Michael Golm, Meik Felser,
Christian Wawersich, Jürgen Kleinöder

A Java Operating System as the Foundation of a Secure Network Operating System

Technical Report TR-I4-02-05 ◆ August 2002

Department of Computer Sciences 4
Distributed Systems and Operating Systems

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Univ. Erlangen-Nürnberg • Informatik 4 • Martensstr. 1 • 91058 Erlangen • Germany
Phone: +49.9131.85.27277 • Fax: +49.9131.85.28732
E-Mail: i4@informatik.uni-erlangen.de • URL: http://www4.informatik.uni-erlangen.de

TECHNISCHE FAKULTÄT
(Faculty of Engineering Sciences)

A Java Operating System as the Foundation of a Secure Network Operating System

Michael Golm, Meik Felser, Christian Wawersich, Jürgen Kleinöder
University of Erlangen-Nuremberg
Dept. of Computer Science 4 (Distributed Systems and Operating Systems)
Martensstr. 1, 91058 Erlangen, Germany
{golm, felser, wawersich, kleinoeder}@informatik.uni-erlangen.de

Abstract

Errors in the design and implementation of operating system kernels and system programs lead to security problems that very often cause a complete breakdown of all security mechanisms of the system.
We present the architecture of the JX operating system, which avoids two categories of these errors. First, there are implementation errors, such as buffer overflows, dangling pointers, and memory leaks, caused by the use of unsafe languages. We eliminate these errors by using Java—a type-safe language with automatic memory management—for the implementation of the complete operating system. Second, there are architectural errors caused by complex system architectures, poorly understood interdependencies between system components, and minimal modularization. JX addresses these errors by following well-known principles, such as least-privilege and separation-of-privilege, and by using a minimal security kernel, which, for example, excludes the filesystem.
Java security problems, such as the huge trusted class library and reliance on stack inspection are avoided. Code of different trustworthiness or code that belongs to different principals is separated into isolated domains. These domains represent independent virtual machines. Sharing of information or resources between domains can be completely controlled by the security kernel.

1 Introduction

There are two categories of errors that cause the easy vulnerability of current systems. The first are implementation errors, such as buffer overflows, dangling pointers, and memory leaks, which are caused by the prevalent use of unsafe languages in current systems. This becomes dangerous when an OS relies on a large number of trusted programs. From the top ten CERT notes (as of January 2002) with highest vulnerability potential six are buffer overflows [4], [5], [6], [7], [8], [9], two relate to errors checking user supplied strings that contain commands thus allowing the user to run arbitrary commands with root privilege [10], [11], one executes commands in emails [12], and one is an integer overflow [13]. The six buffer overflow vulnerabilities could have been avoided by using techniques described by Cowan et al. [15]. However, not all overflow attacks can be detected and the authors recommend the use of a type-safe language.

An argument that is often raised against type-safe systems and software protection is that the compiler must be trusted. We think that this is not a very strong argument for the following three reasons. (i) Traditional systems, such as Unix, also use compilers to compile trusted components, like the kernel and system programs. Security in such a system relies on the assumption that the C compiler contains no bugs or trojan horses [61]. (ii) Only the compiler backend that translates the type-safe instruction set into the instruction set of the processor and the verifier that guarantees type-safety must be trusted. (iii) The additional effort that must be put into the verification of two components—the compiler backend and the verifier—pays off with reduced verification effort for many trusted system programs. Most vulnerabilities in current systems are caused not by bugs in the kernel but by bugs in system programs.

The second category of errors—the architectural errors— is more difficult to tackle. The three CERT notes related to the execution of commands in strings and emails are critical because the vulnerable systems violate the principle of least-privilege [52]. Thus, in current mainstream systems it is not the question whether the proper security policy is used, but whether security can be enforced at all [39]. Violations of the principle of *least-privilege*, an uncontrolled cumulation of functionality, many implementation errors, complex system architectures, and poorly understood interrelations between system components make current systems very vulnerable. This is a problem that affects all applications, because applications are built on top of an operating system and can be only as secure as its trusted programs and the underlying kernel.

As it will never be possible to develop software of moderate complexity that is free of errors one must assume that every complex application contains security critical errors. The realization that these errors can not be avoided in current systems led to the proliferation of firewalls that are responsible to shield potentially vulnerable systems from potentially dangerous traffic. Application developers and deployers react to the restriction of a firewall by tunneling traffic over open ports, for example the http port 80. The security community reacts by building traffic analyzers that analyze the TCP stream and the protocols above TCP and http. As it becomes more and more expensive to cure the symptoms it becomes more attractive to fix the deeper underlying causes of the security problems.

