Empowering Mobile Code Using Expressive Security Policies*

V.N. Venkatakrishnan
venkat@cs.sunysb.edu

Ram Peri†
ramp@cs.sunysb.edu

R. Sekar
sekar@cs.sunysb.edu

Department of Computer Science
Stony Brook University
NY, 11794

ABSTRACT

Existing approaches for mobile code security tend to take a conservative view that mobile code is inherently risky, and hence focus on confining it. Such confinement is usually achieved using access control policies that restrict mobile code from taking any action that can potentially be used to harm the host system. While such policies can be helpful in keeping "bad applets" in check, they preclude a large number of useful applets. We therefore take an alternative view of mobile code security, one that is focused on empowering mobile code rather than disabling it. We propose an approach wherein highly expressive security policies provide the basis for such empowerment, while greatly mitigating the risks posed to the host system by such code. Our policies are represented as *extended finite state automata*, (a generalization of the finite-state automata to permit the use of variables) that can enforce these policies efficiently. We have built a prototype implementation of our approach for Java. Our implementation is based on rewriting Java byte code so that security-relevant events are intercepted and forwarded to the policy enforcement automata before they are executed. Early experimental results indicate that such expressive, enabling policies can be supported with low overheads.

General Terms

Security

Keywords

Mobile code security, security policies, code transformation

1. Introduction

With the growth of distributed computer network systems and the Internet, there has been an increasing demand to support *mobile code* — code that is downloaded from remote, possibly untrusted systems. The best known examples of this are Java applets, but there are also several other examples such as agent based systems, document/email attachments, and executable content such as Postscript files. Usually, code that is downloaded is executed with the privileges of the user who downloads it. This introduces a number of serious security and safety issues.

Existing approaches for mobile code security tend to take a conservative view that mobile code is inherently risky. Consequently, the primary goal of these approaches is to *confine* mobile code so as to ensure that it can do no harm. This goal is achieved by enforcing stringent access control policies that prevent mobile code from executing any action that can potentially compromise the security of the host system running the code. For instance, Java applets are denied access to read or write any files resident on their host computer, since malicious applets may be able to use such access to corrupt user data or reveal it to unauthorized parties. While such access control policies are helpful in keeping "bad applets" in check, they have the unfortunate side effect of precluding a large number of useful applications of mobile code.

We take an alternative view of mobile code security, one that is focused on *empowering* mobile code rather than *disabling* it. In our approach, expressive security policies provide the basis for such empowerment, while greatly mitigating the risks posed to the host system by such code. For instance, we can allow some local file access to applets without incurring significant risks using a policy such as "an applet can create new files in `/tmp` directory, but cannot delete files except those created by the application itself." Such policies cannot be expressed or enforced using existing mobile code security frameworks.

A policy in our language is specified using an extended finite state automata (EFSA), a generalization of the finite-state automata to permit the use of variables. An EFSA can enforce these policies efficiently via runtime monitoring of mobile code. The transitions of these automata are over an alphabet of events such as function calls, method invocations and exceptions. We have built a prototype implementation of our approach for Java. Our implementation is based on rewriting Java byte code so that security-relevant events are intercepted and forwarded to the policy enforcement automata before they are performed. Early experimental results indicate that such expressive, enabling policies can be supported with low overheads.

The rest of this paper is organized as follows: We begin with several example scenarios in Section 2 that illustrate the weaknesses of existing frameworks for mobile code security and motivate the

†Author's current email is `rperi@bloomberg.net`.

*This research is supported in part by a ONR University Research Initiative grant N000140110967 and NSF grants CCR-0098154, CCR-0208877 and CCR-0205376.

development of more expressive policy frameworks. In Section 3, we present an overview of our policy language and describe our implementation approach for enforcing these policies. Preliminary performance measurements are presented in Section 4. We summarize the related work in Section 5. Finally, concluding remarks appear in Section 6.

2. Motivating Examples

In general, a security policy needs to address the concerns of confidentiality, integrity and availability. Confidentiality is typically achieved by limiting the information that mobile code can access and/or limiting how the code may use this information. Thus, a security policy needs to define acceptable ways of processing, storing and transmitting sensitive information. To ensure integrity, security policies need to control operations performed by mobile code that modify system state. To ensure availability, security policies need to address resource usage by mobile code.

Previous work on mobile code security, such as Java security, has focused on simple access control policies. Subsequent work of Evans et al [6] extended these policies to address resource usage, whereas the work of Erlingsson et al [4] allowed policies on operation sequences. While these extensions address some of the weaknesses of simple access control mechanisms, we illustrate in this section how they are still not able to express security policies that are adequate for a number of applications of mobile code. Our approach for mobile code security policies and their enforcement is very much motivated by this discussion.

