History-based Access Control for Mobile Code

Guy Edjlali
Dept. of ECE
Wayne State University
Detroit, MI 48202

Anurag Acharya
Dept. of Computer Science
University of California
Santa Barbara, CA 93106

Vipin Chaudhary
Dept. of ECE
Wayne State University
Detroit, MI 48202

Abstract

In this paper, we present a *history-based* access-control mechanism that is suitable for mediating accesses from mobile code. The key idea behind history-based access-control is to maintain a selective history of the access requests made by individual programs and to use this history to improve the differentiation between safe and potentially dangerous requests. What a program is allowed to do depends on its *own* behavior and identity in addition to currently used discriminators like the location it was loaded from or the identity of its author/provider. History-based access-control has the potential to significantly expand the set of programs that can be executed without compromising security or ease of use. We describe the design and implementation of *Deeds*, a history-based access-control mechanism for Java. Access-control policies for *Deeds* are written in Java, and can be updated while the programs whose accesses are being mediated are still executing.

1 Introduction

The integration of mobile code with web browsing creates an access-control dilemma. On one hand, it creates a social expectation that mobile code should be as easy to download and execute as fetching and viewing a web page. On the other hand, the popularity and ubiquity of mobile code increases the likelihood that malicious programs will mingle with benign ones.

To reassure users about the safety of their data and to keep the user interface simple and non-intrusive, systems supporting mobile code have chosen to err on the side of conservatism and simplicity. Depending on its *source*, mobile code is partitioned into *trusted* and *untrusted* code. Code is considered trusted if it is loaded from disk [9, 12] or if it is signed by an author/organization deemed trustworthy by the user [12, 30]. Untrusted code is confined to a severely restricted execution environment [9] (eg, it cannot open local files or sockets, cannot create a subprocess, cannot initiate print requests etc); trusted code is either given access to all available resources [30] or is given selective access based on user-specified access-control lists [12].

For the programs considered untrusted, these mechanisms can be overly restrictive. Many useful and safe programs, such as a well-behaved editor applet from a lesser-known software company, cannot be used since it cannot open local files. In addition, to implement new resource-sharing models such as *global computing* [6] all communication has to be routed through brokers. This significantly limits the set of problems that can be efficiently handled by such models. For programs considered trusted, these models can be too lax. Errors, not just malice aforethought, can wipe out or leak important data. Combined with a suitable audit trail, *signed programs* [12] do provide the ability to take legal recourse if need be.

In this paper, we present a *history-based* access-control mechanism that is suitable for mediating accesses from mobile code. The key idea behind history-based access-control is to maintain a selective history of the access requests made by individual programs and to use this history to improve the differentiation between safe and potentially dangerous requests. What a program is allowed to do depends on its *own* identity and behavior in addition to currently used discriminators like the location it was loaded from or the identity of its author/provider. History-based access-control has the potential to significantly expand the set of programs that can be executed without compromising security or ease of use. For example, consider an access-control policy that allows a program to open local files for reading as long as it has not opened a socket and allows it to open a socket as long as it has not opened a local file for reading. Irrespective of the source of the program, such a policy can ensure that no disk-resident data will be leaked. Strictly speaking, this is true iff it is possible to intercept *all* access requests being made on behalf of the program – the requests made by itself as well as the requests made on its behalf. The technique we present in this paper is able to intercept all requests.

We first present some examples of history-based access-control policies. Next, we discuss issues that have to be resolved for implementing history-based access-control mechanisms. In section 3, we describe *Deeds*,[1] an implementation of history-based access-control for Java programs. Access-control policies for *Deeds* are written in Java, and can be installed, removed or modified while the programs whose accesses are being mediated are still executing. *Deeds* requires policies to adhere to several constraints. These constraints are checked either at compile-time by the Java compiler or at runtime by the *Deeds* policy manager. We illustrate the

[1]Your deeds determine your destiny :)

5th Conference on Computer & Communications Security
San Francisco CA USA
Copyright ACM 1998 1-58113-007-4/98/11...$5.00

operation of the *Deeds* user interface using snapshots. In section 4.4, we examine the additional overhead imposed by *Deeds* using micro-benchmarks as well as real programs. History-based access-control is not specific to Java or to mobile code. It can be used for any system that allows interposition of code between untrusted programs and protected resources. In section 5, we discuss how a system similar to *Deeds* can be used to mediate accesses to OS resources from native binaries. We conclude with a description of related work and the directions in which we plan to extend this effort.

2 Examples

One-out-of-k: Consider the situation when you want to allow only those programs that fall into well-marked equivalence classes based on their functionality and behavior. For example, you want to allow only programs that provide just the functionality of a browser or an editor or a shell. A browser can connect to remote sites, create temporary local files in a user-specified directory, read files that it has created and display them to the user. An editor can create local files in user-specified directories, read/modify files that it has created, and interact with the user. It is not allowed to open sockets. A shell can interact with the user and can create sub-processes. It cannot open local files, or connect to remote sites. This restriction can be enforced by a history-based access-control policy that:

- allows a program to connect to a remote site if and only if it has neither tried to open a local file that it has not created, nor tried to modify a file it has created, nor tried to create a sub-process;
- allows a program to open local files in user-specified directories for modification if and only if it has created them, and it has neither tried to connect to a remote site nor tried to create a sub-process.
- allows a program to create sub-processes if and only if it has neither tried to connect to a remote site nor tried to open a local file.

In effect, each program is dynamically classified into one of three equivalence classes (browser-like, editor-like or shell-like) based on the sequence of requests it makes. Once a program is placed in a class, it is allowed to access only the resources that are permitted to programs in that class.

Keeping out rogues: Consider the situation where you want to ensure that a program that you once killed due to inappropriate behavior is not allowed to execute on your machine. This restriction can be enforced, to some extent, by a history-based access-control policy that keeps track of previous termination events and the identity of the programs that were terminated.

