$. This takes time mn giving a total time of $4mn$ at this point. To solve, we do a Boolean OR operation on each of the m rows in dm_h (taking time mn for a total time of $5mn$), followed by a Boolean AND operation of the m results taking time m for a total time of $5m^2n$. To finish the comparison we do the above for each dm_i^1 (since there are a total of m one-row matrices, this takes time m for a total time of $5m^3n$), and do a Boolean AND operation on the m results[8] taking time m. Therefore, the total comparison time of dm against dm' is the polynomial $5m^4n$. To solve SAT, if our final result is true then dm matches dm', and we have an instance of SAT that is a satisfying truth assignment. If all of the exponential number of dm's do not satisfy, then our instance of SAT does not have a satisfying truth assignment.

Any instance of SAT which satisfies, will also satisfy our reduction. Any instance of SAT that does not satisfy, will not satisfy our reduction. Therefore, APM is NP-complete. □

6 Implications of Dynamically Changing Security Policies

Currently, a JVM enforces a single static security policy based on the system security policy file(s) read during initialization. Our safety analysis in section 5 proved that the problem of whether a particular dm can appear given a particular PM is an NP-complete problem. The fact that APM is NP-complete shows that the consequences of a given policy are quite difficult for users to evaluate. Even so, JavaSoft is presently working on a mechanism that allows changes to a JVM security policy after initialization (*i.e.,* dynamically changing PM's) via a secure mechanism [8]. The improved flexibility of dynamically changing security policies is not without penalty due to the increase in complexity of JVM implementations. Flawed implementations, regardless of how they became erroneous, are far more detrimental to security than the benefits provided by this flexibility. Using our model, we can explore the implications for currently executing Java programs with dynamically changing security policies.

One possible way to dynamically change the policy may require certain preconditions to be satisfied before the policy change may occur.[9] The simplest requirement would be that all executing threads are executing within the confines of the original sandbox before the new security policy becomes effective. If domain matrices are utilizing resources "outside" of the original sandbox and the change in the security policy must be effective immediately, then there are a few alternatives. The easiest and most secure implementation would require that all

[8] Technically, the final Boolean AND could be a Boolean OR and still find satisfying truth assignments. However, the AND is necessary in order to retain the structure of the matrix comparison between dm and dm'.

[9] This is a common object-oriented technique known as *design by contract.*

of these threads be terminated (via the *stop(dm)* command in our model) before the update occurs. Although this may seem rigid, co-existing security policies (multiple PMs) for the same principal will complicate the implementation of security controls in a JVM and is subject to luring attacks by allowing more permissive threads to linger through changes to the security policy for a JVM.

A less rigid implementation for dynamically changing the security policy for a JVM may depend on the specific changes to the policy files. If a modification to the policy files makes any protection domain(s) more permissive (only adds actions for one or more targets), then each *dm* that traversed any of these protection domain(s) can simply be granted these new permissions. If implemented correctly, this does not pose a problem. If a modification to the policy files makes any protection domain(s) less permissive (removes actions for any target), then the update is more complicated. For each thread of execution (*dm*) that traverses an affected protection domain, either the update does not occur until the protection domain is no longer accessible by the thread, or the thread of execution is simply terminated.

A naive implementation that allows a *dm* to be updated with fewer (or less permissive) actions may result in the executing applet/application not completing its computation. For instance, suppose a currently executing thread has received permission to write to a given file and the new policy disallows this permission (while the thread is writing to the file). Then, subsequent resource access by that thread will either throw a *java.lang.SecurityException* or incorrectly allow the thread to continue writing to the file.

The design for supporting dynamically changing policies has not been defined yet, but this mechanism is intended to be included in future implementations. Using JAM, we can more easily analyze how this mechanism is best implemented and how threads that persist over changing policies may be dangerous if not handled properly. Clearly, the implementation of this capability to dynamically change a security policy for a JVM is complex (and thus error-prone), and as we have seen in the past, this is typically an avenue for discovering loopholes and bypasses to the security controls.

7 Conclusion

The security controls in JDK 1.2 have many desirable features. Stack introspection is not vulnerable to tampering or direct program access even though stack introspection can have high runtime costs.[10] Further, extended stack introspection also offers good backward compatibility with existing Java applets [18] by defaulting to the original sandbox policy if no other security policy is specified.

Extended stack introspection is not a panacea, however. It requires complex changes to the virtual machine. More specifically, each class needing protection must explicitly consult the security system to see whether the class was invoked by an authorized party. This check adds at least one line of code to each class.