2.1 Ability to manipulate temporary files.

Consider a piece of mobile code that needs to create or modify files on a local file system for its internal book-keeping operations. It is perfectly reasonable to allow even untrusted mobile code to do this, as long as the files created/manipulated are unrelated to other applications. This can be ensured using a policy that enforces the following properties:

- *Writes are allowed only to certain directories.* The untrusted code should not be allowed to create files in arbitrary directories. Directories where this code can write may include such directories as `/tmp`.
- *Overwriting or deletion of a file is not permitted except for files created by the same mobile code.* This ensures that untrusted mobile code does not remove files created by other applications.

While it is possible to use simple access control policies to enforce the first property, more expressive notations are needed for the second property. In particular, we need a policy language that can refer to the history of operations carried out in the past, as well as the arguments of these operations. Most existing work in mobile code security policies do not allow such history-sensitive properties. Although the work of Erlingsson [4] allows for history-sensitive policies, their language does not allow argument values for past operations (such as file names) to be remembered for later use.

We point out that the second property is particularly useful, as it allows us to include more directories in which writes are allowed. This way, even mobile applications that create files in arbitrary directories (e.g., current working directory) can be permitted to run without harming the rest of the system. On the other hand, if we are restricted to using only simple access control policies, then the application would have to be further restricted in order that it not destroy files pertaining to other applications. In particular, the untrusted application must be limited to creating files in a special directory that is created explicitly for the purpose of running that code. Although it is possible to write the untrusted application in this manner, it requires advance planning on the part of the person implementing this code. The advantage offered by more expressive policies is that it provides sufficient flexibility to the code consumer so that mobile applications could be run safely, even when the code producer had not anticipated all the possible uses of their code or their potential security implications.

2.2 Access via trusted intermediaries.

Often we encounter situations where a piece of untrusted code needs to access a certain resource. However, permitting direct access to the resource may be risky. One way to deal with this problem is to constrain the mobile application to access the resource through certain operations that are provided by code from a trusted source. From the code consumer's point of view, this code serves as a *trusted intermediary.* Use of this approach can allow untrusted applications to access resources in many contexts such as:

- *Adding an entry to syslog.* To prevent a malicious application from truncating the log or writing bogus entries, we may use a trusted intermediary that provides an operation to log messages to syslog, but ensures that (a) sufficient information to identify the source of the log entry is included, say, in the form of a header prepended to the log message and (b) limiting the number of entries that can be written by an untrusted application
- *Access to local data.* Consider a situation when we have to give a piece of mobile code (e.g., code executing on a browser's space) access to some (chosen) file on the local disk. It would be risky to grant access to the directory in which the files that are needed are present, as this will allow accesses to all files under that directory. Instead, we can allow access through a *trusted* method that presents a file access dialog box to a user, and gets explicit user approval before granting controlled access. (Note that one cannot trust the file dialog if it is part of the mobile code).

We briefly discuss how a policy such as this is difficult to express with the current Java framework. Note that the Java approach is based on the assumption that when multiple code sources are involved in a resource request, the access is allowed only if such access is permitted individually for each of the code sources. In this example, since we do not want to provide access to the resource for the untrusted code, it will be prevented from accessing the resource. A way around this problem in Java is for the trusted library to surround the resource access operations within a `doPrivileged` block, but this requires advance planning by the code producer of the library to anticipate what resource accesses are potentially sensitive and to enclose each such access within a `doPrivileged` block. On the other hand, the language proposed in this paper allows such decisions to be deferred until much later, and provides the flexibility to conceive and enforce such policies.

2.3 Limiting information flow

In many contexts, we are interested in limiting malicious applications from leaking sensitive information. While a comprehensive treatment of information flow requires approaches such as [11], that need to reason about internal structure of a program (such as assignment statements in a program), we can still express many

useful policies that limit information flow in terms of operations made by a program. Examples of such policies include:

- *No access to nonlocal networks after reading sensitive files.* Consider the case of a untrusted freeware web log analyzer. In this case, we may not want to permit access to arbitrary network sites after a mobile application reads the web server log files (this is to prevent leaking of sensitive information). However, we still permit access to local network sites (say, for the purpose of resolving domain names). Moreover, we may permit network access after reading files that are not considered particularly sensitive, e.g., icon files or fonts.
- *No access to create arbitrary files after reading confidential data.* Consider a freeware security scanner program that is downloaded from an untrusted site. In order to perform its task, the scanner will need to examine the permission settings on all files and contents of various files such as password files, boot-time scripts and configuration files. We can grant this access to the application provided we can ensure that any confidential information read by the application cannot leave the application in any manner (including, for instance, core dumps). We still want it to produce an output, so we may permit it to output information on the console and/or a specific log file.

This class of policies requires a language that can refer to sequences of operations and their arguments. Although Schneider's security automata based approach [4] allows operation sequencing, the arguments operations are not made available, so it becomes difficult to capture these policies.