Frustrating peepers: Consider the situation where you want to allow a program to access only one of two relations in a database but not both. One might wish to do this if accessing both the relations may allow a program to extract information that it cannot get from a single relation. For example, one might wish to allow programs to access either a relation that contains the date and the name of medical procedures performed in a hospital or a relation that contains the names of patients and the date they last came in. Individually, these relations do not allow a program to deduce information about treatment histories of individual patients. If, however, a program could access both relations, it could combine the relations to acquire (partial) information about treatment histories for individual patients. This example can be seen as an instance of the Chinese Wall Policy [4]. To block the possibility of a hostile *site* being able to deduce the same information from data provided by two different programs it provides, programs that have opened a socket are, thereafter, not allowed to access sensitive relations and programs that have accessed one of the sensitive relations are, thereafter, not allowed to open sockets.

Slowing down hogs: Consider the situation where you want to limit the rate at which a program connects to its home site. One might wish to do this, for example, to eliminate a form of denial of service where a program repeatedly connects to its home site without doing anything else. This can be enforced by a history-based access-control policy that keeps track of the timestamp of the last request. It allows only those requests that occur after a threshold period.

3 Issues for history-based access-control

Identity of programs: Associating a content-based, hard-to-spoof identity with a program is a key aspect of history-based access-control. That is, given any program, it should be hard to design a substitute program for whom the identity computation generates the same result. An important point to note is that the code for a mobile program can come from multiple sources (from local disk, from different servers on the network, etc). The identity mechanism should associate a single identity with *all the code* that is used by a program. This is important to ensure that a malicious program cannot assume the identity of another program by copying parts or all of the program being spoofed.

Efficient maintenance of request-histories: Wallach et al [38] mention that a collection of commonly used Java workloads require roughly 30000 crossings between protection domains per CPU-second of execution. Given this request-frequency, it is imperative that access-control checks on individual requests be fast. Simple logging-based techniques are likely to be too expensive. Fortunately, the request-history for many useful policies can be summarized. For example, the request-history for a policy that allows a program to open local files for reading if it has not opened a socket and allows it to open a socket if it has not opened a local file for reading can be summarized by a pair of booleans – one that records if the program has ever opened a socket and the other that records if it has ever opened a local file.

Persistence of policies and histories: Persistent request-histories are required to block attacks that consist of running a sequence of programs each of which makes requests that are allowed by the access-control policy but when taken as a whole, the complete sequence of requests violates the constraints the policy tries to enforce.

Grouping privileges: History-based mechanisms can provide extremely fine-grain access-control. Not only is it possible to control accesses to individual objects/resources, it is possible to differentiate between different patterns of accesses. While this allows us to expand the set of programs that can be executed safely, this level of flexibility can be hard to deal with. Requiring users to specify their preferences at this level of detail is likely to be considered intrusive and therefore ignored or avoided. This problem can alleviated to some extent by grouping acceptable patterns of program behavior and assign intuitive names to these patterns. For example, a policy that allows a program to open no sockets, open local files for reading, to create local files in a user-specified directory and to open a local file for modification only if it has been been created by itself. This set of restrictions allow a simple editor to be executed and can, jointly, be referred to as the *editor* policy.

Composition and fail-safe defaults: History-based access-control policies encode acceptable patterns of program behavior. Different classes of programs might have different behaviors all of which are acceptable to the user. It is, therefore, important to provide automatic composition of multiple policies. An important point to note here is that, by default, the access-control mechanism should be fail-safe [32] – potentially dangerous accesses should be denied unless explicitly granted.

4 Deeds: a history-based security manager for Java

In this section we describe the design and implementation of *Deeds*, a history-based access-control mechanism for Java. We first describe the architecture of *Deeds*. Next, we describe its current implementation and its user interface. In section 4.4, we examine the performance of the mechanisms provided by *Deeds*.

4.1 Architecture

In this subsection, we describe the architecture of *Deeds*. We focus on the central concepts of the *Deeds* architecture: secure program identity and security events.

4.1.1 Program identity

The *Deeds* notion of the identity of a program is based on *all* the downloaded code reachable during its execution. To achieve this, *Deeds* performs static linking on downloaded programs, fetching all non-local code that might be referenced. Local libraries that are part of the language implementation (e.g `java.lang` for Java, `libc` for C) are linked in as shared libraries; a separate copy of non-system-library code is downloaded for every application that uses it.

Deeds concatenates all non-system-library code for a downloaded program and uses the SHA-1 algorithm [34] to compute a name for it. SHA-1 belongs to the group of algorithms known as secure hash functions [31, 34] which take an arbitrary sequence of bytes as input and generate a (relatively) short digest (160-bits for SHA-1). These functions are considered secure because it is computationally hard to construct two byte-sequences which produce the same digest. In addition, the requirement, in this case, that the byte-sequences being compared should represent valid programs increases the difficulty of constructing a malicious program with the same name as a benign one.

In addition to allowing a secure hash to be computed, static linking of downloaded code has other advantages. First, having all the code available allows Just-in-Time compilers to perform better analysis and generate better code. Second, it removes potential covert channels which occur due to dynamic linking – the pattern of link requests can be used to pass information from a downloaded program to the server(s) that is (are) contacted for the code to be linked.

Note that Java allows programs to dynamically load classes. For such programs, it is not possible, in general, to statically determine the set of classes that might be referenced during the execution of the program. *Deeds* rejects such programs and does not allow them to execute.

4.1.2 Security events and handlers

A *Deeds* security event occurs whenever a request is made to a protected resource. Examples of security events include request to open a socket, request to open a file for reading, request to create a file, request to open a file for modification etc. The set of security events in *Deeds* is not fixed. In particular, it is not limited to requests for operating-system resources. Programmers can associate a security event with any request they wish to keep track of or protect.

Handlers can be associated with security events. Handlers perform two tasks: they maintain an event-history and check whether it satisfies one or more user-specified constraints. If any of the constraint fails, the handler raises a security-related exception. For example, a handler for the security event associated with opening a socket can record whether the program currently being executed has ever opened a socket. Similarly, a handler for the security event associated with opening a file for reading can record whether the program has opened a file for reading.

