IRM Enforcement of Java Stack Inspection*

Úlfar Erlingsson
deCODE Genetics
Lynghǻls 1, 110
Reykjavík, Iceland
ulfar@decode.is

Fred B. Schneider
Department of Computer Science
Cornell University
Ithaca, New York 14853
fbs@cs.cornell.edu

Abstract

Two implementations are given for Java's stack-inspection access-control policy. Each implementation is obtained by generating an inlined reference monitor (IRM) for a different formulation of the policy. Performance of the implementations is evaluated, and one is found to be competitive with Java's less-flexible, JVM-resident implementation. The exercise illustrates the power of the IRM approach for enforcing security policies.

1. Introduction

Java was designed to support construction of applications that import and execute untrusted code from across a network. The language and run-time system enforce security guarantees for downloading a Java applet from one host and executing it safely on another. In Sun's Java implementation [12, 14, 11], some of these security guarantees involve run-time checks by the JVM (Java Virtual Machine), others involve load-time checks on the JVML (Java Virtual Machine Language) bytecode files defining JVM *classes*—the unit of JVM binary code and of Java object hierarchies—and still others follow from the syntax of JVML and the Java programming language.

The JVM run-time checks enforce access-control policies that associate access rights with the class that initiates the access. The *sandbox policy* of early (pre Java 2) JVM implementations distinguishes between code residing locally and code obtained from across the network. The more recent Java 2 *stack inspection policy* refines this. In Java 2, whether an access is permitted can depend on the current nesting of method invocations. Enforcement of the stack inspection access-control policy therefore relies on information found on the JVM run-time call stack.

Changing which access-control policy is supported by the JVM requires changing the JVM. Thus, programs expecting Java 2's stack inspection policy to be enforced cannot execute on earlier-generation JVM implementations. On a JVM that enforces the stack inspection policy, applications requiring other access-control policies might be ruled out altogether, might require awkward constructions[1], or might be forced to employ their own application-level custom enforcement mechanisms. Finally, such a JVM includes mechanisms that may or may not be needed for executing any given Java application. For embedded applications, where memory is at a premium, the size of the JVM footprint is crucial; there is considerable incentive to omit unused enforcement mechanisms.

This paper describes an alternative to putting access-control enforcement in a run-time environment, such as the JVM. We show how an *in-lined reference monitor* (IRM) can be merged into Java applications to enforce security policies like stack inspection. With the IRM approach, a trusted rewriter instruments applications with checks that cannot be circumvented and that cause execution to be monitored for violations of a specified security policy.[2] Two IRM implementations of stack inspection are reported—one is a reformulation of security passing style proposed in [19, 20]; the other is new and exhibits performance that is competitive with existing commercial JVM-resident implementations.

*Supported in part by ARPA/RADC grant F30602-96-1-0317, AFOSR grant F49620-94-1-0198, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-99-1-0533, National Science Foundation Grant 9703470, and a grant from Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

[1] For example, certain access-control policies can be implemented with stack inspection only by creating multiple copies of the same class in different code bases or by creating multiple instances of identical class loaders.

[2] The IRM approach is capable of enforcing EM policies [16], an extremely rich class that includes mandatory and discretionary access control, Chinese Wall, type enforcement, and the Clark-Wilson commercial policy but that excludes certain information flow policies.

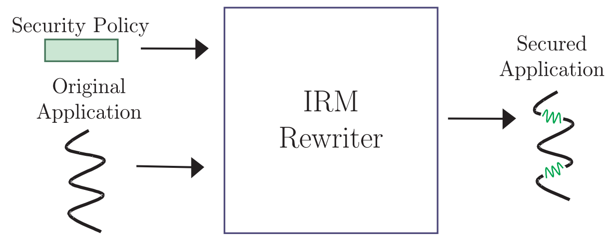

Figure 1. IRM approach to security policy enforcement.

Java 2's stack inspection policy is a particularly challenging one to enforce with an IRM because state relevant to policy enforcement (the JVM run-time call stack) is not directly accessible to Java applications. That we are able to obtain a new implementation exhibiting competitive performance reflects well on the practicality of the IRM approach. And having an IRM implementation for stack inspection means that Java 2 programs can be run on earlier generation JVM implementations, that variants of stack inspection as well as entirely new security policies can be enforced on Java programs without changing the JVM, and that unused enforcement mechanisms need not bloat Java applications or the JVM.