2.4 Context-dependent policies

Finally, it may be necessary to include the application context while granting permissions to perform certain operations.

- *Allow access to certain operations only in the context of another operation.* For instance, consider the example of a web browser. We may be interested in giving permissions to some scripts/applets based on the domain of the URL being visited. Such permissions may include reading system properties and displaying popup windows. For example, we may only want allow scripts only from `www.redhat.com` to read our operating system version.

Note that, while some of these operations could be done in current browsers through *signed applets*, we are still limited by fact that we could allow an applet to perform all these operations in all cases of webpages and operations, or not at all. Using the existing models, we cannot parametrize the policy specific to some web pages/specific operations.

2.5 Desirable Features of a Policy Framework

Based on our discussion of policy examples, we state some desirable features of a policy specification and enforcement framework.

- *Flexibility to state policies in terms of any externally observable operation and its arguments.*
- *Ability to express policies involving temporal sequencing of operations.* The examples in the earlier section illustrate the importance of policies that specify temporal sequencing of operations. The need for this is seen in history-sensitive policies.
- *Modular specifications with precise and simple semantics.* The semantics must be straightforward and intuitive, matching the intention of the policy developer. There must be a way to modularize large policy specifications.
- *Efficient enforcement.* The language design must facilitate generation of efficient engines for enforcement of policies.

Currently existing policy frameworks do not offer all of these features.

3. Our approach

3.1 Overview

In our approach, security policies are represented using extended finite state automata (EFSA) are specified using a textual language based on regular expressions. Once a security policy is specified through this textual language, a *policy compiler* translates this policies to a EFSA based *policy engine*. The code that delivers security relevant relevant events to this engine constitutes the *runtime environment*. It is introduced in appropriate security relevant points of the piece of untrusted code through a *bytecode transformer*. When the transformed application violates the security policy, the policy engine takes appropriate remedial action like throwing a security exception.

Figure 1 illustrates our approach. The *offline* component consists of the *policy compiler* that generates policy engines from security policies. It also provides additional (runtime interception) information that is used to determine which of these operations (and which arguments to those operations) are relevant to the security policies. This additional information also specifies whether the security policy requires auxiliary information such as code source and thread id. The *load time* component consists of a *bytecode transformer*. It takes as input the untrusted code source and additional runtime interception information (from the policy compiler). It transforms the untrusted application such that code for the runtime environment is introduced at various points in the bytecode. Finally, the *runtime component* consists of the *policy engine* and the *runtime environment*. The runtime environment provides the mechanism for intercepting security relevant events and delivering them to the policy engine. The policy engine decides whether this event corresponds to a security violation, and if so, throws an appropriate exception.

3.2 Language for describing policies

Our policy language is based on our previous work in developing languages for expressing security-relevant behaviors of systems. In our language, security-relevant behavior of a program is modeled in terms of sequences of externally observable actions performed by the program. In the context of Java, such actions, called *events*, include method entries and exits, as well as exceptions. A security policy specifies constraints on the sequence of events that may be produced by a program. These constraints are captured by *extended finite state automata* (EFSAs). These automata, like conventional finite state automata, have states and transitions on pairs of states, and in addition, are augmented with *variables* along the transitions to store event arguments. In this section, we first illustrate the use of EFSAs in security policy descriptions of some of the examples that were introduced in Section 2. Following this, we provide a description of a text-based specification language for such policies.

Ability to create and manipulate temporary files. In this example, shown in Figure 2, *FileCreateOps* refers to operations that are