Multiple handlers can be associated with each event. Handlers maintain separate event-histories. The checks they perform are, in effect, composed using a "consensus voting rule" – that is, one negative vote can veto a decision and at least one positive vote is needed to approve. In this context, a request is permitted to continue if and only if at least one handler is present and none of the handlers raises an exception.

Access-control policies consist of one or more handlers grouped together. The handlers belonging to a single policy maintain a common history and check common constraints. For example, the *editor* policy mentioned earlier would consist of four handlers:

- a handler for socket-creation that records if a socket was ever created by this program. It rejects the request if a file has been opened by this program (for reading or writing).
- a handler for file-creation that associates a creator with each file created by a downloaded program. If the file is to be created in a directory that is included in a list of user-specified directories, it allows the request to proceed. Else, it rejects the request.
- a handler for open-file-for-read that records if a file was ever opened for reading by this program. It rejects the request if a socket has been created by this program.
- a handler for open-file-for-modification that records if a file was ever opened for writing by this program. It rejects the request if a socket has been created by this program or if the file in question was not created by this program.

Deeds allows multiple access-control policies to be simultaneously active. Policies can be installed, removed, or modified during execution. A policy is added by attaching its constituent handlers to the corresponding events. For example, the *editor* policy would be added by attaching its handlers respectively to the socket-creation event, the file-creation event, the open-file-for-read event and the open-file-for-modification event. Policies can be removed in an trusive and therefore ignored or avoided. This problem can alleviated to some extent by grouping acceptable patterns of program behavior and assign intuitive names to these patterns. For example, a policy that allows a program to open no sockets, open local files for reading, to create local files in a user-specified directory and to open a local file for modification only if it has been been created by itself. This set of restrictions allow a simple editor to be executed and can, jointly, be referred to as the *editor* policy.

Composition and fail-safe defaults: History-based access-control policies encode acceptable patterns of program behavior. Different classes of programs might have different behaviors all of which are acceptable to the user. It is, therefore, important to provide automatic composition of multiple policies. An important point to note here is that, by default, the access-control mechanism should be fail-safe [32] – potentially dangerous accesses should be denied unless explicitly granted.

4 Deeds: a history-based security manager for Java

In this section we describe the design and implementation of *Deeds*, a history-based access-control mechanism for Java. We first describe the architecture of *Deeds*. Next, we describe its current implementation and its user interface. In section 4.4, we examine the performance of the mechanisms provided by *Deeds*.

4.1 Architecture

In this subsection, we describe the architecture of *Deeds*. We focus on the central concepts of the *Deeds* architecture: secure program identity and security events.

4.1.1 Program identity

The *Deeds* notion of the identity of a program is based on *all* the downloaded code reachable during its execution. To achieve this, *Deeds* performs static linking on downloaded programs, fetching all non-local code that might be referenced. Local libraries that are part of the language implementation (e.g `java.lang` for Java, `libc` for C) are linked in as shared libraries; a separate copy of non-system-library code is downloaded for every application that uses it.

Deeds concatenates all non-system-library code for a downloaded program and uses the SHA-1 algorithm [34] to compute a name for it. SHA-1 belongs to the group of algorithms known as secure hash functions [31, 34] which take an arbitrary sequence of bytes as input and generate a (relatively) short digest (160-bits for SHA-1). These functions are considered secure because it is computationally hard to construct two byte-sequences which produce the same digest. In addition, the requirement, in this case, that the byte-sequences being compared should represent valid programs increases the difficulty of constructing a malicious program with the same name as a benign one.

In addition to allowing a secure hash to be computed, static linking of downloaded code has other advantages. First, having all the code available allows Just-in-Time compilers to perform better analysis and generate better code. Second, it removes potential covert channels which occur due to dynamic linking – the pattern of link requests can be used to pass information from a downloaded program to the server(s) that is (are) contacted for the code to be linked.

Note that Java allows programs to dynamically load classes. For such programs, it is not possible, in general, to statically determine the set of classes that might be referenced during the execution of the program. *Deeds* rejects such programs and does not allow them to execute.

4.1.2 Security events and handlers

A *Deeds* security event occurs whenever a request is made to a protected resource. Examples of security events include request to open a socket, request to open a file for reading, request to create a file, request to open a file for modification etc. The set of security events in *Deeds* is not fixed. In particular, it is not limited to requests for operating-system resources. Programmers can associate a security event with any request they wish to keep track of or protect.

Handlers can be associated with security events. Handlers perform two tasks: they maintain an event-history and check whether it satisfies one or more user-specified constraints. If any of the constraint fails, the handler raises a security-related exception. For example, a handler for the security event associated with opening a socket can record whether the program currently being executed has ever opened a socket. Similarly, a handler for the security event associated with opening a file for reading can record whether the program has opened a file for reading.

Multiple handlers can be associated with each event. Handlers maintain separate event-histories. The checks they perform are, in effect, composed using a "consensus voting rule" – that is, one negative vote can veto a decision and at least one positive vote is needed to approve. In this context, a request is permitted to continue if and only if at least one handler is present and none of the handlers raises an exception.

Access-control policies consist of one or more handlers grouped together. The handlers belonging to a single policy maintain a common history and check common constraints. For example, the *editor* policy mentioned earlier would consist of four handlers:

- a handler for socket-creation that records if a socket was ever created by this program. It rejects the request if a file has been opened by this program (for reading or writing).
- a handler for file-creation that associates a creator with each file created by a downloaded program. If the file is to be created in a directory that is included in a list of user-specified directories, it allows the request to proceed. Else, it rejects the request.
- a handler for open-file-for-read that records if a file was ever opened for reading by this program. It rejects the request if a socket has been created by this program.
- a handler for open-file-for-modification that records if a file was ever opened for writing by this program. It rejects the request if a socket has been created by this program or if the file in question was not created by this program.