It is well understood that the unsafe nature of the languages C and C++ is the reason for many of today's security problems. There are several projects that try to develop a safe dialect of C. One of these projects created a safe dialect of C, called Cyclone-C [34]. Although Cyclone-C looks similar to C it is not possible to recompile an existing non-trivial C program, such as an OS kernel, without changes. Using Java instead of a Cyclone-C means that it is more difficult to port C programs, but allows to run the large number of existing Java programs without modifications. Furthermore, Cyclone-C programs have a similar performance overhead as Java programs.

There is still the problem that basing the protection on type-safety ties the system to a certain language and type system. But this seems to be no problem at all. Although the Java bytecode was not designed as the target instruction set for languages other than Java, there is a large number of languages that can be compiled to Java bytecode. Examples are Python [50], Eiffel [48], Tcl [35], Scheme [42], Prolog [20], Smalltalk [56], ADA95 [26], and Cobol [47].

Java allows developing applications using a modern object-oriented style, emphasizing abstraction and reusability. On the other hand many security problems have been detected in Java systems in the past [18]. The main contribution of this paper is an architecture for a secure Java operating system that avoids these problems and a discussion of its implementation and performance.

We follow Rushby [51] in his reasoning that a secure system should be structured as if it were a distributed system. With such an architecture a security problem in one part of the system does not automatically lead to a collapse of the whole system's security. Microkernels are well suited as the foundation of such a system. Especially systems that adhere to the multi-server approach, such as SawMill [28], and mediate communication between the servers [33] are able to limit the effect of security violations.

The JX system combines the advantages of a multi-server structure with the advantages of type-safety. It uses type-safety to provide an efficient communication mechanism that completely isolates the servers with respect to data access and resource usage.

The paper is structured as follows. Section 2 gives an overview about Java security and analyzes some weaknesses of the Java security mechanism. Section 3 describes the architecture of JX with the focus on the security architecture. Section 4 describes the performance of the system as a web server. Section 4 discusses how the system meets the requirements of a security architecture. Section 5 describes related work and Section 6 concludes the paper.

2 Java Security

Java security is based on the concept of a sandbox, which relies on the type-safety of the executed code. Untrusted but verified code can run in the sandbox and can not leave the sandbox to do any harm. Every sandbox must have a kind of exit or hole, otherwise the code running in the sandbox can not communicate results or interact with the environment in a suitable way. These holes must be clearly defined and thoroughly controlled. The holes of the Java sandbox are the native methods. To control these holes, the Java runtime first controls which classes are allowed to load a library that contains native code. These classes must be trusted to guard access to their native methods. The native methods of these classes should be non-public and the public non-native methods are expected to invoke the SecurityManager before invoking a native method. The SecurityManager inspects the runtime call stack and checks whether the caller of the trusted method is trusted.

Java version 1 distinguishes between trusted system classes, which were loaded using the JVM internal class loading mechanism, and untrusted classes, which were loaded using a class loader external to the JVM. Implementations of the SecurityManager can check whether the classes on the call stack—the callers of the method—are trusted or untrusted classes. When the caller was a system class the operation usually is allowed otherwise the SecurityManager decides, depending on the kind of operation and its parameters, whether the untrusted class is allowed to invoke the operation[1].

Java version 2 also relies on stack inspection but can define more flexible security policies by describing the permissions of classes of a certain origin in external files.

To sum up, Java security relies on the following requirements:

(1) Code is kept in a sandbox by using an intermediate instruction set. Programs are verified to be type-safe.

1. The real implementation uses the abstraction of *classloader-depth*, which is the number of stack frames between the current stack frame and the first stack frame connected to a class that was loaded using a classloader.

(2) The package-specific and/or class-specific access modifiers must be used to restrict access to the holes of this sandbox: the native methods of trusted classes. As long as the demarcation line between Java code and native code is not crossed, the Java code can do no harm.

(3) The publicly accessible methods of the trusted classes must invoke the SecurityManager to check whether an operation that would leave the sandbox is allowed.

The SecurityManager is similar to a reference monitor, but has a severe shortcoming: it is not automatically invoked. A trusted class must explicitly invoke the SecurityManager to protect itself. The mere number of native methods makes it difficult to assure this. We counted 1312 native methods in Sun's JRE 1.3.1_02 for Linux, which are 2.9 percent of all methods. From these native methods 34 percent are public and even as much as 16 percent are public static methods in a public class. This means that the method can be invoked directly from everywhere without the SecurityManager having a chance to intercept the call. Two of these methods are java.lang.System.currentTimeMillis() and java.lang.Thread.sleep() which provides an interesting opportunity to create a covert timing channel. The fact that covert channels are not exploited can be attributed to the existence of many overt channels. Public, non-final, static variables in public system classes are only one example (we counted 31 of these fields in Sun's JRE).

