Computers & Security (2005) **24**, 192—207

Computers & Security

www.elsevier.com/locate/cose

Performance of the Java security manager

Almut Herzog*, **Nahid Shahmehri**

Department of Computer and Information Science, Linköping University, 581 83 Linkoping, Sweden

Received 7 April 2004; revised 11 August 2004; accepted 18 August 2004

KEYWORDS
Java;
Performance;
Security;
Security Manager;
Access controller;
Permission;
Policy;
CPU execution time

Abstract The Java Security Manager is one major security feature of the Java programming language. However, in many Java applications the Security Manager is not enabled because it slows execution time. This paper explores the performance of the Java Security Manager in depth, identifies the permissions with the worst performance and gives advice on how to use the Security Manager in a more efficient way.

Our performance test shows that the CPU execution time penalty varies between 5% and 100% per resource access statement. This extreme range is due to the fact that some resource accesses are costly (such as file and socket access) and therefore hide the performance penalty for the access control check almost completely. The time penalty is much more noticeable with access to main memory resources (such as Java objects).

In order to achieve reasonable response times, it is of utmost importance to tune garbage collection because the Java Security Manager creates short-lived objects during its permission check. Also, the order of permissions in the policy file can be important.

© 2004 Elsevier Ltd. All rights reserved.

Introduction

Java is a popular programming language, especially in web applications, including applet and servlet technology. Java was designed for use on the Internet and contains strong support for containing untrusted code in a secure environment. One feature of Java's security architecture is the Security Manager. The Security Manager enforces the so-called Java sandbox, a reference monitor in which untrusted code can perform without harming other Java code, the Java virtual machine or the underlying operating system. Stepping out of the sandbox can be allowed if a policy exists that gives code additional access rights. The Security Manager is an optional feature; applet containers use it, but, by default, Java applications do not.

There is evidence that Java programs perform more slowly when the Java Security Manager is enabled (Venkatakrishnan et al., 2002; Herzog and

* Corresponding author.
E-mail addresses: almhe@ida.liu.se (A. Herzog), nahsh@ida.liu.se (N. Shahmehri).

0167-4048/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2004.08.006

Shahmehri, 2002). This is often used as an argument for not using the Security Manager at all or for the use of byte code editing (Pandey and Hashii, 1999). However, no hard data are available as to the actual performance penalty. Sun Microsystems Inc. formulates their findings as follows: ''We've seen some applications where turning off the security manager has helped to a small extent (3—5%) and others where it hasn't helped at all'' (Sun Microsystems, 2004). IBM refers to the performance of their own Java virtual machine in equally vague terms: ''...the installation of a security manager imposed a significant overhead (greater than 30 percent) to the execution of an important benchmark'' (Triplett, 2001). Both sources refer to the performance overhead for a complete application.

However, not every Java statement is subject to access control by the Security Manager; file access, socket access and screen access, etc. typically are (if the Security Manager is enabled). CPU-intense calculations are not. Thus, depending on the type of application, the performance findings with or without Security Manager will be very different. This work explores the performance overhead incurred by the Java Security Manager on single Java statements that *are* subject to access control by the Security Manager. For example, how long does a file access take with and without the Security Manager. We show where and why time is spent in the Java statement that is subject to access control by the Java Security Manager and what that means to applications that aim at making use of the Security Manager.

The following questions are answered:

- What is the memory penalty incurred by running an access check?
- What is the time penalty incurred by running an access check?
- How is performance influenced by
  - the number of protection domains on the call stack?
  - the size of the policy file?
  - different permissions, e.g. FilePermissions or SocketPermissions? Is each permission check of the same complexity?
  - different policies? Is there e.g. a difference in performance if the currently installed policy contains wildcards?
- What should be considered when an application makes use of the Security Manager?

The tests were performed with the SUN Java 2 Software Development Kit (J2SDK) 1.4.2 because its source code is available for analysis. Findings for this Java implementation may not be applicable to Java Virtual Machines of other vendors.

The structure of the paper is as follows. The next section contains an introduction to the Java security model and especially the Security Manager. Then the performed tests and platform are described which is followed by the results. Further the advice to application developers that want to make use of the Security Manager is presented. Last section concludes the paper. Appendix contains test data from the Solaris suite.

The Java Security Manager

The Java Security Manager is part of the Java security model, also named *Java sandbox*. Its goal is to provide ''a very restricted environment in which to run untrusted code'' (Gong, 1997). While the sandbox was originally designed to contain applets, as of Java 1.2 it is also used to provide a secured environment for any Java application.

The following parts make up the Java sandbox (Oaks, 2001; McGraw and Felten, 1999) (see also Fig. 1):

1. The *bytecode verifier* ensures at runtime that Java class files follow the rules of the Java language and helps enforce memory protections.
2. The *class loader* finds the bytecode for a particular class at runtime and defines the namespaces seen by different classes. The class loader protects the integrity of the Java security system in such a way that the security system cannot be replaced by untrusted classes.
3. The *access controller* allows or restricts access to protected resources by checking if a needed

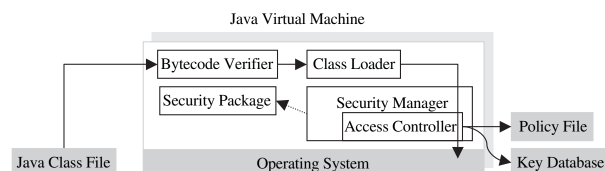

Figure 1 Java security architecture. The bytecode verifier checks the validity of Java class files. The class loader finds needed classes. The security manager wraps the access controller that decides whether user code has access to protected resources or not. The decision is based on the contents of the policy file. Signed user code is authenticated by the access controller using the security package and the key database.

permission exists or is implied by the currently installed policy. The policy contains positive permissions per codebase or signer (see Fig. 4). The access controller is only invoked if a *Security Manager* is installed. The Security Manager is a wrapper for the access controller. It is also the handle for programmers to implement different security policies but does also contain certain hard-coded policies, e.g. the policy that only system threads are subject to access control. The Security Manager is installed by a command line option to the Java virtual machine or by issuing `System.set Security Manager (new Security Manager())` within a Java class. It is possible to subclass the Security Manager, but the access controller is a final class.
4. The *security package* helps authenticate signed Java classes.
5. The *key database* contains a set of keys that are used by the access controller to verify the digital signature of a signed class file.

In this paper, we work exclusively with the Security Manager and access controller and their impact on the execution time of a Java application. It is obvious from a code inspection that the use of the Security Manager *must* incur a performance penalty. Fig. 2 shows an example of how access to a protected resource, here the system properties, is controlled using the Security Manager.

If a Security Manager exists in the Java virtual machine (the check is done in line 5 of Fig. 2), resource access is only granted after the access check in line 6 has returned successfully. If the check is unsuccessful a security exception is raised by the method in line 6 and the method is aborted before the actual resource access that occurs in line 8.