We proceed as follows. Section 2 briefly summarizes our PoET/PSLang toolkit for synthesizing IRMs; PoET/PSLang is a successor to our SASI tool [7]. Section 3 reviews Java 2's stack inspection policy and the primitives that implement this policy. An IRM version of the security-passing style [19, 20] implementation of stack inspection is described in Section 4; an IRM implementation for a new way to support Java 2's stack inspection policy is given in Section 5. Finally, Section 6 concludes with some remarks about the IRM approach and about limitations we discovered in Java's stack inspection policy.

2. Inlined Reference Monitors

For a *reference monitor* [2] to enforce a security policy, (i) it must mediate all events relevant to the policy being enforced, (ii) its integrity must be protected from subversion by applications, and (iii) its presence must be transparent to applications [15]. Address-space isolation has traditionally been employed for ensuring the integrity of reference monitors, but other approaches are also feasible.

With an in-lined reference monitor, a load-time, trusted *rewriter* merges checking code into the application itself and uses program analysis and program rewriting to protect the integrity of those checks. The application is thus transformed by the rewriter into a *secured application*, which is guaranteed not to take steps violating the security policy being enforced. See Figure 1.

Specifying an IRM involves defining

- *security events*, the policy-relevant operations that must be mediated by the reference monitor;
- *security state*, information stored about earlier security events that is used to determine which security events can be allowed to proceed; and
- *security updates*, program fragments that are executed in response to security events and that update the security state, signal security violations, and/or take other remedial action (e.g. block execution).

Policy Enforcement Toolkit (PoET) [6] implements IRMs for JVML applications. A primary concern in the design of PoET was the trusted computing base. PoET comprises approximately 17,500 lines of Java source code and thus increases the size of the trusted computing base by that amount. Although the PoET rewriter does local optimizations on inserted code—to delete (some) superfluous enforcement checks—it does not attempt global program analysis because we feared further increases to the size and complexity of the trusted computing base. In addition, PoET works at the level of JVML (and not the Java programming language).[3] Transforming Java programs instead of JVML programs would make a Java compiler part of the trusted computing base, an unwise choice given the size and complexity of Java compilers. Moreover, by choosing to transform JVML programs, we do not require that source code for an application be available at a site for a security policy to be imposed by the site on that application.

Note that the PoET rewriter need not be run on the same computer as the JVM or even as the Java compiler. Thus, PoET contributes to the size of the trusted computing base without increasing the size of the run-time environment used to execute Java applications. In most cases, PoET will run on the same computer as the JVM, yet it is not difficult to imagine mobile-code and other networked settings

[3] Currently, PoET does not process Java native methods—code written in native machine language—and this restricts what policies can be enforced by excluding some security events. However, this is not a limitation of the IRM approach in general, as demonstrated by x86 SASI [7], which implements IRMs on x86 machine-language applications.

```
IMPORT LIBRARY Lock;
ADD SECURITY STATE \{
    int openWindows = 0;
    Object lock = Lock.create();
\}
ON EVENT begin method
WHEN Event.fullMethodNameIs("void java.awt.Window.show()")
PERFORM SECURITY UPDATE \{
    Lock.acquire(lock);
    if( openWindows = 10 ) \{
      HALT[ "Too many open GUI windows" ];
    \}
    openWindows = openWindows + 1;
    Lock.release(lock);
\}
ON EVENT begin method
WHEN Event.fullMethodNameIs("void java.awt.Window.dispose()")
PERFORM SECURITY UPDATE \{
    Lock.acquire(lock);
    openWindows = openWindows - 1;
    Lock.release(lock);
\}
```

Figure 2. PSLang security policy that allows at most 10 open windows.

where security policies are added to an application before that application is distributed to other sites for execution.