Deeds allows multiple access-control policies to be simultaneously active. Policies can be installed, removed, or modified during execution. A policy is added by attaching its constituent handlers to the corresponding events. For example, the *editor* policy would be added by attaching its handlers respectively to the socket-creation event, the file-creation event, the open-file-for-read event and the open-file-for-modification event. Policies can be removed in an analogous manner by detaching the constituents handlers from the associated events.

Deeds allows policies to be parameterized. For example, a policy that controls file creation can be parameterized by the directory within which file creation is allowed. Policies that are already installed can be modified by changing their parameters. This allows users to make on-the-fly changes to the environment within which mobile code executes.

Deeds provides a fail-safe default [32] for every security event. Unless overridden, the default handler for an event disallows all requests associated with that event from downloaded programs. The default handler can only be overridden by explicit user request – either by a dialog box or by a profile file containing a list of user preferences.

4.2 Implementation

In this subsection, we describe the implementation of *Deeds*. We focus on implementation of program identity, events, event-histories, policies (including conventions for writing them), and policy management.

4.2.1 Program identity

We have implemented a new class-loader for downloading Java programs. A new instance of this class-loader is created for every downloaded program and is used to maintain information regarding its identity. This class-loader statically links a downloaded program by scanning its bytecode and extracting the set of classes that may be referred to during its execution. If the entire program is provided as a single `jar` file, this is straightforward. Else, the class-loader fetches and analyzes the non-local classes referred to and repeats this till transitive closure is achieved. If the scan of the bytecode indicates that the program explicitly loads classes, the linking operation is terminated and the program is not allowed to run.

After the linking operation is completed, the class-loader concatenates the code for all the non-system-library classes that are referenced by the program and uses the implementation of the SHA-1 algorithm provided in the `java.security` package to compute a secure hash. The result is used as the name of the program and is stored in the class-loader instance created for this program. This name can be used to maintain program-specific event-histories. It can also be stored in persistent storage and loaded as a part of the startup procedure.

4.2.2 Events and histories

Events: Two concerns guided our implementation of events: (1) the number of events is not fixed, and (2) the number of handlers associated with individual events could be large. We considered three alternatives. First, we could use a general-purpose mechanism (similar to Java Beans [8] and X [33]) to register events and handlers. The advantage of this approach is that it uses common code to manage all events and their associated handlers; the disadvantage is that all handlers must have the same type-signature which usually implies that the parameters need to be packed by the event manager and unpacked by each handler.

Second, we could dynamically modify the bytecode of the downloaded program to insert/delete calls to handlers at the points where the events are generated (as dynamic instrumentation programs [5, 15] do). To allow a user to modify an executing policy would require us to update the bytecode of running programs. We believe that the complexity of such an implementation is not commensurate with its advantages.

Finally, we could require that the handlers for each event be managed by a different event manager. This approach allows us to avoid the packing and unpacking of parameters as each event manager is aware of the parameters corresponding to its event. The disadvantage of this scheme is that a separate event manager has to be written for each event. However, event managers are highly stylized and can be automatically generated given a description of the event (see Figure 1 for an example).

We selected the third approach for implementing security events in *Deeds*. Combined with automatic generation of event managers, this allowed us to balance the needs of efficiency, implementation simplicity and ease of programming. Examples of *Deeds* security events include `checkRead()` and `checkConnect()`.[2]

History: Given the concern about the size of the log, we have chosen to avoid logging as a general technique for maintaining event-histories. Instead, we have left the decision about how to store histories to individual policies. All policies that we considered were able to summarize event-histories using simple data structures such as counters, booleans or lists. Note that this decision has the potential disadvantage that if policies do desire/need to log events, the lack of a common logging mechanism can result in the maintenance of duplicate logs. This can be fixed by using a selective logging mechanism that logs an event only if requested to do so by one or more handlers associated with the event.

4.2.3 Access-control policies

A *Deeds* access-control policy consists of the data-structures to maintain event-histories, handlers for each of the events that are mediated by the policy, and auxiliary variables and operations to facilitate management of multiple policies. Concretely, an access-control policy is implemented as a Java class that extends the `AccessPolicy` class shown in Figure 2. Handlers are implemented as methods of this class and the event-history is implemented as variables of this class. For example, a handler for the open-file-for-reading event could check if a socket has yet been created by the program. If so, it could raise a `GeneralSecurityException`; else, it could set a boolean to indicate that a file has been opened for reading and return.

When a security event occurs (e.g., when `checkRead` is called), control is transferred to the Deeds Security Manager which determines the class-loader for the program that caused the event using the `currentClassLoader()` method provided by the Java Security Manager. This method returns the class-loader corresponding to the most recent occurrence on the stack of a method from a class loaded using a class-loader. Since a new instance of the class-loader is created for every downloaded program and since this instance loads all non-system-library classes for the program, `currentClassLoader()` always returns the same class-loader every time it is called during the execution of a program. This technique safely determines the identity of the program that caused the security event.

Once the class-loader corresponding to the currently executing program has been determined, the Deeds Security Manager invokes the event manager corresponding to event

[2]Readers familiar with Java will recognize that all the `check*()` methods are security events.

```
public class checkReadManager implements EventManager {
  private static HandlerCheckRead hdlr = new HandlerCheckRead();

  public static void checkRead(FileDescriptor fd,DClassLoader cl)
  throws GeneralSecurityException {
    for (int i=0;i<hdlr.size;i++)
      hdlr.policy(i).checkRead(fd,cl);
  }

  public static void checkRead(String file,DClassLoader cl)
  throws GeneralSecurityException {
    for (int i=0;i<hdlr.size;i++)
      hdlr.policy(i).checkRead(file,cl);
  }

  public static void checkRead(String file,Object context,DClassLoader cl)
  throws GeneralSecurityException {
    for (int i=0;i<hdlr.size;i++)
      hdlr.policy(i).checkRead(file,context,cl);
  }
}
```

Figure 1: Example of an event manager class. Managers for other events would share the same structure but would replace `checkRead` by name the of the particular event. Some administrative details have been left out of the example; these details are common to all event managers.

being processed (e.g., `checkReadManager` in Figure 1). The event manager maintains the set of handlers associated with the event and invokes the handlers in the order they were attached to the event. If any of the handlers throws an exception, it is propagated to the caller; the remaining handlers are not invoked.