However, the checkPropertyAccess()-method in line 6 of Fig. 2 is not as simple as it looks. It hides a long call chain as displayed in Fig. 3. In Fig. 3, indentation and the numbers followed by colons in front of the method names indicate the call level.

```
 1  public final class System {
 2        ...
 3        public static String getProperty(String key) {
 4          ...
 5          if (securitymgr != null) {
 6            securitymgr.checkPropertyAccess(key);
 7          }
 8          return props.getProperty(key);
 9        }
10  }
```

Figure 2 Java code for accessing the protected resource of a system property.

For instance, the top level `java.lang.Security Manager.check Property Access` contains two calls, namely those indicated by `2:`, `java.util.Property Permission.⟨init⟩` and `java.lang.Security Manager.check Permission`. The PropertyPermission constructor ⟨init⟩ creates the permission that is to be checked. This permission is then passed to `java.lang.Security Manager.check Permission` (in line 9 and below) where the main work is done.

The actual access decision is made when the current policy, parsed from a policy file, is compared with the needed permission (cf. line 18 and below). A typical policy file with positive permissions is shown in Fig. 4.

Permissions stem from two abstract classes. Java defines the abstract superclass `java.security.Permission` and an abstract subclass `java.security.Basic Permission`. BasicPermissions do not normally contain action strings such as *read, write, listen, resolve* (see Fig. 4) but consist of a target name only (as *play* in the audio permission of Fig. 4). An exception is the PropertyPermission that is a BasicPermission with target *and* action strings. Examples of direct subclasses of Basic Permissions are AudioPermission, AWTPermission, PropertyPermission, and RuntimePermission. Direct subclasses of `java.security.Permission` are AllPermission, FilePermission, PrivateCredentialPermission, ServicePermission, SocketPermission and UnresolvedPermission. Refer to the Java documentation at http://java.sun.com/j2se/1.4.2/docs/api/index.html for more details on permissions.

Policy checking needs to be done for every protection domain on the call stack (Gong and Schemers, 1998). If, for example, library *x.jar* calls a method in *y.jar* that wants to open a socket, both *x.jar* and *y.jar* need to have the appropriate SocketPermission to succeed.

Permissions, protection domains and the contents of the policy file were important variables in our performance test of the Security Manager.

Test design

The tests were performed on a SUN Ultra SPARC 5, running Solaris 8 with 4 GB of RAM and four CPUs, using the SUN Java 2 Software Development Kit (J2SDK) 1.4.2 for the tests and its source code for code inspections. Running on a single CPU Solaris machine lead to many operating system artifacts in the time measurements. The tests were repeated

```
1    1: java.lang.SecurityManager.checkPropertyAccess
2      2: java.util.PropertyPermission.<init>
3        3: java.security.BasicPermission.<init>
4          4: java.security.BasicPermission.init
5            5: java.lang.String.charAt
6          4: java.security.Permission.<init>
7        3: java.util.PropertyPermission.init
8        3: java.util.PropertyPermission.getMask
9      2: java.lang.SecurityManager.checkPermission
10       3: java.security.AccessController.checkPermission
11         4: java.security.AccessController.getStackAccessControlContext
12         4: java.security.AccessControlContext.optimize
13           5: java.security.AccessController.getInheritedAccessControlContext
14         4: java.security.AccessControlContext.checkPermission
15           5: java.security.AccessControlContext.getDebug
16           5: java.security.ProtectionDomain.implies
17             6: java.security.Policy.getPolicyNoCheck
18             6: sun.security.provider.PolicyFile.implies
19               7: sun.security.provider.PolicyInfo.getPdMapping
20               7: java.util.Collections\$SynchronizedMap.get
21                 8: java.util.WeakHashMap.get
22                   ...
23               7: java.security.Permissions.implies
24                 8: java.security.Permissions.getPermissionCollection
25                   9: java.util.HashMap.get
26                     ...
27                 8: java.util.PropertyPermissionCollection.implies
28                   9: java.util.HashMap.get
29                     ...
```

Figure 3 Call chain for checking access to system properties.

on a 2.4 GHz Intel Xeon machine running Debian Linux 3.0 (1 GB of RAM, one CPU).

For checking the performance we used an operating system tool and a Java tool. The operating system tool was the Solaris `truss` utility for tracing system calls and measuring elapsed CPU time in system and user code (and `time` on Linux and `time` on Solaris to verify `truss` times). This tool was used for exact timing. The Java tool was the JVM-built-in `hprof` tool that makes use of the JVMPI (Java Virtual Machine Profiling Interface) for measuring CPU time spent in methods and threads. `hprof` is accessible through a command line option to the Java Virtual Machine. We also made use of `PerfAnal` (Meyers, 1999) to analyse the output generated by `hprof` in a graphical user interface. The Java tools were used for counting method invocation calls and bottleneck identification.

There were two major test suites. One was a test for checking different permission against different policies using the call `java.lang.SecurityManager.checkPermission()`. This test shows the time for a permission test without actual resource access in order to identify those permissions that are time-consuming to check. The other test monitored actual resource access under the control of the Security Manager and thus shows the normal use of the Security Manager.

All tests were run for one and two protection domains on the call stack. In order to arrive at significant CPU execution times, an access check or resource access would run in a loop. Each check was done with 20 samples and repetitions ranging between one and 20,000 or one and 50,000. The upper limit was chosen with respect to memory consumption to avoid garbage collection during the tests. Each test case was also run once with the `hprof sample` option and once with the `hprof times` option. Each test case contains a CPU consumer method that is constant for each repetition. This was intended to facilitate the comparison of

```
grant codeBase "file:/sw/jars/example1.jar" {
  permission javax.sound.sampled.AudioPermission "play";
  permission java.util.PropertyPermission "java.*", "read";
  permission java.io.FilePermission "/tmp/-", "read,write";
  permission java.net.SocketPermission "localhost:9000", "listen,resolve";
};

grant signedBy "bob" {
  permission java.security.AllPermission;
};
```

Figure 4 Typical Java policy file with one set of permissions for the code base at */sw/jars/example1.jar* and full permissions for all code signed by *bob*.

execution times in hprof with a known method (Section 'Comments on hprof' shows that this did not work).

All test cases were also run once with forced garbage collection at the beginning and the end of the Java program. Verbose output of the garbage collector then showed statistics about allocated and freed memory during program run. Memory statistics were also verified by running the test case with the `-Xaprof` option (Shirazi, 2003).

In the first test suite we tested the check for the following permissions: AWTPermission, RuntimePermission, DelegationPermission, PropertyPermission and the more complex FilePermission, PrivateCredentialPermission, ServicePermission and SocketPermission. Thus, the tested permissions comprise two classically BasicPermissions (AWT and Runtime), one more complex (Property) and all of the explicitly used complex subclasses of `java.security.Permission`. The remaining BasicPermissions are subclasses of BasicPermission with no distinct behaviour of their own; their behaviour is therefore the same as that of AWT- or RuntimePermission. Thus all significantly different permissions were evaluated.

Each permission check was run against different policies. In the first test case, only one permission instance of one class would appear in the policy file, i.e. there would be only one entry for a SocketPermission, PropertyPermission and so on, and that entry would be the needed match as an exact match (`RuntimePermission("exitVM")`). The second test case would be the same but with a wildcard match (`RuntimePermission("*")`). We also tested the permissions against a policy where the code was granted all permissions (`java.security.AllPermission`). In a second step, the permissions were checked against (a) a long policy file containing numerous entries for each permission class, both exact matches and wildcard matches and (b) an even longer policy file with permissions for 54 code bases and a total of 4521 permissions but exact matches for the needed permissions.