The integrity of a PoET-inserted IRM's security state and security updates is protected by JVML type-safety guarantees, since the JVML type system prohibits access to code and data not in the classes originally comprising an application. JVML type-safety also means that JVML applications are unaffected by the presence of checking code that PoET adds to create an IRM. This is because JVML type-safety prevents code from being viewed as data, so code inserted by the PoET rewriter cannot be directly detected by the application that was modified. In addition, JVML type-safety prevents a Java application from mentioning names and types not in that application's original namespace; the PoET rewriter chooses the names and types for any checking code it adds accordingly.[4]

Security policies for PoET are specified using Policy Specification Language (PSLang), an event-oriented, imperative language with Java-inspired syntax. PSLang is a small subset of Java so that the PSLang compiler could be small. In PSLang security policies, any JVM event that could occur during execution of the original application—from method calls to arithmetic operations—can be identified as a security event and, therefore, will trigger execution of an associated security update. PSLang is expressive enough to specify the EM policies of [16].[5]

To illustrate the syntax of PSLang, Figure 2 gives a policy to prevent Java applications from opening more than 10 Java windows. The security state is defined in the `ADD SECURITY STATE` block at the start of the specification. It consists of an integer variable (`openWindows`) and a mutual exclusion lock (`lock`). Variable `openWindows` counts the number of open windows; `lock` is used to protect `openWindows` from concurrent access. Security updates are introduced by `PERFORM SECURITY UPDATE` (two are in Figure 2), and security events are identified by `ON EVENT ... WHEN` tags. The two security events in the policy of Figure 2 specify that the IRM executes security updates prior to method invocations for opening and closing Java windows. Whenever the application attempts to open a window, the JVM executing the application is terminated (because `HALT` is invoked) if 10 windows have already been opened (i.e., `openWindows` $= 10$); otherwise, `openWindows` is incremented. And whenever a window is closed, `openWindows` is decremented.

[4]The presence of an IRM for certain policies cannot be completely hidden. Reflection and the measurement of execution timing can allow a Java application to detect added code (but does not compromise security enforcement).

[5]To be precise, any security policy that can be specified using a security automaton involving transition predicates that are JVM events can be formulated in PSLang.

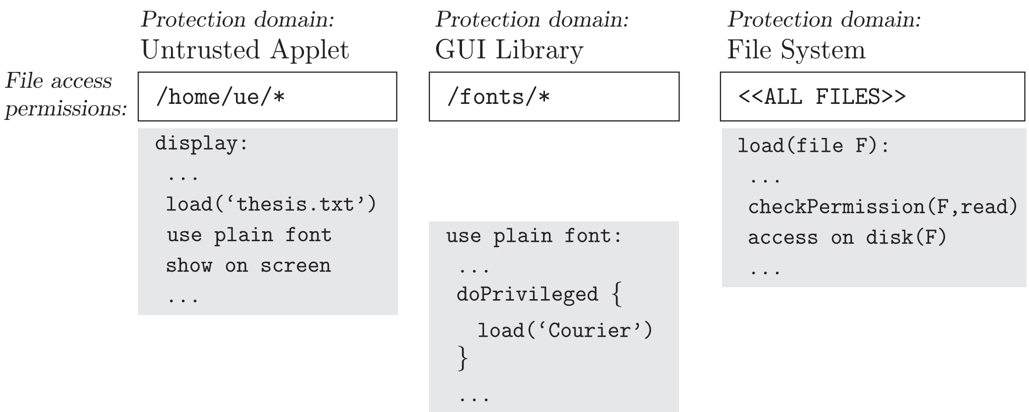

Figure 3. Three Protection Domains.

3. Review of Java 2's Stack Inspection Policy

Java 2's stack inspection access-control policy is based on *policy files* which associate *permissions* with *protection domains*. The policy file read when the JVM starts is what defines the access-control policy for applications then executed by that JVM, as follows.

Protection domains. Each application initially is a sequence of bytes stored outside the JVM. The bytes are fetched by a class loader and then executed by the JVM. Prior to execution, the bytes are assigned to a protection domain in accordance with the source of the bytes (a network address or a file name) and any attached cryptographic signature.[6]

Permissions. Each protection domain implies a set of permissions. This set includes all those permissions associated with the protection domain by the policy file, as well as other implied permissions. The definition of a permission—a class—states what permissions it implies by implementing an `implies` method.

As an example, Figure 3 depicts three protection domains: Untrusted Applet, GUI Library, and File System. Permissions associated with each domain appear in the box below the name of the protection domain; psuedo-code associated with that domain appears below the permissions. Notice that file access permissions are given in the figure using patterns rather than complete file names—the `implies` method would decode those patterns to generate permissions for actual files in the expected way.