Deeds policies are expected to adhere to several constraints. These constraints are checked either at compile-time by the Java compiler or at runtime by the *Deeds* policy manager. These constraints are:

- Handler methods must include a `throws GeneralSecurityException` clause and must have the same name as the security event that they are intended for. The type signature for a handler method must be the same as the type signature of the security event it handles except for one additional argument – the class-loader. See Figure 1 for an illustration.
- A handler method must have the same number of variants as the security event that it is intended for. For example, a checkRead event has three variants – `checkRead(FileDescriptor fd)`, `checkRead(String file)`, and `checkRead(String file, Object context)`. Handlers for this event must have three variants. See Figure 1 for an illustration.
- Parameters of a policy must be explicitly identified. Each parameter must have a default value and a documentation string.
- The vector `targetEvents` specifies which events the handlers in this policy are to be attached to. The specification is in the form of a regular expression which is matched against fully-qualified names. For example, the target events for a checkRead handler could be specified as ``FileIO.checkRead'' or ``*.checkRead''. The former expression specifies only the checkRead event defined in the `FileIO` package whereas the latter specifies all checkRead events irrespective of the package they have been defined in. This specification is needed as Java's hierarchical namespace allows multiple methods with the same name to exist in different regions of the namespace. Since a security event is implemented by a method in a subclass of `EventManager` and since every package can have its own security events, the possibility of name clashes is real. For example, a library to perform file I/O, and a library to interact with a database could both wish to create a `checkRead` event. Since packages are independently developed, extensible systems, such as *Deeds*, cannot assume uniqueness of event names.
- Each policy must be accompanied by its source code and the name of the file containing the source code should be available as a member of the class implementing the policy. We believe that availability of source code of a policy is important to instill confidence in its operation and its documentation.

4.2.4 Policy manager

The *Deeds* policy manager makes extensive use of the Java reflection mechanism [17]. This mechanism allows Java code to inspect and browse the structure of other classes. The policy manager uses reflection to: (1) identify methods that are to be used as handlers (they are declared `public void` and throw the `GeneralSecurityException`); (2) identify parameters and their types; (3) initialize and update parameters; and (4) extract specification of the events that the handlers are to be attached to. In addition, it performs sev-

```
abstract synchronized public class AccessPolicy {
 public String name         ; // name of policy instance
 public Vector parameters   ; // policy parameters
 public Vector targetEvents ; //
 public String srcFileName  ; // source file location
 // these functions have to be provided by every policy
 public abstract String documentation();
 public abstract void   saveHistoryToDisk();
 public abstract void   restoreHistoryFromDisk();

 public Policy(String name) {
   ...
 }
}
```

Figure 2: Skeleton of the AccessPolicy class. The **synchronized** keyword ensures that at most one handler is updating the event-history at any given time.

eral administrative checks such as ensuring that all policy instances have unique names.

The policy manager is also responsible for ensuring that policies are persistent. It achieves this by storing the parameters for each policy instance on stable storage and using them to re-install the policy when the environment is reinitialized (on startup). It also periodically saves the event-history on stable storage.[3]

4.3 User interface

The *Deeds* user interface comes up only on user request and is used for infrequent operations such as browsing/loading/-installing policies. In this section, we describe the functionality of the user interface and present snapshots.

Browsing/viewing/loading policies: The *Deeds* user interface allows users to browse the set of available policies, to view documentation and source code for these policies and to create instances of individual policies. Note that every policy is required to have documentation (via the `documentation()` method) and access to its own source code (via the `srcFileName` member). In addition, every parameter has associated documentation which can be viewed. To load a parameterized policy, users need to specify values for all the parameters of the policy. Note that every parameter has a default value which is displayed. The *Deeds* policy manager uses the Java reflection mechanism to figure out the type of the parameters for display and parsing purposes. These functions of the user interface are illustrated in figures 3 and 4. In Figure 3, a policy is selected by clicking on its name and operations are selected using the buttons. Browsing and loading of individual policies is illustrated in Figure 4.

Installing/uninstalling policies: The *Deeds* user interface allows users to install loaded policies as well as to remove currently installed policies. For an illustration, see Figure 3. A loaded policy can be installed using the `Install Policy` button, and an installed policy can be removed using the `Uninstall Policy` button.

Checkpointing event-histories: *Deeds* allows user to checkpoint the current state of the event-histories for all policies using the `Save Settings` button (see Figure 3).

Browsing/setting default handlers: *Deeds* provides a fail-safe default for every security event. Unless overridden, the default handler for an event disallows all requests associated with that event from downloaded programs. The default handler can only be overridden by explicit user request – either by a dialog box or by a profile file containing a list of user preferences. The *Deeds* user interface allows users to browse and set default handlers for all security events.

4.4 Performance evaluation

There are two ways in which *Deeds* can impact the performance of downloaded code whose accesses it mediates. First, it can add an overhead to each request for protected resources. Second, it can increase the startup latency as it requires fetching, loading, linking and hashing of all non-system-library code before the program can start executing. In this section, we evaluate the performance impact of *Deeds*. All experiments were performed on a Sun E3000 with 266MHz UltraSparc processors and 512MB memory.

To determine the overhead of executing *Deeds* security-event handlers, we used a microbenchmark which repeatedly opened and closed files. A security event was triggered on the request to open a file. We varied the number of handlers from zero to ten. Each handler was identical (but distinct) and implemented the `editor` policy described earlier in this paper. It maintains two booleans, one that tracks if the program has ever tried to create a socket and the other that tracks if the program has ever tried to open a file. Each time it is invoked, it updates the file-opened boolean and checks the socket-opened boolean.

Table 1 presents the results. It shows that even with ten handlers, the overhead of *Deeds* security event handlers is less than 5%. Another point to note is that without any handlers, that is, when the infrastructure added to support security event handlers is not used, the overhead is less than 1%.