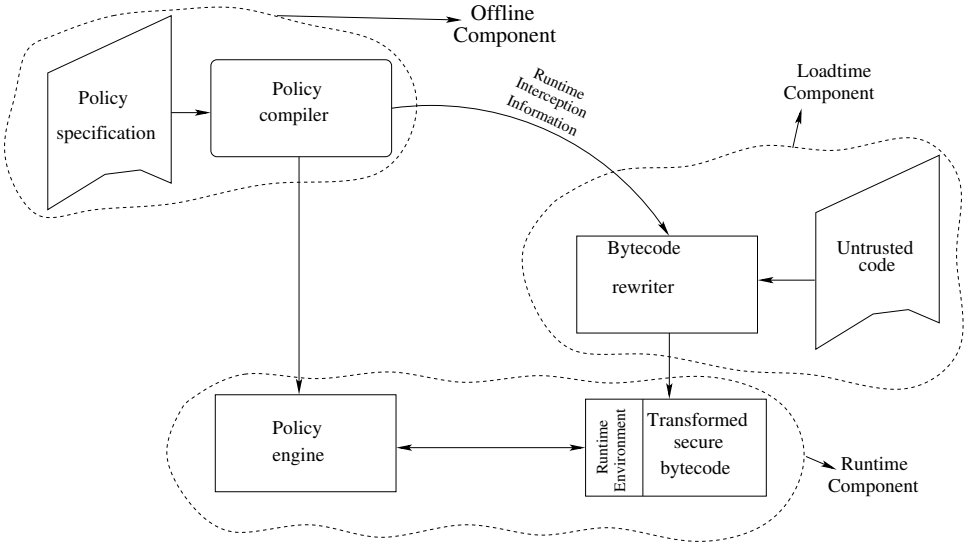

Figure 1: Our code transformation framework

used to create files, and *FileDeleteops* refers to file-deletion operations. In this automaton, the state `S0` records the files that have been created into the state variable *FileList*. Any operation that attempts to delete a file that is not present in *FileList*, involves the transition to state `S1`, where an appropriate exception is thrown.

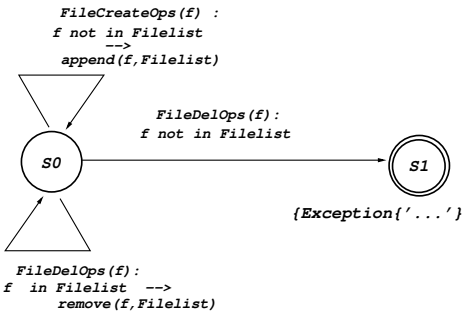

Figure 2: Temporary file handle policy

Limiting information flow. As illustrated in Figure 3, this EFSA describes a information flow policy. Once the application reads sensitive files and/or (the symbol || stands for disjunction) stands for system properties, the transition to state `S1` is taken. From state `S1`, if the application performs network operations, the transition to state `S2` is taken, where an exception is thrown.

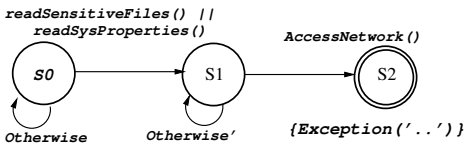

Figure 3: Policy on limiting information flow

Access to resources via trusted intermediaries. This policy is illustrated in Figure 4. (This policy illustrates the "trusted dialog boxes" example). Whenever the application accesses a local file, it is allowed to only do so through a trusted dialog box. After performing the initial action through the dialog-box, the transition to state `S1` is taken. In `S1`, operations to access local files are permitted. A transition back to state `S0` is made when the call to the trusted dialog box function returns. At this point, any access to local files will result in a transition to state `S2`, where an exception will be raised.

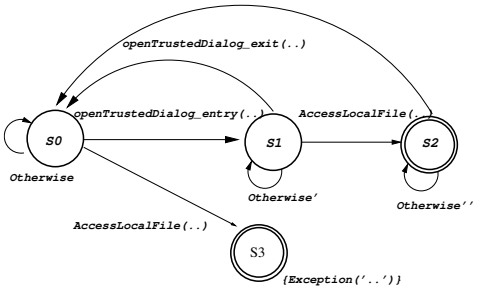

Figure 4: Trusted intermediary policy

Overview of Language. We use a textual language to specify policies, and these specifications are automatically transformed into EFSA based policies by a compiler. The constraints in our textual language are captured using a combination of constructs that resemble regular-expression based patterns and finite state machines. The key components of specifying rules in our language are described below. A more detailed description of the language may be found in [15].

Events. Events may be further classified as follows:

- *Primitive events:* In the context of Java, each method invocation corresponds to two events: one that corresponds to the invocation of the method, and another to return from the method. The arguments of the entry event include all of the method's

arguments at the point of call. The arguments to exit event include all of the method arguments at the point of return, plus the value of the return code from the method. In addition, there can be an event corresponding to each possible exception.

- *Abstract events:* Abstract events can be used to denote classes of primitive events, e.g., we may define a $fileModification\text{-}Ops$ as an event that corresponds to a set of events that modify file attributes. More generally, abstract events may denote any event pattern, and defined using the notation $event(args) = pat$. Event patterns are further described below.

Patterns. The simplest form of patterns, called a primitive patterns, capture the occurrence of a single event. It is of the form $e(x_1, ..., x_n)|cond$, where $cond$ is a boolean-valued expression on the event arguments $x_1, ..., x_n$, and state variables (which are further described below). Complex patterns capture sequencing relationships among events by using *sequencing operators.* Sequencing operators are similar to those used in regular expressions, but operate on events with arguments. We refer to our pattern language as regular expressions over events (REE) to indicate this relationship. The meaning of event patterns and the sequencing operators is best explained by the following definition of what it means for an *event history* H (a sequence of events observed at runtime) to match a pattern:

- *event occurrence:* $e(x_1, ..., x_n)|cond$ is satisfied by the event history $e(v_1, ..., v_n)$ if $cond$ evaluates to $true$ when variables $x_1, ..., x_n$ are replaced by the values $v_1, ..., v_n$.
- *sequencing:* $pat_1 \cdot pat_2$ is satisfied by an event history H of the form H_1H_2 provided H_1 satisfies pat_1 and H_2 satisfies pat_2.
- *repetition:* $pat*$ is satisfied by $H_1H_2 \cdots H_n$ iff H_i satisfies $pat, \forall\ 1 \leq i \leq n$.
- *alternation:* $pat_1||pat_2$ is satisfied by H if either pat_1 or pat_2 is satisfied by H.
- *conjunction:* $pat_1 \wedge pat_2$ is satisfied by H iff both pat_1 and pat_2 are satisfied by H.