A further problem is that the stack inspection mechanism only is concerned with access control. It completely ignores the availability aspect of security. This lack was addressed in JRes [17]. By rewriting bytecodes, JRes creates a layer on top of the JVM. In our opinion, this is the wrong layer for resource control, because resources that are only visible inside the JVM can only be accounted inside the JVM. Examples are CPU time and memory used for the garbage collector (GC) or just-in-time compiler or memory used for stack frames. Furthermore, rewriting bytecodes is a performance overhead in itself and it creates slower programs. Often, Java is perceived as inherently insecure due to the complexity of its class libraries and runtime system [22]. As will be described in Section 3, JX avoids this problem by not trusting the JDK class library.

3 JX Security Architecture

This section describes the aspects of the JX architecture that are relevant to security.

3.1 JX architecture

JX is a single address space system. All code runs in one physical address space; an MMU is not used. Protection is based on the type-safety of the Java bytecode instruction set. A small microkernel contains low-level hardware initialization code and a minimal Java Virtual Machine (JVM).

The JX system is structured into domains (see Figure 1). Each domain represents the illusion of an independent JVM. A domain has a unique ID, its own heap including its own garbage collector, and its own threads. Thus domains are isolated with respect to CPU and memory consumption. They can be terminated independently from each other and the memory that is reserved for the heap, the stack and domain control structures can be released immediately when the domain is terminated.

All domains execute 100% Java code. The microkernel represents itself also as a domain. Because this domain has the ID 0 it is called DomainZero. DomainZero contains all C and assembler code that is used in the system.

JX does not support native methods and there is no trusted Java code that must be loaded into a domain. There is no trust boundary within a domain which eases administration and allows a domain complete freedom in what code it runs. Because the domain contains no trusted code it is a sandbox that is completely closed. We create a new hole by introducing capabilities, called *portals*.

Portals are proxies [55] for a service that runs in another domain. Portals look like ordinary objects and are located on a domains heap, but the invocation of a method synchronously transfers control to the service that runs in another domain. Parameters are copied from the client to the server domain.

Portals and services can not be created explicitly by the programmer. They "magically" appear during portal communication. When a domain wants to provide a service it can define a portal interface, which must be a subinterface of jx.zero.Portal, and a class that implements this interface. When an instance of such a class is passed to another domain the portal invocation mechanism creates a service in the source domain and a portal in the destination domain. This architecture has a bootstrap problem: A domain can

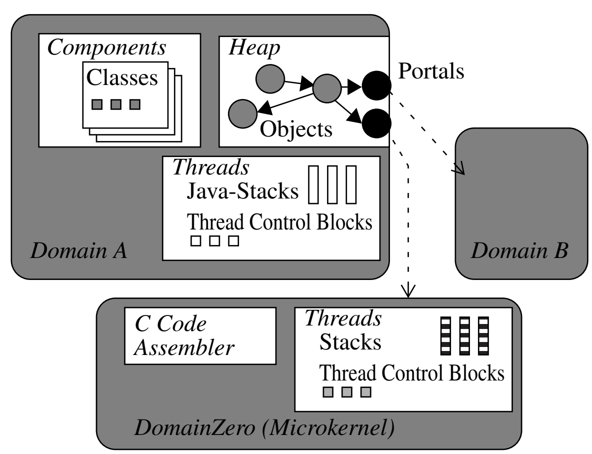

Figure 1: Structure of the JX system

obtain new portals solely by using existing portals. Therefore each domain possesses an initial portal: a portal to a naming service. Using this portal the domain can obtain other portals to access more services. When a domain is created, the creating domain can pass the naming portal as a parameter of the domain creation call. When no naming portal is specified in the createDomain[2] call, the default Naming portal of the creating domain is passed to the created domain. The naming service of the microkernel is used only by the initial domain (DomainInit) which implements a naming service in Java and passes this naming service to all domains it creates. Because DomainInit looks up all portals from the microkernel on startup no interaction with the microkernel naming service by any domain is needed after DomainInit has completed its initialization.

The implementation of the portal mechanism had to fulfil the following requirements:

- It must not be possible to explicitly create a portal object.
- It must be possible to terminate a domain and release all its resources independent of its current communication relationships.
- As services are created by the microkernel they must also be automatically removed when they are no longer needed. The data structures necessary to control a service must be placed on the domains heap and a garbage collector must be able to move them.

With the following implementation all these requirements are met. A service is represented by a service control block (SCB) that is stored on the server domain's heap. The SCB has a reference to the object that contains the implementation of the portal methods, a thread that is used to execute the methods, and a queue of waiting senders (Figure 2).