To evaluate the impact on startup latency, we compared

[3]Note that individual policies are free to save the event-history as frequently as they wish.

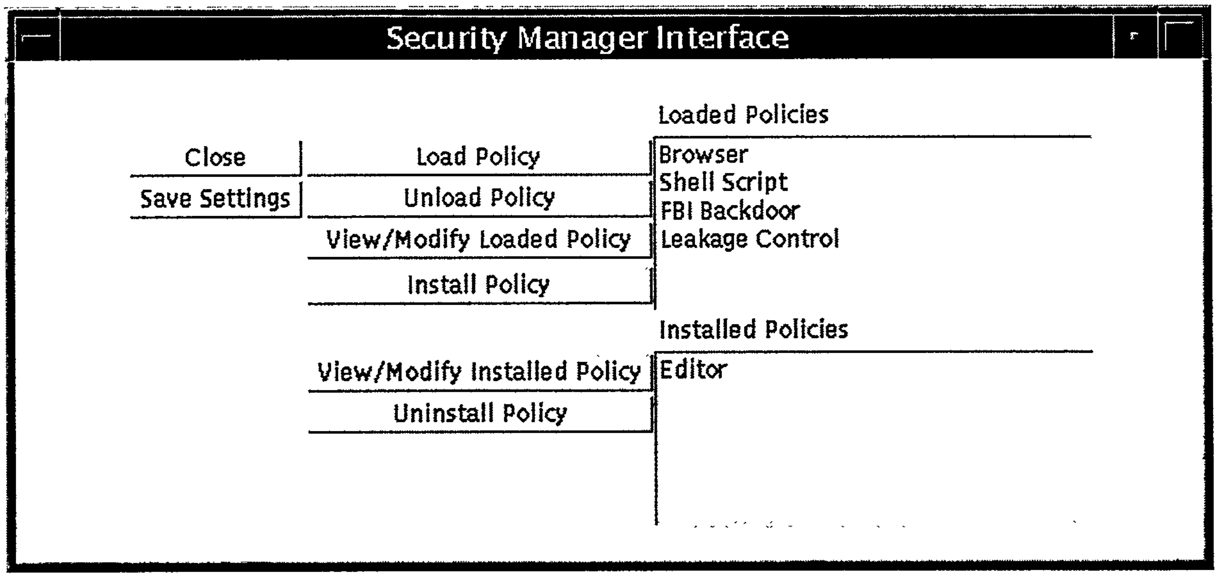
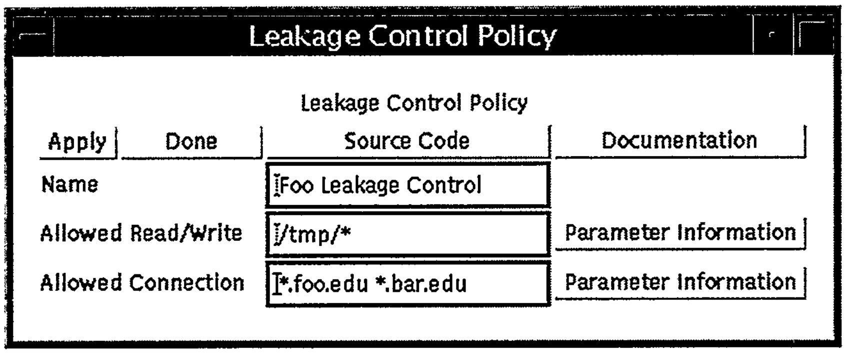


Figure 3: Graphical interface to the Deeds Security Manager



Name: Foo Leakage Control
Allowed Read/Write: /tmp/*
Allowed Connection: *.foo.edu *.bar.edu

Figure 4: Graphical interface for loading a policy

the time it takes to load, analyze and link complete Java applications using the *Deeds* class-loader to the time it takes to load just the first file using existing class-loaders. In both cases, all the files were local and were in the operating-system file-cache. For this experiment, we selected seven complete Java applications available on the web. The applications we used were: (1) `news-server`, the Spaniel News Server [36] which manages and serves newsgroups local to an organization; (2) `jlex`, the JLex [23] lexical analyzer; (3) `dbase`, the Jeevan [22] platform-independent, object-oriented database; (4) `jawavedit`, the JaWavedit audio file editor [21] with multi-lingual voice synthesis, signal processing, and a graphical user interface; (5) `obfuscator`, the Hashjava [14] obfuscator for Java class files; (6) `javacc`, the JavaCC [20] parser generator; and (7) `editor`, the WingDis editor [40].

Table 2 presents results for the latency experiments. As expected, the additional startup latency increases with the number of files as well as the total size of the program. Note this does not represent an increase in end-to-end execution time. Existing class-loaders already parse the bytecodes of class files as a part of the Java verification process; signed applets require computation of a similar hash function. Instead, the increase in startup latency is caused by moving the processing for all the class files before the execution begins. We expect that, once downloaded, programs of this size and these types (lexer/parser generators, editors, news server, database etc) will be reused several times. In that case, the program can be cached as a whole (instead of individual files) and the additional startup latency has to be incurred only once.

5 Discussion

History-based access-control for native binaries: History-based access-control is not specific to Java or to mobile code. It can be used for any system that allows interposition of code between untrusted programs and protected resources. Several operating systems (Solaris, Digital Unix, IRIX, Mach and Linux) allow users to interpose user-level code between an executing program and OS resources by intercepting system calls. This facility is usually used to implement debuggers and system call tracers. It has also been used to implement a general-purpose code interposition mechanism [24], a secure environment for helper applications used by browsers to display files with different formats [11] and a user-level file system [1]. It is also well-

Number of handlers	0	1	2	3	4	5	6	7	8	9	10
Percent overhead	0.7	1.8	2.6	2.4	2.9	3.5	4.2	3.9	4.3	4.1	4.3

Table 1: Overhead of *Deeds* security event handlers. The overhead was measured using a microbenchmark which repeatedly opened and closed files. Each handler was identical (but distinct) and implemented the `editor` policy.