When a variable occurs multiple times within a pattern, an event history satisfies the pattern only if the history instantiates all occurrences of the variable with the same value. For instance, the pattern $e_1(x) \cdot e_2(x)$ is not satisfied by the event history $e_1(a)e_2(b)$, but is satisfied by $e_1(a)e_2(a)$.

We use $!e|cond$ to denote the nonoccurrence of an event e, or the occurrence of e where the condition $cond$ is violated.

Rules. Our policy language allows response actions to be associated with policies using rules of the form $pat \rightarrow action$. The $action$ component is executed whenever *a suffix of the event history matches* pat. In general, the reaction component consists of a sequence of statements, each of which is either an assignment to a state variable, or invocation of a support function provided by the runtime system. Such support functions may be used for a variety of purposes such as denying resource access, throwing security-related exceptions or terminating programs or threads.

3.3 Implementation

We describe the implementation of the three components of our framework in the following section.

Compilation of Security Policies. Efficient matching of security policy rules is critical for the performance of our runtime engines. Our approach for solving this problem is based on compiling the patterns into an EFSA, in a manner analogous to compiling regular expressions into finite-state automata. EFSA are simply standard finite state automata (FSA) that are augmented with a fixed number of *state variables,* each capable of storing values such as integers, strings, etc. Every transition in the EFSA is associated with an event, an enabling condition involving the event arguments and state variables, and a set of assignments to state variables. The final states of the EFSA may be annotated with actions, which, in our system, correspond to the reactions given in our rules. For a transition to be taken, the associated event must occur and the enabling condition must hold. When the transition is taken, the assignments associated with the transition are performed.

An EFSA is normally nondeterministic. The notion of acceptance by a nondeterministic EFSA (NEFA) is similar to that of an NFA. A deterministic EFSA (DEFA) is an EFSA in which at most one of the transitions is enabled in any state of the EFSA.

We have shown that translating a NEFA to a DEFA can result in an unacceptable increase in the size of the automaton. Therefore we have developed a new approach that is based on translating NEFA into a *quasi-deterministic extended finite state automata* (QEFA). QEFA eliminates most of the sources of nondeterminism that are present in the NEFA, while still ensuring that their sizes are acceptable. A complete treatment of QEFA and the compilation algorithm can be found in [15].

Runtime Environment. The runtime environment is responsible for intercepting and forwarding events to the policy engine. There are two basic approaches for implementing the runtime system. One approach is to modify the JVM implementation so that relevant method calls are forwarded to the policy engine. This approach has the benefit of low overhead for interception, but has the drawback that it is highly dependent on the internals of the JVM. This makes the approach hard to implement and potentially error-prone. Moreover, the internals may change across JVM versions, thus requiring the runtime environment to be reimplemented for each JVM release.

An alternative approach is to integrate the code for method interception right into the program to be monitored[1]. Note that the class file format is standardized and does not change across JVM releases. This factor decouples the runtime implementation from JVM releases, and hence makes it portable across different JVM versions. The drawback is that the overhead for method interception will be somewhat higher than the JVM modification approach, due to the fact that the interception code itself would be implemented using several JVM instructions. In addition, there is a additional cost involved in byte code rewriting. Given that there are already significant startup costs associated with starting up Java programs, such class loading, byte-code verification and Just-In-Time compilation, we believe that the additional byte code rewriting overhead will not substantially alter the overall loading time. Thus, portability is the more important concern for our project, and hence we have chosen byte-code rewriting approach.

[1]Such an approach is difficult in a type unsafe language such as C due to the fact that a malicious piece of code has several ways to circumvent the checking code integrated within itself. This is not a problem in a type-safe language such as Java.

```
    void network_write(FileOutputSteream f){
        byte[] b;
        .......
        monitor.deliver_event(write_entry,
                              f.getClass(), f, b);
        f.write(b);
        monitor.deliver_event(write_exit,
                              f.getClass(), f, b);
    }
```

Figure 5: Caller modification

```
Class X{
  void network_write( BufferedOutputSteream f){
    byte[] b;
    .......
    f.write(b); // No transformation here
  }
}
class BufferedOutputStream{
  public void write(byte[] b){
    monitor.deliver_event(write_entry,
                          BufferedOutputStream, this, b);
    this.original_write(b);
    monitor.deliver_event(write_exit,
                          BufferedOutputStream, this, b);
  }
}
```

Figure 6: Callee modification

3.4 Byte-code Rewriting.

In our approach, events are delivered to the policy engine using a `deliver_event` method. If `monitor` denotes the policy engine, the effect of introducing these calls at the source code level is shown in Figures 5 and 6. In these figures, a function named `network_write` is being transformed. In Figure 5, the transformation is done on the caller code, whereas in Figure 6, the transformation takes place in the callee code.