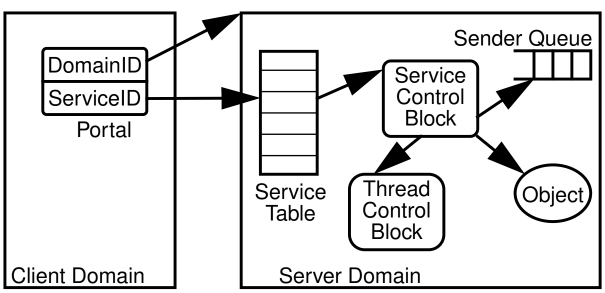

Figure 2: Portal data structures

A portal contains no direct pointer to the Service Control Block (SCB) because the SCB is stored on the heap and can be moved by the garbage collector. Using direct pointers would require updating all portals to a service during a GC cycle of the service domain. This would require a scan of the heaps of all domains which does not scale well. Therefore a portal contains the index of the service in a domain-local service table, a pointer to the Domain Control Block (DCB) and a domain ID. DCBs are one of the few global data structures of JX. Because the DCB of a domain is reused when a domain terminates and portals can outlive the domain in which the service is located, the DCB pointer could point to a DCB that contains not information about the terminated service domain but a newly created domain. Therefore the portal contains also a unique domain ID, which is checked against the ID in the DCB before the DCB is used.

Although the portal is located on the heap of the client domain the Java code has no way to access its contents. The type of the portal reference is the jx.zero.Portal interface, which, as an interface, has no fields. Thus it is not possible to forge a portal to access an arbitrary service.

Services are removed automatically when no portal to the service exists. To detect this condition the SCB contains a reference counter that counts the number of portals to the service. When a portal is passed to another domain a portal to the same service is created in the other domain and the reference counter is incremented. When a portal is garbage collected the finalization cycle decrements the reference counter of the service. When a domain terminates all portals can be considered garbage and a finalization cycle is performed before the heap memory is released.

3.2 JX as a capability system

Portals are capabilities [19]. A domain can only access other domains when it possesses a portal to a service of the other domain. The operations that can be performed with the portal are listed in the portal interface.

Although the capability concept is very flexible and solves many security problems, such as the confused deputy [30], in a very natural way, it has well known limitations. The major concern is that a capability can be used to obtain other capabilities, which makes it difficult, if not impossible, to enforce confinement [62]. JX as described up to now can not enforce confinement. Thus an additional mechanism is needed: a reference monitor that is able to check all portal invocations and the transfer of portals between domains.

3.3 The reference monitor

A reference monitor must be tamper-proof, mediate all accesses, and be small enough to be verified.

A reference monitor for JX must at least control incoming and outgoing portal calls. There are two alternatives for the implementation of such a reference monitor:

Proxy. Initially a domain has access only to the naming portal that is passed during domain creation. To obtain other portals the name service is used. The parent domain can

2. createDomain is a method of the DomainManager service which runs in DomainZero.

implement this name service to not return the registered portal but a proxy portal which implements the same interface. This proxy can then invoke a central reference monitor before invoking the original portal.

Microkernel. The portal invocation mechanism inside the microkernel invokes a reference monitor on each portal call and passes sender principal, receiver principal, and call parameters to the reference monitor.

These two implementation alternatives have the following advantages and drawbacks. The proxy solution needs no modification of the microkernel and thus avoids the danger of introducing new bugs. As long as no reference monitoring is needed, the proxy solution does not cause any additional cost. The microkernel solution must check in every portal invocation sequence whether a reference monitor is attached to the domain. Because the domain control block, which contains this information, is already in the cache during the portal invocation, this check is nearly for free. On the other hand, the proxy solution requires the name service to create a proxy for each registered portal. During a method invocation at such a portal the whole parameter graph must be traversed and when a portal is found it must be replaced by a proxy portal.

We rejected the proxy approach, because it requires a rather complex implementation and it is difficult to assure that each portal is "encapsulated" in a proxy portal.

We modified the microkernel to invoke the reference monitor when a portal call invokes a service of the monitored domain (inbound) and when a service of another domain is invoked via a portal (outbound). The internal activity of a domain is not controlled. The same reference monitor must control inbound and outbound calls of a domain, but different domains can use different monitors. A monitor is attached to a domain when the domain is created. When a domain creates a new domain, the reference monitor of the creating domain is asked to attach a reference monitor to the created domain. Usually, it will attach itself to the new domain but it can - depending on the security policy - attach another reference monitor or no reference monitor at all.