[6] The Java class loader used to fetch those bytes can also be involved in determining the protection domain of those bytes [13]. Since new class loaders can be created at runtime, protection domains can be created dynamically, thereby helping to overcome the static nature of policy files.

For a permission P, invoking the `checkPermission`(P) method of Java 2 throws a security exception if access should not be allowed to proceed; it otherwise has no visible effect. Whether a security exception is thrown depends on the protection domains assigned to the methods from which control has not yet returned—methods having frames on the JVM call stack when `checkPermission`(P) is invoked. Specifically, when `checkPermission`(P) is invoked, the JVM call stack is traversed from top to bottom (i.e., starting with the frame for the method containing the `checkPermission`(P) invocation) until either the entire stack is traversed or an invocation is found within the scope of a `doPrivileged` block. In that traversal, the stack frames encountered are checked to make sure their associated protection domains imply permission P; if some frame doesn't, a security exception is thrown.

Observe that `doPrivileged` supports a form of rights amplification. Without `doPrivileged` or some equivalent, it would be impossible to invoke methods that require permissions not already held by the invoker. Such rights amplification is crucial, for example, when untrusted code invokes a system routine. A system routine is trusted to perform adequate checks before exercising the power that comes with the more powerful permissions in its associated protection domain; it should also be trusted to invoke only methods that are similarly prudent. So, a construct like `doPrivileged` that allows an invoked method to exercise permissions beyond those of its invoker is both sensible and useful.

The psuedo-code in Figure 3 illustrates how `doPrivileged` is used. `display` directly invokes the `load` method of File System and invokes the `use plain font` method of GUI Library. Also note that `use plain font` invokes `load`—loading a font may require

Method call/return: $A \rightarrow B$ At start of B, look up protection domain P_B for B's code and push P_B on the thread-local `domainStack`. At return from B (either normally or by a thrown exception), pop `domainStack`, removing P_B.	**doPrivileged** $\{S\}$ Push a distinguished token **doPriv** on `domainStack`, at the beginning of the `doPrivileged`, and pop the token off at the end (whether an exception was thrown or not).
checkPermission(P) Scan `domainStack` from top to bottom (without modifying it), and look at each protection domain p. Throw a security exception if p does not imply P, but accept if $p =$ **doPriv** or the bottom of `domainStack` is reached.	**Create thread:** T Set the `domainStack` of T to contain a copy of the contents of the `domainStack` of its parent thread.

Table 1. IRM_{SPS} implements security-passing style.

loading a file that contains bit maps for the font. We then have:

- In invoking `load('thesis.txt')`, the `checkPermission` will throw a security exception if protection domains File System (the frame at the top of the stack) and Untrusted Applet (the next and bottom frame on the stack) do not each imply the needed permissions for reading that file. They do if `thesis.txt` resides in `/home/ue`.

- In invoking `load('Courier')` while executing in `use plain font`, the `checkPermission` will throw a security exception if protection domains File System (the frame at the top of the stack) and GUI Library (the next frame on the stack) do not each imply the needed permissions for reading that file. They do if `Courier` resides in `/fonts`. Untrusted Applet is not checked for permissions, because the invocation of `load` in GUI Library is within the scope of a `doPrivileged`.

Java's stack inspection policy also handles dynamic creation of threads. When a new thread T is created, T is given a copy of the existing run-time call stack to extend. The success of subsequently evaluating `checkPermission` in thread T thus involves permissions associated with the call stack (or some other representation of the permissions implied by the call stack) when T is created.

4. A Security-Passing Style IRM

The first work on modifying JVML programs to enforce stack inspection is described in [19, 20]. There, an additional variable is introduced to replicate information from the JVM run-time call stack. This variable is changed upon invoking or returning from a method call as well as upon entering or exiting the scope of a `doPrivileged` block; the variable is scanned when `checkPermission` is evaluated. The resulting scheme is called *security-passing style* (SPS) because the new variable is passed to method invocations as an additional argument.

SPS is an example of the IRM approach, so it will be no surprise that we were able to use PoET and build IRM_{SPS}, an implementation of SPS. The security updates that IRM_{SPS} associates with each security event—method call and return, `checkPermission`, `doPrivileged`, and thread creation—are sketched in Table 1; the actual PSLang formulation requires less than three pages and appears as Appendix A of [8].