In the second test suite the resource accesses listed below were timed with and without a Security Manager. This was to see how an actual resource access, as in a real program, is influenced by the absence or presence of a Security Manager.

- Java-defined resources
  - Policy access using `Policy.getPolicy()` which makes use of the BasicPermission SecurityPermission. Policy access is an operation performed in memory, no access to operating system specific code is needed.
  - Property access using `System.getProperty(prop)` which makes use of the basic (but special) permission PropertyPermission. This is also an access check done in memory but the PropertyPermission is more complex.
  - Redirecting output using `System.setOut(System.out)` which makes use of the BasicPermission RuntimePermission. This is another BasicPermission to verify findings from policy access.
- Java-mediated operating system resources that make use of advanced permissions
  - file access using `new FileInputStream(filename)`,
  - socket access using `new ServerSocket(port)`.

These resource accesses were tested with the same policies as described with the first test suite. Thus the resource access test suite has also examples of both basic and advanced permissions.

Results

In the following, we present the results of our performance tests. Data are found in Tables 3—6. Complete test data as well as the test programs are available at www.ida.liu.se/~almhe.

Memory penalty

Using the Security Manager incurs a memory penalty of at least 40 bytes per resource access check (see Table 1). These 40 bytes are used for an access control object and a protection domain object. The memory startup penalty for using the Security Manager can be as low as 14 kB but is dependent on the size of the policy files that are needed. Policy files are not parsed at Java Virtual Machine startup but when the first resource access occurs. A large policy file needs a lot of memory. The long policy file with 54 code bases and 4521 permissions uses only 300 kB on disk but needs 1900 kB in memory.

Apart from this memory allocation during policy parsing, memory is also allocated in the constructor of the permission object that is about to be checked. As that permission object is created only once regardless of the number of protection domains on the call stack, having a second protection domain seldom uses more than 8 bytes of memory (which are spent on a larger protection domain object). Only PrivateCredentialPermissions and PropertyPermissions allocate memory during their

Table 1 Memory allocation in bytes for one resource access with and without Security Manager (SM) and with different policies

Test case	Protection domains		Allocated objects
	One	Two	
Policy			
No SM	0	0	
SM with any policy	40	48	AccessControlContext (ACC), ProtectionDomain (PD)
Setout			
No SM	0	0	
SM with any policy	64	72	RuntimePermission, ACC, PD
Property			
No SM	0	0	
SM with any policy	72	80	PropertyPermission, ACC, PD
Partial wildcard	393	722	4(8) String, 2(4) StringBuffer, 3(6) char[], PropertyPermission, ACC, PD
File			
No SM	80	80	
SM with any policy	184	192	File, FilePermission, FilePermissionCollection, ACC, PD
Socket			
No SM	228	228	
AllPermission, wildcard	480	488	3 String, StringBuffer, char[], SocketPermission, int[], ACC, PD
Exact	504	512	4 String, StringBuffer, char[], SocketPermission, int[], ACC, PD
Many codebases	574	582	5 String, StringBuffer, 2 char[], SocketPermission, int[], ACC, PD
Enorm-exact	739	747	6 String, 2 StringBuffer, 4 char[], 3 byte[], SocketPermission, int[], ACC, PD
PrivateCredentialPermission			only data from the permission check are used, no actual resource access
No SM	Not applicable		
AllPermission	40	48	Permission created at program start, ACC, PD
Wildcard	136	*240*	3(6) HashmapIterators, ACC, PD
Other policies	232	*432*	6(12) HashmapIterators, ACC, PD

The policy check on the top row of the table shows the minimum memory consumption for the Security Manager consisting of allocations for an access control context object and a protection domain object. Any startup memory penalty is omitted. Only objects relevant to the Security Manager are shown in the column *Allocated Objects*. Numbers in italics indicate that memory is allocated per protection domain.

implies()-methods,[1] which is invoked per protection domain (see Table 1).

The memory consumption of all BasicPermissions is about 64 bytes. This is so because BasicPermissions (with the exception of the PropertyPermission) consist of one, often constant, target. Some BasicPermissions are created only once during the lifetime of the Java Virtual Machine as static objects and need not be allocated for subsequent checks (for example the policy access permission in the first row of Table 1).

However, the memory consumption of complex permissions (e.g. file, socket, PrivateCredentialPermission) is related to the length of the string that represents the target of the permission that is to be checked. The FilePermission with target "/tmp/a" consumes less memory than the same FilePermission with target "/tmp/abc/def/ghi/jkl". Thus, the values in Table 1 for those permissions are only examples. The actual memory consumption depends on the specific permission passed to the access check.

Unlike target strings, actions are constant strings that are converted to a mask of type `int` upon the construction of the permission and no

[1] The implies()-method, implemented by each permission, decides whether a given permission contains or *implies* another permission and thus makes the actual access control decision. For instance, FilePermission("/tmp/*", "read,write") implies FilePermission("/tmp/a", "read") but FilePermission("/sw/emacs", "read") does not imply FilePermission("/sw/java", "read").

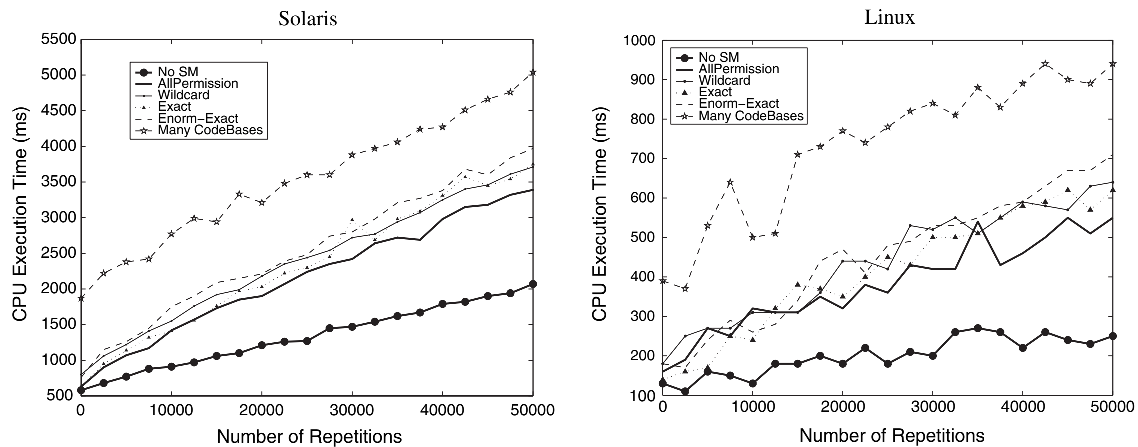

Figure 5 Typical performance for policy access containing a constant CPU consumer method for Solaris (left) and Linux (right). An AllPermission performs fastest of all permissions. Exact match and wildcard match are approximately equal. *Enorm-exact* contains other exact security permissions and is slightly slower. Worst is the exact match in a policy file with 54 code bases and 4521 permissions due to the high startup penalty. Note the remarkable difference of factor 5 for the absolute execution times between the two platforms. The more erratic curve for Linux may come from the higher resolution of its *y*-axis and/or from the shorter execution times that may be more difficult to measure with good precision.

subsequent operations are done on the strings. Thus, action strings do not have any impact on memory consumption.