It must be guaranteed, that while the access check is performed, the state to be checked can only be modified by the reference monitor. When this state only includes the parameters of the call, these parameters could be copied to a location that is only accessible by the reference monitor. When the state includes other properties of the involved domains, the activity of these domains must be suspended. For these reasons the access check is performed in a separate domain, not in the caller or callee domain.

The list of parameters is accessed using an array of VMObject portals. VMObject is a portal which allows access to an object of another domain. The reference monitor furthermore gets the Domain portal of the caller domain and the callee domain. To accelerate the operation of the reference monitor, the Domain portal is a portal which can be inlined by the translator. On an x86 it takes only two machine instructions to get the domain ID given the Domain portal.

The main problem is to obtain a consistent view of the system during the check. One way is to freeze the whole system by disabling interrupts during the check. This would work only on a uniprocessor, would interfere with scheduling, and allow a denial-of-service attack. Therefore, our current implementation copies all parameters from the client domain to the server domain up to a certain per-call quota. These objects are not immediately available to the server domain, but are first checked by the security manager. When the security manager approves the call the normal portal invocation sequence proceeds.

3.4 Making an access decision

Spencer et al. [58] argue that basing an access decision only on the intercepted IPC between servers forces the security server to duplicate part of the object server's state or functionality. We found two examples of this problem. In UNIX-like systems access to files in a file system is checked when the file is opened. The security manager must analyze the file name to make the access decision, which is difficult without knowing details of the file system implementation and without information that is only accessible to the file system implementation. The problem is even more obvious in a database system that is accessed using SQL statements. To make an access decision the reference monitor must parse the SQL statement. This is inefficient and duplicates functionality of the database server.

There are three solutions for these problems:

(1) The reference monitor lets the server proceed and only checks the returned portal (the file portal).

(2) The server explicitly communicates with the security manager when an access decision is needed.

(3) Design a different interface that simplifies the access decision.

Approach (1) may be too late, especially in cases where the call modified the state of the server.

Approach (2) is the most flexible solution. It is used in Flask with the intention of separating security policy and enforcement mechanism [58]. The main problem of this solution is, that it pollutes the server implementation with calls to the security manager. The Flask security architecture was implemented in SELinux [40]. In SELinux, the list of permissions for file and directory objects have a nearly one-to-one correspondence to an interface one would use for these objects. This makes approach (3) the most promising approach. Our two example problems would be solved by parsing the path in the client domain. In an analogous manner the SQL parser is located in the client domain and a parsed representation is passed to the server domain and intercepted by the security manager. This has the additional advantage of moving code to an untrusted client, eliminating the need to verify this code. Section 3.11 gives further details about the design of the file server interface.

3.5 Controlling portal propagation

In [36] Lampson envisioned a system in which the client can determine all communication channels that are available to the server *before* talking to the server. We can do this by enumerating all portals that are owned by a domain. As we can not enforce a domain to be *memoryless* [36], we must also control the future communication behavior of a domain to guarantee the confinement of information passed to the domain.

Several alternative implementations can be used to enumerate the portals of a domain:

(1) A simple approach is to scan the complete heap of the domain for portal objects. Besides the expensive scanning operation, the security manager can not be sure, that the domain will not obtain portals in the future.

(2) An outbound intercepter can be installed to observe all outgoing communication of the domain. Thus a domain is allowed to posses a critical portal but the reference monitor can rejects it's use. The performance disadvantage is that the complete communication must be checked, even if the security policy allows unrestricted communication with a subset of all domains.

(3) The security manager checks all portals transferred to a domain. This can be achieved by installing an inbound interceptor which inspects all data given to a domain and traverses the parameter object graph to find portals. This could be an expensive operation if a parameter object is the root of a large object graph. During copying of the parameters to the destination domain, the microkernel already traverses the whole object graph. Therefore it is easy to find portals during this copying operation. The kernel can then inform the security manager, that there is a portal passed to the domain (method createPortal()). The return value of createPortal() decides whether the portal can be created or not. The security manager must also be informed if the garbage collector destroys a portal (destroyPortal()). This way reference monitor can keep track of what portals a domain actually possesses.

Confinement can now be guaranteed with two mechanisms that can be used separately or in combination: (i) the control of portal communication and (ii) the control of portal propagation.