Application	`newsserver`	`jlex`	`dbase`	`jawavedit`	`obfuscator`	`javacc`	`editor`
Number of classes	24	40	104	125	144	81	212
Total code size (KB)	120	289	514	508	483	578	979
First class size (KB)	5	1	11	2.5	4	7	1.5
Loading classes	0.2s	0.3s	0.5s	0.9s	1.0s	0.8s	1.3s
Parsing bytecodes	0.1s	0.1s	0.2s	0.5s	0.5s	0.4s	1.2s
Hashing bytecodes	0.1s	0.5s	0.7s	1.1s	1.4s	2.6s	2.9s
Additional latency	0.4s	0.9s	1.4s	2.5s	2.9s	3.8s	5.6s

Table 2: Breakdown of additional startup latency incurred by *Deeds*.

suited for implementing a *Deeds*-like history-based access-control mechanism to mediate access to OS resources from native binaries.

Pre-classified program behaviors: The *one-out-of-k* policy described in section 2 classifies program behaviors in an on-line manner. A program gets classified as a browser, an editor, or a shell depending on whether it has connected to a remote site, has opened local files for modification, or has created a sub-process. To be able to do this for a wide variety of program behaviors, the policy that does the classification and subsequent management of privileges has to contain code to handle all these behaviors. An alternative scheme would be to allow program-providers to label their programs with pre-classified behavior patterns and to allow the users to specify which behaviors they would like to permit. The policies governing individual behaviors could be added/deleted as need be. While this scheme would require agreement on the labeling scheme, it is no more complex than the MIME-types-based scheme that is already in use for displaying/processing different data formats. This scheme is similar to *program-ACLs* and related defenses proposed for trojan-horse attacks [25, 39].

Joint-authorization: Commercial applications, such as contracts and purchase orders, may require multiple authorizations since the organization may wish to reduce the risk of malfesance by dispersing trust over several individuals. History-based access-control policies can be used to implement joint-authorization [37] or *k-out-of-n-authorizations* [3]. For example, a policy may require that three out of five known individuals must make the same request within the last T units of time for the request to be granted; else the request is denied. In this case, the history consists of the requests that have been made in the last T units of time.

6 Related work

The primary problem for access-control mechanisms for mobile code is to be able to differentiate between different programs executing on behalf of the same user and to provide them with different privileges based on their expected behavior and/or potential to cause damage. A similar problem occurs in the context of trojan-horse programs and viruses. To deal with such programs, several researchers have developed mechanisms to limit the privileges of individual programs based on their expected behavior [10, 25, 26, 27, 28, 39]. Karger [25] uses information about file extensions and behavior of individual programs to determine the set of files that a program is allowed to access (eg. a compiler invoked on `x.c` is only allowed to create `x.{o,u,out}`). Lai [28] replaces the inference mechanism by an explicit list of files accessible by a program. Wichers et al [39] associate *program-ACLs* with each file thereby limiting the set of programs that can access each file. King [26] uses a regular-expression-based language to specify the set of objects each operation can access. Ko et al[27] use a language based on predicate logic and regular expressions to specify the security-relevant behavior of privileged programs and propose to use this specification for intrusion detection. All these approaches assume that the set of programs to be run are fixed and their behaviors are known. The mobile code environment is different as the set of programs that will execute is inherently unknown. History-based access-control is able to classify programs in an on-line manner and to thereafter execute them within an environment with appropriate privileges. For example, the *one-out-of-k* policy dynamically classifies downloaded programs into one of three classes: browsers, editors and shells.

The use of secure hash functions to derive a content-based name for software has been proposed by Hollingsworth et al [16]. They propose to use these names for configuration and version management of large applications and application suites (such Microsoft Office).

An important feature of *Deeds* is its capability to install and compose multiple user-specified policies. Several researchers have proposed languages to allow users to specify access-control policies and frameworks to compose these policies [3, 13, 18, 19]. Three of them [13, 18, 19], propose logic-based declarative languages and use inference mechanisms of various sorts to compose policies. Blaze et al [3] propose a language that contains both assertions and procedural filters and use a mechanism similar of that used in *Deeds* to implement composition. Access-control policies for *Deeds* are entirely procedural. Furthermore, they can be updated while the programs whose accesses are being controlled are still executing.

Two research groups have recently proposed constraint languages for specifying security policies temporal aspects. Simon&Zurko [35] propose a language for specifying temporal constraints such as `HasDone`, `NeverDid`, `NeverUsed` and `SomeoneFromEach` for separation of duty in role-based environments. These predicates correspond to summaries of event-histories in *Deeds* terminology.

Mehta&Sollins [29] have independently proposed a constraint language for specifying simple history-based access-control policies for Java applets. This work was done in parallel with ours [7]. The approach presented in their paper has two major limitations. First, they use the domain name of the server that provides the applet as its identifier. This assigns the same identifier to all applets from the same host. In addition, it is vulnerable to DNS spoofing attacks. They suggest that this problem can be fixed by using the identity of the author/supplier of an applet as its name. This assigns the same identifier to all applets from a single author/supplier and results in a single merged history. It is not clear how such a merged history would be used as the predicates and variables in their language are applet-specific. Even if each supplier provides only one applet, this is a viable solution only if all the classes referenced by the applet are provided in a single `jar` file or are required to be signed by the same principal. Otherwise, it is possible for a malicious server that is able to spoof IP addresses to intercept intermediate requests for dynamically linked classes and provide malicious substitutes. The second limitation of their approach is that it provides a small and fixed number of events. This limits the variety and power of the policies that can be developed.

The event model used in *Deeds* is similar to that used in the SPIN extensible operating system [2]. An interesting feature of SPIN is the use of dynamic compilation to improve the performance of event dispatching [5]. If the performance of event dispatching becomes a problem for *Deeds* (eg. if individual events have a large number of handlers) we can use a similar technique.

7 Current status and future directions

Deeds is currently operational and can be used for stand-alone Java programs. We are in the process of identifying a variety of useful patterns of behaviors and evaluating the performance and usability of *Deeds* in the context of these behaviors.