Time penalty

Test data from two Unix platforms (Solaris and Debian Linux) show a great difference in absolute execution times but a great similarity in relative times (cf. Fig. 5). In the following, we will therefore use relative times (compare test cases with each other). We are rather confident that results on other operating systems will show similar relative results because very little platform dependent code is used in the Security Manager. However, the results are probably not applicable to Java implementations other than the used J2SDK from Sun. IBM's implementation, for instance, uses caching of permissions and has a special ''handling of file name representation to avoid the high cost of file name canonicalization unless needed'' (Triplett, 2001) that would result in quite different test results.

Using a Security Manager with the fastest of all permissions, namely the AllPermission, *non-operating system resource accesses* have an execution time that is approximately 100% slower than without the Security Manager enabled. However, time-consuming *operating system accesses* such as opening a file or opening a socket are not much influenced by the presence or absence of a Security Manager. File access is 30% times slower and socket opening is only 5% times slower under an AllPermission compared with no Security Manager. When run with permissions other than the All Permission, these execution times increase (but not much) depending on the permission and the effective policy. Note that these results must not be applied to a whole application. The test only used single resource access statements while a complete application would contain a lot of code that is unaffected by the absence or presence of a Security Manager.

It is noteworthy that a lengthy or even huge policy file does not impair the finding of a matching permission once the startup penalty has worn off. However, if there are many permissions of the same kind in one grant statement in the policy file (as in our ''enorm-exact'' test cases in Tables 3—6), performance is influenced by that number and sometimes by the order of the permissions in the policy file. A comparison of the performance for different policies is shown in Fig. 5 for access to the policy object protected by a SecurityPermission.

The additional execution time for a resource access statement under a Security Manager is spent mostly in `java.security.AccessController.getStackAccessControlContext` (see line 11 in Fig. 3), a costly native method. The native code is not very platform-specific; it simply performs an operation on the passed objects that must be performed outside of the Java sandbox.

Time is also spent finding the matching permission collection in the policy object (which is filled

```
1    1: java.lang.SecurityManager.checkPermission
2      2: java.security.AccessController.checkPermission
3        3: java.security.AccessController.getStackAccessControlContext
4        3: java.security.AccessControlContext.optimize
5          4: java.security.AccessController.getInheritedAccessControlContext
6        3: java.security.AccessControlContext.checkPermission
7          4: java.security.AccessControlContext.getDebug
8          4: java.security.ProtectionDomain.implies
9            5: java.security.Policy.getPolicyNoCheck
10           5: sun.security.provider.PolicyFile.implies
11             6: sun.security.provider.PolicyInfo.getPdMapping
12             6: java.util.Collections\$SynchronizedMap.get
13               7: java.util.WeakHashMap.get
14                 8: java.util.WeakHashMap.getTable
15                   ...
16             6: java.security.Permissions.implies
```

Figure 6 Call chain for system property access with AllPermission policy.

with data from the policy files upon the first resource access) and checking its permissions against the needed permission. The difference in execution time for different permissions comes from this latter part and depends largely on the implementation of the implies()-method of the different permissions and permission collections.

Different permissions

All permissions except the PrivateCredentialPermission have approximately the same execution time when checked against an exact match or wildcard match with only one matching permission in the policy. The PrivateCredentialPermission is almost 40% slower than the average of the other permissions for exact matches and 20% slower for wildcard matches.

Full wildcard matching takes on average 95% of the time of an exact match, and is thus a bit faster than exact matching. This is so because full wildcard permissions are detected at policy parsing time and remembered in a private variable. This private variable is checked prior to any other checks in the respective permission collection.

Partial wildcard matches such as `Property Permission ("user.*", "read")` or `Socket Permission ("*.google.com:*", "resolve")` perform slightly slower than exact matches. In the case of the PropertyPermission the increased execution time is related to the fact that strings are allocated for a partial wildcard match. This is not so for other partial permissions where the extra time is exclusively spent in the implies()-method of the respective permission (in String- and HashMap-operations).

All permissions exhibit the same execution time when run against an AllPermission policy. At first glance, it is surprising that the permission check is not much faster when run against an AllPermission policy. However, when one compares the call chain for an exact match (cf. Fig. 3) with the AllPermission call chain of Fig. 6, one notices that most of the calls are identical. Only the end is shortened. Instead of checking for implications in the Property Permission collection (line 27 in Fig. 3), the AllPermission is found in `java.security.Permissions.implies` (line 16 in Fig. 6) and the permission check returns successfully from there.

Permissions and permission collections

The execution time of the important implies()-method (see line 8 in Fig. 6 or line 16 in Fig. 3) depends eventually on the implementation of the implies()-method in the actual permission *and* on the implies()-method of the actual permission collection.

Permission collections are needed to successfully resolve permissions that are disseminated over more than one permission grant. If e.g. a SocketPermission is needed that allows *connect* as well as *accept*, one permission in the policy could allow the *connect*, another could allow the *accept*, and together they grant the needed permission.

A matching permission in the policy is found by making use of permission collections and the contained permissions.

Example. If a piece of code from the code base */sw/jars/bob.jar* wants to connect to the host www.ida.liu.se on port 9000 it needs the permission `p1` = new SocketPermission (''http://www.ida.liu.se:9000'', ''connect''). For the policy file in Fig. 7 the finding of the needed granted permission follows the following pattern: the collection of all SocketPermissions (as shown in Fig. 7) are acquired in `java.security.Permissions.implies()`. In the implies()-method of the SocketPermission collection, the implies()-method

```
grant codeBase "file:/sw/jars/bob.jar" {
  ...
  permission java.net.SocketPermission "localhost:1024-", "accept";
  permission java.net.SocketPermission "localhost:80", "listen";
  permission java.net.SocketPermission "www.ida.liu.se:*", "accept,connect";
  permission java.net.SocketPermission "www.google.com:*", "accept,connect";
  permission java.net.SocketPermission "www.vivisimo.com:*", "accept,connect";
  permission java.net.SocketPermission "www.isy.liu.se:*", "accept,connect";
  permission java.net.SocketPermission "*.sun.com:*", "accept,connect";
  ...
};
```

Figure 7 Part of a policy file with SocketPermissions. Full host names such as www.google.com are resolved to check if they result in the same IP-address as the needed permission.

of each granted permission is called on *p1* until a positive match is found.

The implies()-method for a SocketPermission is extremely complex in that it not only does a textual comparison between the needed and the granted permission but also tries to resolve hostnames in a DNS-lookup.

When searching for matching SocketPermissions, the order of the SocketPermissions in the policy file are important. Frequent hits should therefore be put at the *end* of the permission list for that code base in the policy file. This avoids time-consuming DNS look-ups that try to match the needed permission to the granted permission. In the example above, all full hostnames (www.isy.liu.se, www.vivisimo.com, www.google.com) are resolved in order to check if they result in the same IP-address as the needed www.ida.liu.se.