Figure 3 shows the complete reference monitor interface. Figure 4 shows the information that is available to the reference monitor.

```
public interface DomainBorder {
    boolean outBound(InterceptInfo info);
    boolean inBound(InterceptInfo info);
    boolean createPortal(PortalInfo info);
    void destroyPortal(PortalInfo info);
}
```

Figure 3: Reference monitor interface

```
public interface InterceptInfo extends Portal {
   Domain getSourceDomain();
   Domain getTargetDomain();
   VMMethod getMethod();
   VMObject getServiceObject();
   VMObject[] getParameters();
}

public interface PortalInfo extends Portal {
   Domain getTargetDomain();
   int getServiceID();
}
```

Figure 4: Information interfaces

3.6 Principals

A security policy uses the concept of a *principal* [19] to name the subject that is responsible for an operation. The principal concept is not known to the JX microkernel. It is an abstraction that is implemented by the security system outside the microkernel, while the microkernel only operates with domains. Mapping a domain ID to a principal is the responsibility of the security manager. We implemented a security manager which uses a hash table to map the domain ID to the principal object. We first considered an implementation where the microkernel supports the attachment of a principal object to a domain. The biggest problem of such a support would be the placement of the principal object. Should the object live in the domain it is attached to or in the security manager domain? Both approaches have severe problems. As the security manager must access the object it should be placed in the security manager's heap. But this creates domain interdependencies and the independence of heap management and garbage collection, which is an important property of the JX architecture, would be lost. Thus, a numerical principal ID seemed to be the only solution. But having a principal ID has no advantages over having a domain ID, so finally we concluded that the microkernel should not care about principals at all.

The security manager maps the unique domain ID to a principal object. Once the principal is known, the security manager can use several policies for the access decision, for example based on a simple identity or based on roles [24].

To service a portal call the server thread may itself invoke portals into other domains. To avoid several problems (trojan horse, confused deputy [30]) the server may want to downgrade the rights of these invocations to the rights of the original client. The most elegant solution of these problems is a pure capability architecture. In the JX architecture this would mean that the server uses only portals that were obtained from that particular client. This requirement is difficult to assure in a multi-threaded server domain that processes requests from different clients at the same time. Because the server threads use the same heap, a portal may leak from one server thread to another. A better solution is to allow the reference monitor to downgrade the rights of a call. To allow the reference monitor to enforce downgrading rights to the rights of the invoker, each service thread (a thread that processes a portal call) has the domain ID of the original client attached to it. This information is passed during each portal invocation. The reference monitor has access to this information and can base the access decision on the principal of the original domain, instead of the principal of the immediate client.

3.7 Revocation of memory objects

There is a special kind of portals, called *fast portals*. Fast portals can only be created by DomainZero. They are executed in the context of the caller. The semantics of a fast portal is known to the system and it's methods can be inlined by the translator. An example for a fast portal is the Memory portal. We solved the confinement problems of capabilities by introducing a reference monitor that is invoked when a portal is used. This is not practical with memory portals for performance reasons, although it could be done. Therefore memory portals support revocation. When the reference monitor detects that a portal is passed between two domains (createPortal()) it could revoke the access right to the memory object for the source domain or reject passing of the memory portal.

3.8 Minimizing the JDK class library

The JVM and the class library of the Java Development Kit (JDK) can not easily be separated from each other.

In JX the JDK is not part of the trusted computing base (TCB). However, there are some classes, whose definition is very tightly integrated with the JVM specification [38][29]. Although these classes (except Object) are implemented outside the runtime system, the runtime system must know about their existence or even know part of their internal structure (fields and methods). These structural requirements are checked by the verifier.

The class Object is the base class of all classes and interfaces. It contains methods to use the object as a condition variable, etc. In JX Object is implemented by the runtime system. The class String is used for strings. Because String is used inside the runtime system, it is required that the String class does exist in a domain and that the first field is a character array. The runtime system needs to throw several exceptions, such as ArrayIndexOutOfBoundsException, NullpointerException, OutOfMemoryError, StackOverflowError. It is required that these classes and their superclasses RuntimeException, Exception and Throwable exist in a domain. There are no structural requirements for these classes. Arrays are type compatible to the interfaces Cloneable and Serializable. These interfaces also must exist in a domain.

Classes are represented by the portal jx.zero.VMClass. But because Object contains a method getClass(), it is required that java.lang.Class exists and contains a constructor which has one parameter of type VMClass.

3.9 Structure of the Trusted Computing Base

Figure 5 shows the structure of the trusted computing base (TCB). In the TCB we include all system components that the user trusts to perform a certain operation correctly. The central part of the system is the *integrity kernel*. Compromising the integrity kernel allows an intruder to crash the whole system. Built on the integrity kernel is the *security kernel*. The security kernel represents the minimal TCB. In a typical system configuration the TCB will include the window manager and the file system. Users will trust the file system to store their data reliably. Compromising the security kernel or the rest of the TCB leads to security breaches, such as disclosure of confidential data or unauthorized modification of data, but not to an immediate system crash. It may lead to a system crash when a compromised security kernel allows access to the integrity kernel. This design is reminiscent of the protection rings of Multics.