[10] In the worst case, the depth of the stack is traversed before permissions are granted as modeled by the $PERMIT$ function.

As we previously noted, changes to the JVM could destabilize the whole system. This adds risks because the major commercial browser vendors (Microsoft and Netscape) have diverged in their implementation of the JVM.

Inevitably the flexibility afforded by the JDK 1.2 security controls introduces complexity and more opportunity for error-prone JVM implementations. Our formal security model provides a more rigorous and unambiguous specification of the intended behavior of the security control design of JDK 1.2. By formalizing definitions of the security controls, we were able to: (1) identify potential points of variance in JVM implementations (different stack introspection algorithms and the privileged mechanism), and (2) provide a means to compare the actual behavior of the security controls in different JVM implementations.

Our safety analysis focused on whether a particular *dm* can appear in future states because it is not possible to limit what can appear in a policy file. Our safety analysis showed that APM is NP-complete (*cf.* [10, 11, 17]). Therefore, if expressing security by APM is of interest to designers and implementors, then any particular policy file configuration can be inspected (not necessarily efficiently) using methods appropriate for NP-complete problems.

In response to the complexity of static policy files, we used our model to explore the impact of dynamically changing policy files. As additional features of the Java security architecture in JDK are announced, we plan to reuse our model to analyze the ramifications of those features.

Acknowledgements

The authors would like to thank John McDermott, Carl Landwehr, Catherine Meadows, and Richard E. Newman for their contributions to this paper.

References

1. Balfanz, D. and Gong, L.: *Experience with Secure Multi-Processing in Java,* In Proceedings of the International Conference on Distributed Computing Systems, Amsterdam, Netherlands, May 1998.
2. Bell, D. E., and LaPadula, L. J.: *Secure Computer System: Unified Exposition and Multics Interpretation,* Technical Report MTR-2997, MITRE, Bedford, Mass., March 1976.
3. Cook, S. A.: *The Complexity of Theorem-Proving Procedures,* In Proceedings of the Third Annual ACM Symposium on the Theory of Computing, ACM, Pages 151-158, 1971.
4. Dean, D., Felton, E. W., and Wallach, D. S.: *Java Security: From HotJava to Netscape and Beyond,* In Proceedings of the 1996 IEEE Symposium on Security and Privacy, pages 190-200, May 1996.
5. Denning, D.: *Cryptography and Data Security.* Addison-Wesley, 1982.
6. McGraw, G. and Felton, E. W.: *Java Security: Hostile Applets, Holes, and Antidotes.* John Wiley & Sons, 1997.
7. Garey, M. R., and Johnson, D. S.: *Computers and Intractability,* Bell Telephone Laboratories, Inc., Murray Hill, New Jersey, Page 259, 1979.

8. Gong, L.: *JDK 1.2 Security Architecture.* Sun Microsystems, Inc., Palo Alto, California, March 1998.
9. Gong, L., Mueller, M., Prafullchandra, H., and Schemers, R.: *Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2.*, In Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, December 1997.
10. Harrison, M. H. and Ruzzo, W. L.: *Monotonic Protection Systems*, Foundations of Secure Computation, Academic Press, New York, 1978, 337–365.
11. Harrison, M., Ruzzo, W., and Ullman, J.: *Protection in Operating Systems*, Communications of the ACM, 19(8), Pages 461-471, August 1976.
12. Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification.* Addison-Wesley, 1996.
13. Lampson, B. W.: *Protection*, In Proceedings of the 5th Princeton Symposium on Information Sciences and Systems, Princeton, New Jersey, March 1971, Pages 437-443.
14. McDermott, J. and Gelinas, R.: *Prototyping a Java-based Defense Against Storage Spoofing*, Submitted for publication.
15. Accessible from http://www.microsoft.com.
16. http://developer.netscape.com/library/documentation/signedobj.
17. Sandhu, R. S.: *The Typed Access Matrix Model*, In Proceedings of the IEEE Symposium on Research in Security and Privacy, IEEE Computer Society, Oakland, California, Pages 122-136, 1992.
18. Wallach, D. S., Balfanz, D., Dean, D., and Felton, E. W.: *Extensible Security Architectures for Java*, In Proceedings fo the 16^{th} ACM Symposium on Operating Systems Principles, October 1997.
19. Weissman, C.: *Security Controls in the ADEPT-50 Time-Sharing System*, In Proceedings of the Fall Joint Computer Conference, Vol 35, Pages 119-133, 1969.
20. Yellin, F.: *Low Level Security in Java*, In Proceedings of the 4^{th} International World Wide Web Conference, Boston, Massachusetts, December 1995.