SocketPermissions have a long history of being complicated and difficult to handle because of their interaction with the DNS server. Table 2 gives references to SocketPermission problems in the Java bug database.

Table 2 References to SocketPermission problems in the Java bug database at http://developer.java.sun.com/developer/bugParade/

Bug ID	Description
4155463	Clients get access errors when loading applets through proxies.
4320895	SocketPermission.implies() broken when IP not accessible.
4414825	SocketPermission.implies seems wrong for unresolvable hosts.
4975882	Reverse DNS calls in SocketPermission undesirable.
5004073	Impossible to use Security Manager with unstable DNS.

Another complex permission is the PrivateCredentialPermission. Even though it makes use of a standard permission hashmap for finding an exact match (if any), the implies()-method is extremely time-consuming even in an exact match policy with only one matching entry. An allocation of 257 bytes for each permission check (all memory consumed by hash map iterators) contributes to the long execution time. Full wildcard matching performs somewhat better since it "only" needs eight iterator objects to verify the permission instead of 18(!) for an exact match.

None of the other permissions have equally complex and time-consuming implies()-methods. The BasicPermission implementation uses a quick hash map lookup in the implies()-method of the BasicPermission collection to check if a needed permission is granted. Thus, the order in the policy file is not important for BasicPermissions. The PropertyPermission collection provides for both wild card matching and action string matching in its implies()-method and is thus slightly more complex. PropertyPermissions, ServicePermissions and FilePermissions are traversed in the reverse order of their appearance in the policy file. Frequent hits should thus be placed at the end of the permission list in the policy file to shorten execution time.

Protection domains

When an object *x* from library *x.jar* invokes a method of an object *y* from library *y.jar* and *y* accesses a resource that is protected by the Security Manager, both *x.jar* and *y.jar* must have permission to do so. This means that the *protection domain* of both *x.jar* and *y.jar* must be checked for the needed permission. Thus, adding call chains certainly has an impact on the performance of the Security Manager.

Adding a second protection domain without a Security Manager prolongs the CPU execution time by about 3%. A second protection domain *with* the Security Manager prolongs the CPU

execution time on average by 5% compared to the execution with one protection domain. Least impact is noticed for full wildcard matches (1%). The largest impact is noticed for exact matches (9%). The worst exact match is the SocketPermission with an increase of 18%. The AllPermission incurs the same increase in CPU execution time as when run without Security Manager, namely 3%.

Policy file

The policy file is read from disk and parsed into a policy object the first time a policy decision is needed. If the policy file is long, this first access takes considerable time. For instance, running a simple AWTPermission check for one protection domain once against a policy that contains permissions for 54 code bases with a total of 4251 permissions takes 1940 ms (on Solaris). However, it takes 4780 ms for 20,000 runs (on Solaris). This means that even at 20,000 repetitions 40% of the execution time is spent in startup.

Java virtual machine options

Each permission check was repeated a few thousand times to arrive at significant execution times. It was thus mandatory to suppress garbage collection during time measurements because the performance of the garbage collector was not interesting at this point. Increasing the heap size to a minimum (and maximum) of 128 MB was sufficient for our tests. To provide for the many "young deaths", i.e. objects that die soon after their creation, the young generation size was increased to half of the heap (options to the Java Virtual Machine: `-Xmx128m -Xms128m -XX:NewRatio=1`).

Prior to this adjustment, when using the Java Virtual Machine default settings, heavy garbage collection was occurring at all times, making time measurements useless and taking a lot of time. Memory tuning is thus especially advisable when running the Java Virtual Machine with a Security Manager. Useful help on monitoring and tuning garbage collection can be found in a Sun white paper (Sun Microsystems, 2003). This paper is not specific to the Security Manager; it applies to any application that has performance problems due to excessive invocations of the garbage collector.

Comments on hprof

When choosing the measurement tools for the test, there were high hopes for the Java built-in hprof tool that promised time measurements on a per-method basis. However, it was soon noticed that time measurements with the `hprof` tool could not be used to determine the performance without `hprof`—either in a relative or absolute manner; i.e. there is no simple, reliable relation between the execution time measured with `hprof` and the execution time measured with operating system tools `truss` or `time`. This is because the performance of `hprof` depends on the needed depth of profiling and the number of method calls on the stack—and not solely on the performance of the application that is profiled. The more methods that are called by the application and its called subroutines, the longer the execution takes in `hprof`. However, in our test cases it held true that applications A (fast as measured with an operating system timer), B (medium), and C (slow) will perform accordingly in `hprof` but not with the same proportions. Consequently, the presented test results rely on the operating system tools `truss` or `time` for time measurements.

However, `hprof` is useful for identifying bottlenecks. The `hprof times` option shows the exact number of times a method was invoked while the `hprof sample` option allows visual call chain analysis with the `PerfAnal` tool.

Advice to application developers

This section contains advice for application developers that work on applications that would benefit from the access control provided by the Security Manager (e.g. that run code from more or less untrusted sources). The suggestions do not only make use of the performance findings but includes also related items.

- Find out if the target application contains a lot of access to Security Manager-mediated resources or not. This is best done by subclassing the Security Manager with a class that does nothing but count the times the checkPermission()-method of the Security Manager was invoked and possibly keep track of the permissions that were about to be checked. The application should then be tested for a suitable length of time and the collected data evaluated.
- The Security Manager should probably *not* be enabled if
  - there are frequent SocketPermission requests for many different hosts,

  - there are frequent PrivateCredentialPermission requests,
  - the policy files are huge and the application is restarted too frequently for the startup penalty to wear off,
  - the application contains native library calls in the jar-file that will be subject to the Security Manager. Code in native libraries is beyond the control of the Security Manager, and guarding the Java layer while the native code layer has full access is not reasonable.
  - the policy files reside on a file system that can easily be compromised on the operating system level,
  - maintenance of policy files is too cumbersome (due to e.g. code changing often, many different jar-files, etc.),
  - all code owners demand (and receive) the AllPermission for their application,
  - the application is already experiencing performance problems. The Security Manager will certainly not speed up the application.
- The Security Manager can probably be enabled if
  - the previous list does not apply,
  - the application makes little use of resources that are protected by the Security Manager,
  - the application uses mostly BasicPermissions,
  - there is more heap available in case the heap size must be increased due to garbage collection problems,
  - the policy file can be tuned so that the most frequent FilePermissions, PropertyPermissions, ServicePermissions and SocketPermissions can be put at the end.
- When designing and implementing proprietary permissions that are specific for an application-defined resource, it is of great importance that the implies()-method is implemented efficiently and without allocating memory (if possible). This is the one bottleneck in the Security Manager that the application developer can control.

Conclusion

We have examined the performance of the Security Manager for Sun's Java implementation in depth. Test results show that there is an execution time penalty of approximately 100% for Java-defined resources when the resource access runs under a Security Manager. The resource access to operating system resources (files, sockets) is so expensive that it hides the time penalty of the Security Manager almost completely.