In the caller transformation, some preliminary analysis of the byte-code is needed to ensure that security checks are indeed performed before every security-sensitive method call. This requires, for instance, that there be no control transfer statements that can skip the `deliver_event` statement. On the other hand, no such analysis is required for callee transformation. Another difficulty with caller transformation is that exact type information is unavailable at the call site, and it hence requires us to use expensive runtime operations such as `getClass` to get this information. In the callee transformation case, we know the exact type information at rewriting time.

There are also some benefits to caller transformation. First, it is possible to incorporate information about calling context — for instance, if we wanted to use the codebase information (which identifies the principal that originated a piece of mobile code) in security checks, this information can be passed in as an additional argument to `deliver_event`. In the callee transformation, we no longer have the codebase information about the caller, but only about the callee. However, the callee codebase information is not very useful for making such access control decisions — for instance, the callee in the example corresponds to system codebase, which is usually given unlimited privileges. A second difficulty concerns modification of system classes. JVM implementation prohibits overriding of the class loader for system classes. Therefore, in practice, the system classes have to be rewritten through an offline process, and the results stored on the disk. Since we do not want to store many different versions of the system libraries, it becomes impractical to try to tailor the system classes with respect to different security policies. The net result would be a rewrite that results in every system method to be intercepted, thereby increasing overheads. For this reason, we have currently chosen to use caller transformation. It may turn out later that this cost is not any higher than the savings achieved by avoiding class lookup operations required in the caller transformation. In that case, our implementation choice will likely change.

The following issues need to be addressed in byte-code rewriting:

- *Exceptions:* As mentioned earlier, we would like to treat Java exceptions as events, in much the same way that we treat method invocations and exits. However, one has to be careful when exceptions arise in the code. Note that when exceptions arise in a method, the method invocation is not completed. The policy engine has to be notified that such method entry events will not have corresponding method exit events. We do this by associating an additional parameter to method invocations that capture the call-depth information. When an entry event at dept k is followed by an exception event at depth $k' < k$ with no intervening entry or exit events, then the policy engine knows that all of the recent entry events with depth $\geq k$ have been aborted due to the exception. The call depth information can be sent in as an additional piece of information to the `deliver_event` function.
- *Native methods:* Java supports running of native methods via the Java Native interface. While native methods are a convenient mechanism for running platform specific code, and for increasing overall system performance, they make security policy enforcement very difficult. One can address security in the presence of native methods by interception of system calls as done in [14], but we do not address this problem in this paper.
- *Threads:* Our current (preliminary) implementation does not handle multi-threaded programs. To handle them, one needs to change `deliver_event` so that the id of the current thread is passed along as an additional parameter. The thread id can be used in policies, which would enable us to express a range of policies that handle threads well. For instance, we can have policies that treat each thread uniformly, enforcing properties on each one independent of the actions taken by other threads. It is also possible to ignore a thread altogether, and treat all of the events as a single sequence. Finally, it is possible to capture properties that involve multiple threads simultaneously — this would enable us to express security policies that address aspects such as synchronization and race conditions.

4. Performance results

In this section, we discuss our experimental results obtained from our prototype implementation. All the results presented in this section were obtained on a machine running on a 1.4GHz Pentium 4 processor with 512 MB of RAM. The JVM used for these experiments was SUN Microsystem's Java Development Kit (JDK) version 1.3.1 running on Linux operating system with the 2.4 kernel. The main objective of our experiments was to measure the following costs:

- *Interception overhead:* This is the cost associated with introduction of additional method calls that wrap the original method calls in the application.

Application	LimitWrite	DeleteOnlyOwned	FileOpens	TarSpecific
BigLoop	7% (56.5s)	17% (18.45s)	10% (11.5s)	NA
Jtar	5% (2.01s)	7% (2.01s)	5% (2.01)	10.5%(2.01s)

Figure 7: Policy enforcement costs

- *Bytecode modification overhead:* An overhead cost is incurred as part of parsing the classfile and rewriting it.
- *Monitoring code overhead:* This is the cost associated with the execution of the code that is part of the monitoring automaton.

We separate the costs into these categories for closer inspection of the costs incurred in these stages. We briefly discuss these costs and show the total overhead of enforcing some example policies.