In the PSLang that specifies IRM_{SPS}, variable `domainStack` replicates policy-relevant information from the JVM run-time call stack; this variable is local to each thread (and is equivalent to the additional explicit argument to method invocations employed in [19, 20]). It is worth noting exactly how IRM_{SPS} handles security updates associated with a method call from A to B. Permissions for B could be added to security state `domainStack` either inside method A or inside method B. But performing the update inside method A turns out to be less desirable in part because when B is a virtual method (the Java equivalent of a function pointer), a dynamic lookup would be required to determine its permissions. Therefore, IRM_{SPS} does the security update inside method B.

Performance Overhead

In order to understand the performance of stack inspection implementations, we must know the frequency and cost of relevant security events in actual applications. We therefore measured four applications: the Jigsaw 2.01 web server [3], Sun's `javac` Java 1.1 compiler [12], the `tar` utility [5], and an MPEG video player [1]. All were run

	Method calls	doPrivileged	checkPermission count	checkPermission avg checked	New threads
Jigsaw	2,476,731	1,002	5,333	18.7	71
`javac`	1,456,970	0	1,067	12.4	0
`tar`	19,580	0	6,509	8.6	0
MPEG	35.997.662	101	205	5.7	201

(a) Frequency of stack inspection primitives.

Method call	doPrivileged	checkPermission	New thread
1.00μs	1.66μs	7.7μs	6.5μs

(b) Benchmarked cost of IRM_{SPS} primitives (at stack depth 10).

	JVM	IRM_{SPS}
Jigsaw	6.2%	20.1%
`javac`	2.9%	46.2%
`tar`	10.1%	3.0%
MPEG	0.9%	72.5%

(c) Overhead of JVM-resident and IRM_{SPS} implementations.

Table 2. Assessing stack inspection performance.

using modern JVMs[7] with garbage collection disabled on a 300Mhz Pentium II running Windows 98. Since quantifying access-control overhead was of interest, the first three benchmark applications used the same set of 500 small synthetic Java source files as their input.

Table 2(a) shows how many times the various stack inspection primitives were invoked in the benchmarked applications. The cost of `doPrivileged`, `checkPermission`, and thread creation can be relative to the size of the JVM call stack, and—because `checkPermission` is dominant—we also report the average number of accessed stack frames ("avg checked") for that operation. So that the numbers are less dependent on irrelevant implementation details, stack inspection primitives used in the construction of permission objects have not been counted. For instance, not counted are the `doPrivileged` invocations for creating each `java.io.FilePermission` object in Sun's implementation.

Table 2(b) shows the overhead, in microseconds, for the IRM_{SPS} stack inspection primitives. The values shown are averages from a synthetic benchmark of the primitives. The primitives in the last three columns were benchmarked using a stack depth of 10—each operation accessed 10 stack frames.

Table 2(c) compares the run-time overhead of Sun's JVM-resident implementation of stack inspection and IRM_{SPS}. The column labeled JVM gives the percentage overhead between running the application on Java 2's JVM with stack inspection enabled versus without stack inspection enabled; the column labeled IRM_{SPS} gives the percentage overhead between running the application with IRM_{SPS} on Java 1.1[8] versus without any IRM.

The measurements in Table 2 do not include the cost of constructing permission objects or of executing their `implies` methods. This better quantifies the relative differences in overhead between stack inspection implementations. The numbers shown are based on the average execution time for 15 runs of the synthetic benchmarks and the applications. Percentages in Table 2(c) relate two of these averages. For each average we computed, the standard deviation was found to be small enough to be ignored in interpreting the numbers.

The JVM-resident implementation is considerably cheaper for Jigsaw, `javac`, and MPEG. This is not surprising because of the per method call cost of IRM_{SPS} and the large number of method calls each of these applications makes. However, when an application has many permission checks relative to the number of method calls, IRM_{SPS} may exhibit less overhead than the JVM-resident implementation. This is because IRM_{SPS} can amortize

[7]For JDK 1.1.7, we used Symantec Java! JustInTime Compiler Version 3.10.107(i); for JDK 1.2, we used Sun's distribution that employs Symantec Java! JustInTime Compiler Version 3.00.078(x).

[8]We employed Java 1.1's JVM to measure the overhead of IRM_{SPS} because the stack inspection implementation already present in Java 2's JVM would otherwise distort the measurements.