PrivateCredentialPermissions and SocketPermissions perform worst. FilePermissions, though seemingly complex, perform better than Property Permissions and rank fine among the BasicPermissions. Pure BasicPermissions perform the best.

All permission checks have the same execution time under an AllPermission policy, but that execution time is still closer to the execution time of a full wildcard match than to the execution time without Security Manager.

It is useful to tune the policy file so that frequent matches of the same permission class are put at the end of the enumeration of permissions of that class.

If the application appears unresponsive, this may be the result of excessive garbage collections. As the Security Manager at times creates short-lived objects it is especially useful to increase the young generation size and also give the application as much heap as possible.

In a server environment (e.g. web servers) where the policy file is possibly large due to many code bases, it is useful to force parsing of the policy file at startup time to avoid a costly parsing when the first permission check occurs.

This performance test shows that the performance of the Security Manager may not be as bad as sometimes assumed and that general statements about the impact of the Security Manager on a whole application are rather useless. Performance depends on the frequency of Security Manager-related access control and on the type of permissions that are needed. A performance boost can be achieved by memory tuning and, to a lesser degree, the right order of permissions in the policy file. Using the access control of the Java Security Manager is a clean and secure design and should be given priority over other solutions as long as the hard-coded and built-in resource checks are sufficient.

Appendix

The gist of the performance data is contained in the following tables. All data shown here come from the test suite on the Solaris operating system using the truss utility. Tables 3 and 4 contain data from the test of running permissions through `java.lang.SecurityManager.checkPermission()` (test suite 1). Tables 5 and 6 contain data about execution times for actual resource access (test suite 2).

Table 3 CPU execution times in milliseconds on the Solaris operating system for checking different BasicPermissions (AWTPermission, RuntimePermission, PropertyPermission, DelegationPermission) including the time used for running a control loop

Test case	# Repetitions				$y = mx + b$	
	1	10,000	25,000	50,000	m	b
P-AWT-one-allperm	670	1180	1930	3260	0.051636	718.09
P-AWT-one-enorm-exact	810	1560	2380	3830	0.057345	942.97
P-AWT-one-exact	760	1520	2320	3710	0.057309	903.41
P-AWT-one-manycb	1940	2520	3510	4780	0.056571	2005.2
P-AWT-one-null	460	870	1250	1970	0.028675	527.85
P-AWT-one-wildcard	720	1450	2380	3530	0.054187	872.89
P-AWT-two-allperm	640	1170	2050	3320	0.052094	758.09
P-AWT-two-enorm-exact	850	1620	2490	3930	0.059927	1010.3
P-AWT-two-exact	790	1600	2520	4020	0.061429	938.03
P-AWT-two-manycb	1900	2680	3520	5060	0.059979	2060.5
P-AWT-two-null	470	840	1250	1960	0.030795	524.38
P-AWT-two-wildcard	810	1460	2430	3830	0.058852	884.83
P-Run-one-allperm	660	1250	2060	3210	0.050364	779.43
P-Run-one-enorm-exact	870	1500	2440	3830	0.057351	953.32
P-Run-one-exact	800	1520	2260	3640	0.056384	910.81
P-Run-one-manycb	1830	2570	3610	4770	0.055865	1997.1
P-Run-one-wildcard	790	1430	2240	3590	0.055205	907.91
P-Run-two-allperm	600	1260	2050	3310	0.052525	769.69
P-Run-two-enorm-exact	800	1740	2430	3810	0.058556	997.95
P-Run-two-exact	790	1640	2360	3920	0.060478	921.32
P-Run-two-manycb	1780	2560	3450	5010	0.064114	1929.5
P-Run-two-wildcard	830	1490	2340	3660	0.056197	911.68
P-Prop-one-allperm	590	1320	1890	3190	0.050712	743.11
P-Prop-one-enorm-exact	830	2020	2340	3720	0.051782	1110.2
P-Prop-one-enorm-partwildcard	870	1680	2710	4120	0.062125	1063.5
P-Prop-one-exact	770	1560	2310	3700	0.05574	931.2
P-Prop-one-manycb	1870	2540	3360	5210	0.058265	1923.8
P-Prop-one-partproperty	780	1610	2450	4190	0.061958	949.55
P-Prop-one-wildcard	720	1450	2300	3710	0.056161	848.3
P-Prop-two-allperm	660	1360	2020	3420	0.052265	765.71
P-Prop-two-enorm-exact	810	1640	2420	3840	0.057029	1026.1
P-Prop-two-enorm-partwildcard	890	1690	2790	4380	0.068878	1063.7
P-Prop-two-exact	800	1500	2510	3930	0.060499	927.47
P-Prop-two-manycb	1900	2690	3480	4930	0.058868	2032.1
P-Prop-two-partproperty	750	1840	2700	4450	0.068681	988.16
P-Prop-two-wildcard	760	1520	2440	3870	0.06001	906.35
P-Deleg-one-allperm	660	1280	2070	3280	0.051112	737.39
P-Deleg-one-enorm-exact	850	1530	2480	3850	0.057652	977.69
P-Deleg-one-exact	810	1490	2620	3770	0.058218	922.11
P-Deleg-one-manycb	1850	2810	3380	4810	0.054488	2072.5
P-Deleg-two-allperm	630	1220	2090	3370	0.052779	718.09
P-Deleg-two-enorm-exact	810	1660	2690	4530	0.070078	962.27
P-Deleg-two-exact	710	1480	2410	4080	0.06293	897.17
P-Deleg-two-manycb	1800	2760	3660	5310	0.064571	2010.9

Note that a delegation permission does not support wildcard matching.

Table 4 CPU execution times in milliseconds on the Solaris operating system for checking subclasses of `java.security.Permission` (FilePermission, PrivateCredentialPermission, ServicePermission, SocketPermission) including the time used for running a control loop