JX is a component-based system. A component consists of a number of classes and a file that describes the component. This file also contains the information on what other components the component depends on. The modularization and explicit dependencies allows to remove unnecessary functionality with a few configuration changes. For example in a server system the window manager may not be part of the TCB, while in a thin client system the file system may not be needed. A user may even decide not to trust the file system and store the data in an own data base.

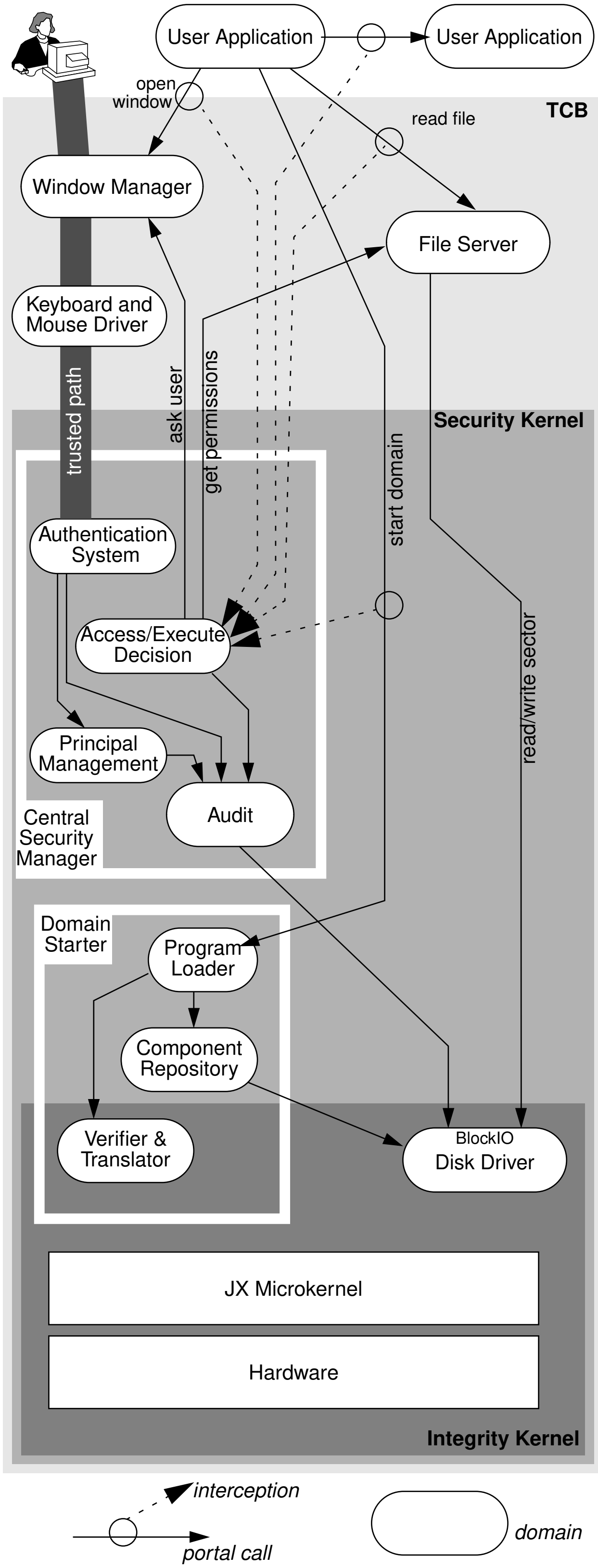

Figure 5: Typical TCB structure

It is important that there are no dependencies between the inner kernels and the outer ones. The security manager, for example, must not store its configuration in the file system but use its own simple file system.

Tamper-resistant auditing. The system must assure that all security relevant events are persistently stored on disk and cannot be modified. To be certain that the audit trail is tamper-proof we use a separate disk and write this disk in an append only mode. We do not use a file system at all but write the messages unbuffered to consecutive disk sectors. We do not use any buffering and the audit call only returns when the block was written to disk. Writing at consecutive disk sectors avoids long distance head movements and gives a rate of 630 audit messages per second[3]. Writing one audit message needs 1582 µseconds. Given that a file access which can be satisfied from the buffer cache is in the tenth of µseconds auditing each file access adds considerable overhead. The size of a typical audit message is between 35 and 40 bytes. The disk is used as a ring buffer: when the last sector is reached we wrap to the first one and overwrite old logs. This avoids a problem often encountered when logging to a file system: when the file system is full logs get lost. Usually, the most recent logs are the most valuable. With the above mentioned message rate of 630 messages/second and a message size of 40 bytes we have a time window of 110 hours using a 10 GBytes disk. Under normal operation the time window is much larger, because the message rate is well below its maximum.