In the near term, we plan to develop a history-based mechanism for mediating access to OS resources from native binaries. We also plan to explore the possibility of using program labels to indicate pre-classified behaviors and automatic loading/unloading of access-control policies to support this.

In the longer term, we plan to explore just-in-time binary rewriting to insert event generation and dispatching code into downloaded programs. This would allow users to create new kinds of events as and when they desire. Currently, new kinds of events are created only by system libraries.

Acknowledgments

We would like to thank anonymous referees for their insightful comments which helped improve the presentation of this paper.

References

[1] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. Extending the operating system at the user level: the Ufo global file system. In *Proceedings of the 1997 USENIX Annual Technical Conference*, 1997.

[2] B. Bershad, S. Savage, P. Pardyak, E. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the spin operating system. In *Proc of the 15th ACM Symposium on Operating System Principles*, pages 267–84, 1995.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc of the 17th Symposium on Security and Privacy*, pages 164–73, 1996.

[4] D. Brewer and M. Nash. The Chinese Wall Security Policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, 1989.

[5] C. Chambers, S. Eggers, J. Auslander, M. Philipose, M. Mock, and P. Pardyak. Automatic dynamic compilation support for event dispatching in extensible systems. In *Workshop on Compiler Support for Systems Software*, 1996.

[6] B. Christiansen, P. Cappello, M. Ionescu, M. Neary, K. Schauser, and D. Wu. Javelin: Internet-based parallel computing using Java. In *Proceedings of the 1997 ACM Workshop on Java for Science and Engineering Computation*, 1997.

[7] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. Technical report, University of California, Santa Barbara, 1997.

[8] R. Englander. *Developing Java Beans.* O'Reilly & Associates, 1997.

[9] J. Fritzinger and M. Mueller. Java security. Technical report, Sun Microsystems, Inc, 1996.

[10] T. Gamble. Implementing execution controls in Unix. In *Proceedings of the 7th System Administration Conference*, pages 237–42, 1993.

[11] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[12] L. Gong. New security architectural directions for Java. In *Proceedings of IEEE COMPCON'97*, 1997.

[13] C. Gunter and T. Jim. Design of an application-level security infrastructure. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[14] The HashJava code obfuscator. Available from 4thPass Software,810 32nd Avenue South, Seattle, WA 98144[4].

[15] J. Hollingsworth, B. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *SHPCC*, 1994.

[16] J. Hollingsworth and E. Miller. Using content-derived names for caching and software distribution. In *Proceedings of the 1997 ACm Symposium on Software Reusability*, 1997.

[4] *http://www.sbktech.org/hashjava.html*

[17] C. Horstmann and G. Cornell. *Core Java 1.1*, volume I - Fundamentals. Sun Microsystems Press, third edition, 1997.

[18] T. Jaeger, A. Prakash, and A. Rubin. Building systems that flexibly control downloaded executable context. In *Proc of the 6th Usenix Security Symposium*, 1996.

[19] S. Jajodia, P. Samarati, V. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 474–85, 1997.

[20] The JavaCC parser generator. Available from Sun Microsystems Inc. 901 San Antonio Road, Palo Alto, CA 94303 USA[5].

[21] The JaWavedit Audio File Editor. Available from Florian Bomers' web site[6].

[22] The Jeevan object-oriented database. Available from W3apps Inc., Ft. Lauderdale, Florida[7].

[23] The JLex lexical analyzer generator. Available from the Department of Computer Science, Princeton University[8].

[24] M. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, 1993.

[25] P. Karger. Limiting the damage potential of the discretionary trojan horse. In *Proceedings of the 1987 IEEE Syposium on Research in Security and Privacy*, 1987.

[26] M. King. Identifying and controlling undesirable program behaviors. In *Proceedings of the 14th National Computer Security Conference*, 1992.

[27] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings. 10th Annual Computer Security Applications Conference*, pages 134–44, 1994.

[28] N. Lai and T. Gray. Strengthening discretionary access controls to inhibit trojan horses and computer viruses. In *Proceedings of the 1988 USENIX Summer Symposium*, 1988.

[29] N. Mehta and K. Sollins. Extending and expanding the security features of Java. In *Proceedings of the 1998 USENIX Security Symposium*, 1998.

[30] Microsoft Corporation. *Proposal for Authenticating Code Via the Internet*, Apr 1996. *http://www.microsoft.com/intdev/security/authcode.*

[31] R. Rivest. The MD5 message-digest algorithm. RFC 1321, Network Working Group, 1992.

[32] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep 1975.

[33] R. Scheifler and J. Gettys. *X Window System : The Complete Reference to Xlib, X Protocol, Icccm, Xlfd.* Butterworth-Heinemann, 1992.

[34] Secure hash standard. Federal Information Processing Standards Publication, FIPS, PUB 180-1, April 1995.

[35] R. Simon and M. Zurko. Separation of duty in role-based environments. In *Proceedings of the IEEE Computer Security Foundations Workshop'97*, 1997.

[36] The Spaniel News Server. Available from Spaniel Software[9].

[37] V. Varadharajan and P. Allen. Joint actions based authorization schemes. *Operating Systems Review*, 30(3):32–45, 1996.

[38] D. Wallach, D. Balfanz, D. Dean, and E. Felten. Extensible security architecture for Java. In *SOSP 16*, 1997.

[39] D. Wichers, D. Cook, R. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo. PACL's: an access control list approach to anti-viral security. In *USENIX Workshop Proceedings. UNIX SECURITY II*, pages 71–82, 1990.

[40] The WingDis Editor. Available from WingSoft Corporation, P.O.Box 7554, Fremont, CA 94537[10].

[5] *http://www.suntest.com/JavaCC*
[6] *http://rummelplatz.uni-mannheim.de/ boemers/JaWavedit*
[7] *http://www.w3apps.com*
[8] *http://www.cs.princeton.edu/ appel/modern/java/JLex*

[9] *http://www.searchspaniel.com/newsserver.html*
[10] *http://www.wingsoft.com/javaeditor.shtml*