4.1 Interception overhead

The interception overhead is the additional time introduced in the application execution time through the additional bytecode method wrappers. For every method call M that is of interest to the security policy, the calling sequence makes two more calls. The first call is made before the execution of the method call of interest, and the other call is made after the method exit. These calls are used to make transitions in the automaton corresponding to the policy engine.

To measure the interception overhead we constructed a program that performed a method call in a loop iterating 10^7 times. We compiled it along with a policy specifications that contained null action statements, and no state variables, thus leaving the monitor with just the interception capability. We obtained an overhead of less than 5% of the program running time.

4.2 Bytecode modification overhead

Currently, we use the Byte Code Engineering Library (BCEL) [2, 1] to do the transformations. This tool is meant for offline transformation of bytecodes. The toolkit offers convenience and flexibility in transforming bytecodes, but is particularly not efficient. For instance we incur an overhead of .650s for transforming a 4800 byte classfile. We are currently developing a hand-crafted program to do the bytecode rewriting. The bytecode transformation is straightforward, and has to make a linear pass on the classfile to insert the event delivery instructions. We do not anticipate the overheads of this operation to be significant, especially in comparison to classloading process and bytecode verification.

4.3 Overhead on various policies

The following policies were implemented and we measured the performance results for implementing these policies.

- **LimitWrite**. This is a policy that enforces a constraint on filesystem usage. In this policy, we impose a 10MB limit on the amount of data that could be written by an application on the filesystem.
- **DeleteOnlyOwned**. This policy states that an application can overwrite/delete only those files created by the application.
- **TarSpecific**. This is a policy on the jtar (description of this application is given below) application and enforces file access permissions and enforces a disk space usage limit on the files written by the application.
- **FileOpens**. This is a conventional access check policy where the application is checked for access rights.

We have implemented and have tested these policies on the following programs.

- *BigLoop*, a toy application that repeats one of the following operations about 10^6 times: (a) open a new file in the `/tmp` directory and write 100 bytes to it, (b) open three files and delete files from previous iterations, or (c) write 100 bytes to a file.
- *Jtar*, an application that we downloaded from `www.ice.com`, that is a Java clone of the conventional Unix tar facility. We tested this application against all the policies mentioned above. The program used a filesystem that consisted of about 3800 small files (of size less than 1000 bytes).

Figure 7 illustrates the overheads we incur in enforcing the policies listed above on the *BigLoop* and *Jtar* applications. The results are shown as percentage overheads to the cost without enforcing these policies, which is given in parentheses (in seconds).

Comparison with Java Security. We tested the performance of Java enabled with the Security Manager for the *FileOpens* policy. (This is the only policy that could be implemented in the security manager without any modification to Java.) The experiment resulted in an significant overhead of over 120%, as the stack inspection operation is relatively an expensive operation.

We note, however, that a direct comparison of our results with Java is not very meaningful — Java security policy is capable of dealing with multiple codebases, whereas our policy, as implemented, deals with only one codebase. The comparison is presented to establish two points. First, our policies incur acceptable overheads. Second, Java's approach of predefining all of the security-relevant operations makes it difficult to perform optimizations based on security policies of interest. For instance, although we may be interested only in file open operations, we still incur interception overhead on many other operations such as network operations and other runtime operations. The predefinition also means that we cannot implement policies that require interception of other operations, e.g., the *LimitWrite* policy cannot be captured as it requires interception of write operations.

Comparison with Naccio. Evans [6], in describing their experimental results, describes the *LimitWrite* experiment with an earlier version of the same *Jtar* program. They present a performance penalty of close to 25%. We have to be careful in direct comparison of results as the Java VM has passed through a few revisions since the publication of their results. The only thing we can infer from the results is that our performance is competitive to the performance of their approach.

5. Related work

Model carrying code (MCC) [13], presented in NSPW 2001, is our general framework for ensuring the security of mobile code. This approach enables a mobile-code consumer to understand and formally reason about what a piece of mobile code can do; check if the actions of the code are compatible with his/her security policies and if so, execute that code. This framework has several components including *model generation*, *security policies*, *consistency resolution* and *runtime monitoring*. Our research in expressive security policies for the MCC framework has yielded the results presented in this paper.

In Java [7, 8], the permissions available for programs from a code source are specified through a *security policy*. The security policy assigns permissions to various *code sources*. At runtime, programs are checked for compliance with the security policy through runtime monitoring. The implementation of security checking is done by a technique known as *stack inspection* [16]. When a piece of code performs a security relevant operation, the effective set of permissions that correspond to the code sources that are in the execution sequence is computed by inspecting the runtime stack. The operation is allowed only if it is implied by this set of permissions.