Test case	# Repetitions				$y = mx + b$	
	1	10,000	25,000	50,000	m	b
P-File-one-allperm	640	1210	2090	3290	0.051538	739.6
P-File-one-enorm-exact	800	1540	2280	3630	0.053912	957.87
P-File-one-enorm-partwildcard	980	1570	2420	3690	0.054156	1045.1
P-File-one-exact	760	1430	2260	3600	0.054691	922.67
P-File-one-longfile	770	1460	2500	3780	0.058374	902.02
P-File-one-manycb	1840	2630	3260	4700	0.053881	1999.1
P-File-one-wildcard	830	1470	2340	3660	0.056592	898.47
P-File-two-allperm	600	1240	2050	3290	0.051813	739.39
P-File-two-enorm-exact	810	1600	2460	3820	0.057953	962.54
P-File-two-enorm-partwildcard	890	1650	2590	4060	0.060099	1017
P-File-two-exact	770	1610	2330	3720	0.05587	981.29
P-File-two-longfile	820	1470	3020	4070	0.063039	964.44
P-File-two-manycb	1770	2580	3490	4680	0.057657	1981.4
P-File-two-wildcard	830	1520	2390	3890	0.05961	936.35
P-Priv-one-allperm	650	1280	2010	3280	0.052192	731.33
P-Priv-one-enorm-exact	850	1790	2890	4820	0.077834	971.7
P-Priv-one-enorm-partwildcard	860	1810	3050	4710	0.078151	1042.8
P-Priv-one-exact	810	1680	3070	4770	0.078052	979.58
P-Priv-one-manycb	1920	2810	3960	6280	0.080301	1996.2
P-Priv-one-wildcard	750	1570	2590	4300	0.066042	931.28
P-Priv-two-allperm	630	1340	2050	3340	0.052795	733.41
P-Priv-two-enorm-exact	830	1970	3530	5970	0.099979	1034.2
P-Priv-two-enorm-partwildcard	730	2120	3550	5860	0.099558	1030.9
P-Priv-two-exact	720	1980	3400	5960	0.10174	928.77
P-Priv-two-manycb	1860	3100	4560	7090	0.099309	2082.4
P-Priv-two-wildcard	820	1750	2910	4520	0.072338	995.3
P-Serv-one-allperm	620	1300	2050	3270	0.052119	727.91
P-Serv-one-enorm-exact	800	1490	2480	3920	0.059268	956.35
P-Serv-one-exact	690	1520	2330	3710	0.057335	918.47
P-Serv-one-manycb	1890	2550	3380	4740	0.054945	2028.2
P-Serv-one-wildcard	690	1440	2360	3800	0.056681	851.98
P-Serv-two-allperm	660	1330	1980	3310	0.052831	748.22
P-Serv-two-enorm-exact	810	1590	2600	4490	0.067704	978.76
P-Serv-two-exact	820	1640	2470	3900	0.059605	955.05
P-Serv-two-manycb	1770	2730	3570	5130	0.062971	2016.1
P-Serv-two-wildcard	740	1540	2430	3940	0.06201	909.68
P-Sock-one-allperm	580	1260	2040	3300	0.052104	744.49
P-Sock-one-enorm-exact	810	1690	2460	4140	0.060857	950.89
P-Sock-one-enorm-partwildcard	950	1700	2540	4070	0.064468	1032.1
P-Sock-one-exact	740	1480	2360	4310	0.058608	930.46
P-Sock-one-manycb	1910	2540	3380	4710	0.055719	2007.4
P-Sock-one-wildcard	820	1570	2260	3630	0.054306	952.76
P-Sock-one-wildhost	820	1590	2700	3800	0.056525	1004.9
P-Sock-one-wildsocket	760	1620	2390	3730	0.055345	987.26
P-Sock-two-allperm	630	1350	2230	3430	0.052997	737.87
P-Sock-two-enorm-exact	810	1650	2610	4810	0.071725	933.95
P-Sock-two-enorm-partwildcard	910	1740	2860	4530	0.071231	1069.1
P-Sock-two-exact	870	1570	2480	3950	0.058686	991.85
P-Sock-two-manycb	1840	2700	3600	4970	0.060135	2071.8
P-Sock-two-wildcard	780	1620	2360	4040	0.061522	893.79
P-Sock-two-wildhost	810	1640	2510	3950	0.06053	996.22
P-Sock-two-wildsocket	810	1580	2510	4040	0.062447	953.53

Table 5 CPU execution times in milliseconds on the Solaris operating system for access to Java-mediated resources (the policy object, resetting of stdout, access to properties) including the time used for running a control loop

Test case	# Repetitions				$y = mx + b$	
	1	10,000	25,000	50,000	m	b
R-Pol-one-allperm	630	1420	2240	3390	0.053366	827.22
R-Pol-one-enorm-exact	760	1750	2480	3970	0.06014	1023.1
R-Pol-one-exact	630	1410	2300	3750	0.060748	818.86
R-Pol-one-manycb	1870	2770	3600	5040	0.057423	2097.7
R-Pol-one-null	580	910	1270	2070	0.028525	618.28
R-Pol-one-wildcard	800	1550	2440	3710	0.056281	981.03
R-Pol-two-allperm	650	1390	2350	3530	0.055678	799.42
R-Pol-two-enorm-exact	830	1630	2600	4130	0.062332	1050.2
R-Pol-two-exact	630	1490	2490	3940	0.062582	837.77
R-Pol-two-manycb	1820	2660	3830	5240	0.063517	2079.2
R-Pol-two-null	550	940	1300	2050	0.028	650.45
R-Pol-two-wildcard	800	1570	2430	3780	0.058644	987.17
R-Setout-one-allperm	620	1380	2400	3730	0.059891	747.91
R-Setout-one-enorm-exact	810	1600	2620	4240	0.067096	977.29
R-Setout-one-exact	670	1490	2530	4090	0.068312	788.81
R-Setout-one-manycb	1890	2760	3640	5310	0.066665	2029
R-Setout-one-null	460	810	1310	2100	0.033247	515.46
R-Setout-one-wildcard	780	1690	2510	4050	0.065501	905.74
R-Setout-two-allperm	640	1410	2290	3760	0.061886	768.03
R-Setout-two-enorm-exact	900	1780	2790	4690	0.071143	1008
R-Setout-two-exact	610	1610	2500	4210	0.069429	838.5
R-Setout-two-manycb	1900	2810	3890	5710	0.071174	2050.6
R-Setout-two-null	460	850	1410	2170	0.032519	554.12
R-Setout-two-wildcard	700	1700	2560	4240	0.068727	915.56
R-Prop-one-allperm	560	1290	2220	3890	0.059694	748.08
R-Prop-one-enorm-exact	820	1630	2580	4170	0.064675	983.53
R-Prop-one-enorm-partwildcard	960	2030	2920	4850	0.073361	1173.5
R-Prop-one-exact	650	1510	2350	3760	0.059501	832.41
R-Prop-one-manycb	1840	2510	3540	5110	0.062062	2018.9
R-Prop-one-null	540	830	1290	2080	0.030571	530.92
R-Prop-one-partproperty2	790	1770	2970	4940	0.078951	955.68
R-Prop-one-wildcard	790	1620	2630	4220	0.063756	954.61
R-Prop-two-allperm	640	1480	2370	4210	0.062649	764.18
R-Prop-two-enorm-exact	840	1650	2720	4490	0.070987	974.78
R-Prop-two-enorm-partwildcard	940	1920	3130	5180	0.082878	1081.8
R-Prop-two-exact	670	1500	2590	4420	0.068005	824.09
R-Prop-two-manycb	1880	2820	3690	5650	0.067829	2102.3
R-Prop-two-null	520	900	1260	2120	0.03159	546.89
R-Prop-two-partproperty2	770	1960	3340	5410	0.087839	1048.2
R-Prop-two-wildcard	750	1640	2580	4380	0.068592	947.98

A test case name has the following syntax

`[P|R]-[`*Name:*`]-[one|two]-[`*Policy*`|null]`

where `P` or `S` denote whether a permission (`P`) is tested or a resource (`S`). *Name:* denotes a short test name, either an abbreviation of the tested permission or the resource access statement. one or two denote the number of protection domains for which the test was run. *Policy* denotes the policy file that was used where null denotes that no Security Manager was used.

The policies used are described below and can be accessed at www.ida.liu.se/~almhe.

- allperm: The AllPermission.
- exact: The policy file is short and contains exactly the needed permissions for each protection domain.