Method call/return: $A \rightarrow B$
Nothing.

doPrivileged $\{S\}$
At the beginning of the `doPrivileged` push the current JVM call stack frame number onto `privStack`; at the end pop it off (whether an exception was thrown or not).

checkPermission(P)
Let `bottom` be the privileged stack frame number on top of `privStack`, or 0 if there is none. Scan the current JVM call stack from top to `bottom` and find the protection domain p for each stack frame—reject if ever p does not imply P. If there was no privileged stack frame, likewise scan the `ancestralStack`.

Create thread: T
Let the `ancestralStack` of T be either a copy of the `ancestralStack` of its parent thread, with the current JVM call stack pushed on top, or—if there's a privileged stack frame number on `privStack`—the top portion of the current JVM call stack up to that privileged frame.

Table 3. IRM_{Lazy} uses the JVM call stack.

the cost of creating `domainStack` over a large number of `checkPermission`'s and each `checkPermission` is likely to be as cheap, or cheaper, under IRM_{SPS}. The results for `tar` illustrate this benefit.

An Improved SPS Implementation Scheme

The overhead of an SPS stack inspection implementation would be improved if the security state (i.e., `domainStack`) were not updated on each method call. In fact, updates need to be made only when a method call crosses protection domains—method calls within the same protection domain repeatedly push the same permission onto `domainStack`, and `checkPermission` is unaffected by replacing sequences of identical stack frames with a single frame.

The implementation of [19, 20] exploits this insight. The implementation comprises 12,800 lines of Java code, of which 1700 lines implement an analysis to determine whether invoked methods are in the same or different protection domains as the invoker and 6900 lines are produced by JOIE, the generic JVML rewriter [4]. With these optimizations, [19, 20] reports overall security enforcement overheads of between 13% and 17% of total execution time—still relatively high when compared to the overheads on the same applications run under the JVM-resident implementation stack inspection. Adding this optimization to IRM_{SPS} did not seem worthwhile, given the performance gains we achieve in other ways with the IRM implementation of the next section.

5. A New IRM Stack Inspection Implementation

Sun's implementation of stack inspection profits from having direct access to the JVM call stack, because no overhead is then incurred at method calls in order to keep track of nested invocations for subsequent `checkPermission` evaluation. Since method calls are the common case, the performance advantages of this design should be obvious.

In order to specify such a scheme in PSLang, some facility is needed for accessing the JVM run-time call stack. Fortunately, such can be found in Java. First, Java provides an interface so that exceptions can print a textual description of the JVM call stack when they are thrown; second, the Java `SecurityManager` contains a protected method `getClassContext` that returns a copy of the JVM call stack as an array of `Class` objects, each a unique identifier for the code at that JVM call stack frame. The PoET runtime makes this latter interface accessible to PSLang specifications (as part of PoET's `System` library) by extending the `SecurityManager`.

Table 3 sketches security events and updates for IRM_{Lazy}, an IRM stack inspection implementation that uses the JVM call stack. (The actual five page PSLang formulation appears as Appendix B of [8].) Notice how work has been moved from method call/return to the implementation of `doPrivileged`, `checkPermission`, and new thread creation (which all must make a copy of the call stack when they are invoked). `doPrivileged` pushes the frame number for the stack frame at the top of the current JVM call stack onto a separate thread-local variable, `privStack`. This frame number then serves to bound the segment of the JVM call stack that must be traversed in evaluating `checkPermission`—stack frames appearing lower on that call stack are not checked. For each thread, the relevant stack frames of parent threads are stored in thread-local variable `ancestralStack`, since this information cannot be derived from the current JVM call stack and it is needed in evaluating any `checkPermission` that does not terminate early by reaching a `doPrivileged` frame.

Table 4(a) shows the cost of the stack inspection primi-

Method call	doPrivileged	checkPermission	New thread
0μs	23.4μs	22.4μs	29.8μs

(a) Benchmarked cost of IRM_{Lazy} primitives (at stack depth 10).

	JVM	IRM_{SPS}	IRM_{Lazy}
Jigsaw	6.2%	20.1%	6.4%
`javac`	2.9%	46.2%	2.0%
`tar`	10.1%	3.0%	5.4%
MPEG	0.9%	72.5%	0.4%

(b) Overhead of JVM-resident, IRM_{SPS}, and IRM_{Lazy} implementations.