The security policy is a list of entries mapping code sources to permissions. There are some disadvantages with having such a simple policy language and enforcement scheme. The policy language is not very expressive, and this precludes specification of some interesting classes of policies, that have been described earlier in this paper. In addition, security checks are scattered all over the Java API implementation which is of the order of several thousand lines, rather than for only those classes which are of interest to the policy. Also, the set of attributes that could be inspected in the security checks are fixed by the policy specification language and the JDK implementation, and is rigid and does not offer flexibility.

Naccio [6] is an interesting example of a system that allows convenient specifications of security policies as abstract resource manipulations. The system then generates new system libraries that includes code checking necessary to enforce the security policy. In this sense, our model is closer to the Naccio model than the Java model. The main difference between the approach presented in Naccio and our approach is in the policy language. Our language supports specification of policies that check not just invariant properties, but several other interesting classes as well.

Schneider [12] presents a formal treatment of runtime monitoring mechanisms that work by monitoring steps of a target and terminating execution that would violate the policy being enforced. This class of mechanisms is termed (Execution Monitoring) EM. [4] presents in-line reference monitors as part of the Security Automata based Software Isolation (SASI) approach. The main difference between their approach and ours is that our approach has the ability to capture and remember argument values to events. This ability, as illustrated in the examples section, makes an important difference in the classes of policies that could be enforced. In [5] they implement Java stack inspection based approach based on bytecode rewriting and show performance measurements. Although our work adopts a similar bytecode modification approach for implementation, our focus is mainly on specifying expressive policies.

Edjlali et al. [3] describe a history-based access control mechanism for Java and provide several motivating examples for the use of such policies. The main contribution of their work is in providing a framework for implementing such policies in Java. Their implementation is related to the set of events that are identified by system libraries. Our focus is on developing a high-level language to declaratively specify such policies on events such as arbitrary function calls. Interception of arbitrary function calls is crucial for enforcing some of the policies (such as the *trusted intermediary* policy) that were discussed in our examples section.

One area that we have not discussed in detail in this paper concerns policies that control information flow. A policy which allows no network operations after local file reads is an example of an information flow policy. Other information flow policies discuss what a third party observer can deduce about the system by noticing its external behavior. Runtime monitoring approaches do not always capture all cases of information flow [12]. Myers [11] discusses a static analysis based approach to ensure safe information flow through labels.

Complimentary to our approach, there has been a considerable amount of work on policy languages that have focused on ease of use in policy specifications. Hoagland [10] describes a simple graphical policy description language that adds ease of use to security policy specification. Hauswirth [9] describes a framework for specifying policies that is focused on ease of use of specifications through higher level abstractions and graphical tools.

6. Conclusion

In this paper, we have presented our case for providing flexible policy specifications that address the need for empowering mobile code, yet mitigating the security risks. We have presented a few example scenarios that motivate this discussion. We also presented an expressive policy language that supports fine grained policy specification based on observable event sequences. In addition, we have presented an implementation in the context of Java and have discussed issues related to the implementation. Finally, we have presented preliminary results from our implementation.

7. REFERENCES

[1] *BCEL API Documentation available at http://bcel.sourceforge.net/docs/index.html*.

[2] M. Dahm. Byte code engineering. In *In Proceedings of JIT 99*, 1999.

[3] G. Edjlali, A. Acharya, and V. Chaudhary. History based access control for mobile code. In *Proceedings of ACM Computer and Comunications Security conference, 1998*.

[4] U. Erlingsson and F. B. Schneider. Sasi enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigm Workshop Ontario, Canada*, 1999.

[5] U. Erlingsson and F. B. Schneider. Irm enforcement of java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Oakland, California, May 2000.

[6] D. Evans and A. Tywman. Flexible policy directed code safety. In *Proceedings of the 1999 IEEE conference on Security and Privacy*, 1999.

[7] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Number ISBN: 0201310007. Addison-Wesley Pub Co, 1998.

[8] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. 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.

[9] M. Hauswirth, C. Kerer, and R. Kurmanowytsch. A secure execution framework for java. In *Proceedings of the 7th ACM Conference on Computer and Communtications Security*, pages 43–52, 2000.

[10] J. Hoagland, R. Pandey, and K. Levitt. Specifying security policies using a graphical approach. Technical report, University of California,, 1999.

[11] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering Methodology*, 1999.

[12] F. B. Schneider. Enforceable security polcies. Technical report, Cornell University, 1999.

[13] R. Sekar, C. Ramakrishnan, I. Ramakrishnan, and S. Smolka. Model carrying code: A new paradigm for mobile code security. In *Proceedings of the New Security Paradigms Workshop*, 2001.

[14] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the USENIX Security Symposium*, 1999.

[15] P. Uppuluri. Pattern matching based intrusion detection systems. Technical report, Computer Science, StonyBrook, 2001.

[16] D. S. Wallach and E. W. Felten. Understanding java stack inspection. In *1998 IEEE Symposium on Security and Privacy*, pages 52–63, 1998.