Table 6 CPU execution times in milliseconds on the Solaris operating system for access to operating system resources (files and sockets) including the time used for running a control loop

Test case	# Repetitions				$y = mx + b$	
	1	4000	10,000	20,000	m	b
R-File-one-allperm	560	3130	5840	11,550	0.51261	754.81
R-File-one-enorm-exact	920	2830	5400	10,690	0.49647	979.59
R-File-one-enorm-partwildcard	790	3030	6140	11,430	0.50432	1025.3
R-File-one-exact	600	3200	5950	10,190	0.4941	940.37
R-File-one-manycb	1930	4190	7810	12,400	0.51256	2121
R-File-one-null	490	1870	4140	8410	0.3904	581.3
R-File-one-tmpasterisk	680	2960	6460	10,840	0.50668	937.03
R-File-one-tmpminuswildcard	780	3140	5850	11,300	0.48819	1012.8
R-File-one-wildcard	760	2760	6340	11,780	0.52126	872.12
R-File-two-allperm	670	3280	5800	11,530	0.50938	820.01
R-File-two-enorm-exact	810	3000	5680	11,460	0.53183	879.25
R-File-two-enorm-partwildcard	830	3190	6510	11,930	0.51274	1059.2
R-File-two-exact	730	2670	5860	12,140	0.5203	828.87
R-File-two-manycb	1940	4360	7400	12,630	0.5146	2180.7
R-File-two-null	510	2240	4730	9100	0.42858	518.97
R-File-two-tmpasterisk	730	3090	5530	10,380	0.47017	1028.3
R-File-two-tmpminuswildcard	730	2990	5640	11,690	0.50916	794.12
R-File-two-wildcard	800	3270	5920	10,790	0.50061	1142.9
R-Sock-one-allperm	640	5170	11,820	20,900	1.0181	1072.3
R-Sock-one-enorm-exact	870	5780	12,250	22,100	1.0782	1544.5
R-Sock-one-enorm-partwildcard	810	5380	12,300	23,390	1.1011	1368.7
R-Sock-one-exact	690	5660	12,090	21,950	1.0421	1341.8
R-Sock-one-manycb	1850	7020	12,710	22,730	1.0441	2495
R-Sock-one-null	550	5000	10,420	20,160	0.94245	848.8
R-Sock-one-wildcard	780	5600	11,520	21,550	0.99932	1431.9
R-Sock-one-wildhost2	760	5400	10,940	20,940	0.98901	1179.8
R-Sock-one-wildsocket	1350	5270	11,910	21,970	1.0241	1298.9
R-Sock-two-allperm	650	5250	11,240	21,650	1.0357	993.99
R-Sock-two-enorm-exact	860	5760	13,080	23,100	1.1082	1413.7
R-Sock-two-enorm-partwildcard	930	6340	12,480	22,990	1.1116	1579.8
R-Sock-two-exact	650	5600	11,830	22,890	1.095	1082.4
R-Sock-two-manycb	1910	6420	13,290	24,090	1.0698	2270.2
R-Sock-two-null	680	4210	9260	20,870	0.99097	336.89
R-Sock-two-wildcard	850	5000	11,400	22,660	1.0713	1160.6
R-Sock-two-wildhost2	780	5750	11,220	20,930	0.99178	1204.1
R-Sock-two-wildsocket	830	5540	11,520	21,120	1.003	1349.6

- wildcard: The policy file is short and contains the needed permissions for each protection domain as wildcard permissions.
- enorm-exact: The policy file is 131 lines long and contains six exact permissions of each used permission class.
- enorm-partwildcard: The policy file is 146 lines long and contains about six partial wildcard permissions for each used class. As partial wildcards are not supported by all permissions this is only tested for PropertyPermissions, FilePermissions, PrivateCredentialPermissions and SocketPermissions.
- manycb: The policy file contains 54 code bases and 4521 permissions but the needed permissions are exact matches. It is like the *exact* policy but with more code bases in the file.
- partproperty, partproperty2: The wildcard policy file but with one partial match PropertyPermission. Only used for PropertyPermission and property access tests.
- wildhost, wildhost2, wildsocket: The wildcard policy file but with one partial match SocketPermission. Only used for SocketPermission and socket access tests.

- longfile, tmpasterisk, tmpminus: The wildcard policy file but with one special FilePermission, either *a-long-path-name*/-, /tmp/* or /tmp/-.

All execution times are given in milliseconds. For each test four timings are supplied. Those are not average values but what was measured for the given repetition. The linear formula $y = mx + b$ supplies the execution time y for the number of repetitions x calculated from all 20 measurements and not only from the supplied four samples.

References

Gong L. Java 2 platform security architecture, <http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html>; 1997.

Gong L, Schemers R. Implementing protection domains in the Java development kit 1.2. In: Proceedings of the Internet society symposium on network and distributed system security (NDSS'98); 1998. p. 125–34.

Herzog A, Shahmehri N. Using the Java sandbox for resource control. In: Proceedings of the 7th nordic workshop on secure IT systems (NordSec'02). Karlstad University; November 2002. p. 135–47.

McGraw G, Felten EW. Securing Java: getting down to business with mobile code. Wiley & Sons; 1999.

Meyers N. Java programming on Linux. Waite Group Press; 1999.

Oaks S. Java security. 2nd ed. O'Reilly & Associates, Inc; 2001.

Pandey R, Hashii B. Providing fine-grained access control for Java programs. In: Guerraoui R, editor. Proceedings of the 13th European conference for object-oriented programming (ECOOP'99). Lecture notes in computer science 1628. Springer-Verlag; June 1999. p. 449–73.

Shirazi J. Java performance tuning. 2nd ed. O'Reilly & Associates, Inc; January 2003.

Tuning garbage collection with the 1.42 java virtual machine, <http://java.sun.com/docs/hotspot/gc1.4.2>; 2003.

FAQ about SUN ONE application server performance, <http://java.sun.com/docs/performance/appserver/AppServerPerfFaq.html>; 2004.

Triplett D. Spotlight on Java Performance, <http://www.106.ibm.com/developerworks/ibm/library/j-berry/>; December 2001.

Venkatakrishnan V, Peri R, Sekar R. Empowering mobile code using expressive security policies. Proceedings of the new security paradigms workshop (NSPW'02), ACM Press; September 2002. p. 61–8.

Almut Herzog received her M.S. in Medical Informatics from Heidelberg University, Germany, in 1994. Between 1994 and 1999 she worked as a software developer in the area of medical information systems. She is currently pursuing her Ph.D. studies at the Department of Computer and Information Science, Linköping University, Sweden. Her research interest is in computer security, and specifically Java security and security policy management.

Nahid Shahmehri received her Ph.D. in 1991, in the area of programming environments. Since 1994 her research activities have been concerned with various aspects of engineering advanced information systems, e.g. security. Examples of current projects are trust in middleware for peer-to-peer-based applications and Digital Rights Management for content and software distribution. She has been a full professor in Computer Science at Linköping University since 1988. She is Chairperson of the Swedish Section of the IEEE Chapter for Computer/Software and subject area editor for database systems in the Journal of Systems Architecture.

Available online at www.sciencedirect.com