Table 4. Assessing the IRM_{Lazy} stack inspection implementation.

tives with IRM_{Lazy}. As with Table 2(b), reported measurements are averages from a synthetic benchmark that repeatedly performed the subject operation.

Notice that, except for method calls, the measured costs for each stack inspection primitive in Table 4(a) are higher than the IRM_{SPS} costs given in Table 2(b). These higher costs arise because the entire stack is now being copied by the implementations of all but the method call/return stack inspection primitives. Even so, for our benchmark applications, IRM_{Lazy} exhibits overall performance that is superior to IRM_{SPS} and that is competitive with Sun's JVM-resident implementation. This is seen in Table 4(b), and it is a consequence of method call/return invocations dominating performance of our benchmarks. Where IRM_{Lazy} performs better than the JVM-resident implementation, it is because of optimizations in our PSLang specification, which do a better job of eliminating redundant work in permission checking.[9]

6. Concluding Remarks

The idea of separating mechanism from the policy that directs this mechanism is advocated often. Java 2's support for the stack inspection access-control policy involves a mechanism (in the JVM) and the flexibility to direct that mechanism through policy files, protection domains, and permission classes (with their `implies` methods). Our IRM realizations of stack inspection actually draw a somewhat different line between policy and mechanism. With no JVM-resident mechanism, there is considerable flexibility about what policies can be enforced using the IRM approach and about when that choice of policy must be made.

This flexibility allows enforcement of policies that alter or extend what the JVM implements today. One might now contemplate remedying the various deficiencies in the Java 2 stack inspection access-control policy, allowing

- changing protection domains, permissions, and the `implies` method after execution of an application is commenced, enabling straightforward creation of new protection domains as execution proceeds;
- the coupling between protection domains and bytecode origin to be refined so that, for example, an application's state is used in determining the protection domain for code; and
- the operation of `doPrivileged` to be extended so that only a subset of the privileges in a protection domain are amplified in a block of code.

It now even becomes possible to enforce different security policies on different Java applications, raising questions about detecting and resolving incompatibilities between those policies. However, these questions about policy composition are independent of whether or not the IRM approach is being used to enforce policies.

The IRM approach is flexible because it allows security events and security updates to be associated with any application event. This degree of flexibility can be only approximated by wrapping security enforcement code around an interface, as done by Naccio [9] (for method calls) and Generic Software Wrappers [10] (for system calls). Software-based fault isolation (SFI) [18] enforces a memory protection policy by object-code editing, and recent work on distributed virtual machines also is concerned with enforcing security policies by code rewriting [17]. Clearly, the set of enforceable security policies is restricted if, as in this related work, only some—not all—potential security events can be monitored, only some security state maintained, and only some types of security updates supported.

Flexibility is a double-edged sword. The IRM approach is not only flexible enough to implement Java 2's stack inspection (in multiple ways!) and to implement a host of variants that address apparent limitations in the policy, but it is

[9] Similar optimizations are done in IRM_{SPS}.

also flexible enough to allow policies to be defined that have unanticipated consequences or vulnerabilities. We have no way to guarantee that our PSLang formulations of stack inspection are indeed the policy supported by Sun's distribution. To get such assurance, we would need a formal specification of Sun's stack inspection implementation and we would need a logic for PSLang specifications. Neither exists. But PSLang could easily be given a formal semantics in terms of security automata, and then it would not be difficult to reason about and/or simulate PSLang policies in order to gain confidence that they describe what is intended.

Even without a logic for reasoning about PSLang specifications, the exercise of formulating stack inspection in PSLang, a formal language, did prove enlightening. Writing the PSLang security updates forced us to ask questions about what really happens when security events occur. Surprising things about the semantics of stack inspection came to light:

- If a new thread is created from within a `doPrivileged` block then that thread will continue to enjoy amplified privileges—even though its code might not be within the scope of a `doPrivileged` block and even after its creator has exited from within the `doPrivileged`. This is because the new thread starts execution with a copy of its creator's call-stack (whose top frame is marked as being within the scope of a `doPrivileged`).

- When a class B extends some class A but does not override A's implementation of a method $foo()$, then the protection domain for A (and not B) will always be used by `checkPermission` for *foo*'s stack frame. Because B can extend A in ways that may affect the semantics of *foo*, (such as by overriding other methods), one might argue that the wrong protection domain is being consulted.[10]

Both of these "features" of stack inspection will become apparent to attentive readers of the PSLang formulations that appear in the appendices of [8]. This is not to say that there aren't also surprises in our PSLang formulations or there aren't aspects of the Java 2 behavior that we missed in constructing these formulations. But having—in just a few pages—a complete and rigorous description of the policy being enforced seems like a necessary condition for understanding that policy.

Acknowledgments

Discussions with Li Gong have been helpful as this work has evolved. We also thank Andrew Myers, Andrew Bernard, Michal Cierniak, Robert Grimm, and the program committee for comments on earlier drafts of this paper.

[10] The rationale for the choice that was made is given in [11, §3.11.3].

References

[1] Anders, J. Java MPEG Player. Fakultät für Informatik, Technische Universität Chemnitz, Chemnitz, Germany, `http://rnvs.informatik.tu-chemnitz.de/`.

[2] Anderson, J.P. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), Bedford, Massachusetts, October 1972, volume 2, 58–69.

[3] Baird-Smith, A. Jigsaw: An Object Oriented Server. W3C Note, World Wide Web Consortium, MIT Laboratory for Computer Science, Cambridge, Massachusetts, June 1996, `http://www.w3.org/Jigsaw/`.

[4] Cohen, G., J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. *Proceedings of 1998 Usenix Annual Technical Symposium*, (New Orleans, Louisiana, June 1998), 167–178.

[5] Endres, T. Java Tar Package. ICE Engineering, Inc., Lake Linden, Michigan, `http://www.ice.com/java/tar/`.

[6] Erlingsson, Ú. *The Inlined Reference Monitor Approach to Security Policy Enforcement*, Ph.D. thesis, Cornell University, Ithaca, New York, 2000.

[7] Erlingsson, Ú. and F.B. Schneider. SASI Enforcement of Security Policies: A Retrospective. *Proceedings 1999 New Security Paradigms Workshop* (Caledon Hills, Canada, September 1999), ACM Press, New York.

[8] Erlingsson, Ú. and F.B. Schneider. IRM Enforcement of Java Stack Inspection. Technical Report TR2000-1786, Computer Science Department, Cornell University, Ithaca, New York, February 2000, `http://cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/Display/ncstrl.cornell/TR2000-1786`.

[9] Evans, D. and A. Twyman. Policy-directed code safety. *Proceedings IEEE Symposium on Security and Privacy* (Oakland, California, May 1999), IEEE Computer Society, California, 32–45.

[10] Fraser, T., L. Badger, M. Feldman. Hardening COTS Software with Generic Software Wrappers. *Proceedings IEEE Symposium on Security and Privacy* (Oakland, California, May 1999), IEEE Computer Society, California, 2–16.

[11] Gong, L. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, Addison-Wesley, Menlo Park, California, 1999.

[12] Gosling, J., B. Joy, and G. Steele. *The Java Language Specification*, Addison-Wesley, Menlo Park, California, 1996.

[13] Liang, S. and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. *Proceedings of 1998 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications* (Vancouver, Canada, October 1998), SIGPLAN Notices 33(10), 36–44.

[14] Lindholm, T. and F. Yellin. *The Java Virtual Machine Specification*, 2nd edition. Addison-Wesley, Menlo Park, California, 1999.

[15] Saltzer J.H. and M.D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE 63*, 9 (Sept. 1975), 1278–1308.

[16] Schneider, F.B. Enforceable Security Policies. *ACM Transactions on Information and System Security 2*, 4 (February 2000). To appear.

[17] Sirer, E.G., R. Grimm, A.J. Gregory, B.N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (Kiawah Island, SC, Dec. 1999), ACM, 202–216.

[18] Wahbe, R., S. Lucco, T.E. Anderson, and S.L. Graham. Efficient Software-Based Fault Isolation. *Operating System Review*, 27(5), ACM Press, 1993.

[19] Wallach, D.S. *A New Approach to Mobile Code Security*, Ph.D. thesis, Princeton University, New Jersey, January 1999.

[20] Wallach, D.S. and E.W. Felten. Understanding Java Stack Inspection. *Proceedings 1998 IEEE Symposium on Security and Privacy* (Oakland, California, May 1998), IEEE Computer Society, California, 52–63.