Trusted path. According to the Orange Book [21] a trusted path is the path from the user to the TCB. Depending on the user interface the TCB must include the window manager or the console driver.

Recent literature generalizes the notion of a trusted path to any communication mechanism within the system. To trust a communication path it is essential to identify the communication partner and provide a communication channel that can not be overheard or modified. Portal communication is such a mechanism.

Usually, the reference monitor limits communication according to a certain security policy. This mechanism works automatically and is transparent to domains. But it is even possible for a domain to explicitly consider portal communication as being performed on a trusted path, because the target domain of a portal can be obtained and this identity can not be spoofed.

3. The following hardware was used for all measurements in this paper: Intel PIII 500 MHz, 512 KB cache, 640 MB RAM, 440BX Chipset, 82371AB PIIX4 IDE, Maxtor 91303D6 disk.

3.10 Maintaining security in a dynamic system

An operating system is a highly dynamic system. New users log in, services are started and terminate, rights of users are changing, etc. To maintain security in such a system, the initial system state must be secure and each state transition must transfer the system into a secure state.

There are two issues to be considered here: the system issue and the security policy issue.

It must be guaranteed that trusted software is not tampered and untrusted software runs in a restricted environment. The system starts with a secure boot process. Provided that no attacker has physical access to the hardware booting from a tamper-proof device, such as a CD-ROM, is sufficient and we do not need a secure boot process as in AEGIS [2] that checks for hardware modifications. We trust the initial domain to correctly start the security services and to attach them to the created domains. Each domain is started with a strictly defined set of rights (portals) and no trusted code. The initial portals always include a naming portal with which other portals can be obtained. To avoid the expensive nameserver lookup it is possible to pass a set of additional portals to a newly created domain. The created domain is automatically associated with a principal. When a domain obtains new portals or communicates using existing portals the security system is involved.

The policy issue is concerned with secure changes of the access rights, additions of principals, etc. How this is done depends on the used security policy and is outside the scope of this paper.

3.11 Securing servers

We use the file system server to illustrate how our security architecture works in a real system. As we discussed in Section 3.4 we use the server interface to make access decisions. For this to work servers must export *securable interfaces*. A securable interface must use simple parameters and provide fine-grained simple operations.

Many servers have a built-in notion of permissions, for example the user/group/other permissions in a UNIX file system. We call them *native permissions*. These permissions can be supplemented or replaced by a set of *foreign permissions*. These permissions could, for example, be access control lists. Because foreign permissions are not supported by the server, there must be a way to store them. The SELinux system [40] uses a file hierarchy in the normal file system to store foreign permissions.

There is some scepticism whether a capability-based system can be compatible to the JDK (see the discussion of capabilities in [63]). We proved that this is possible by implementing a component that implements the java.io.* classes in terms of our capability-based filesystem interface (Figure 6).

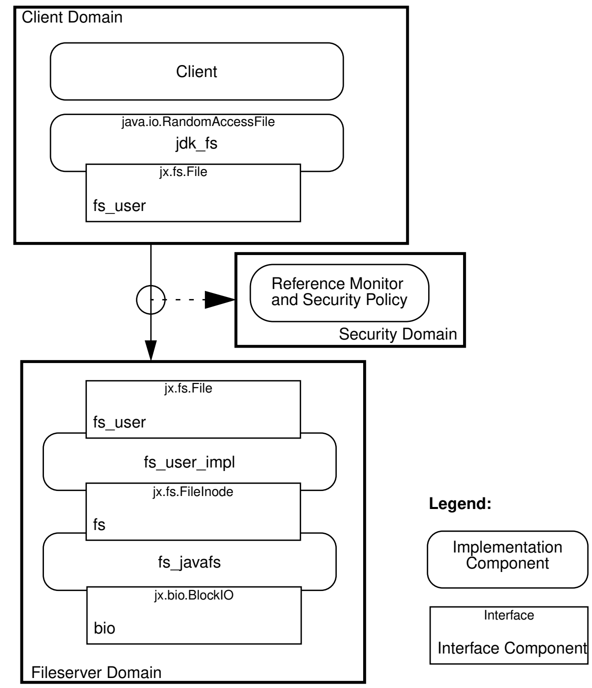

Figure 6: Filesystem layers

The implementation component jdk_fs contains implementations for the java.io.* classes and uses portal interfaces from the fs_user interface component to access the file system. These portals access service objects that are implemented in the fs_user_impl component.

Code that uses the java.io classes can run unmodified on top of our implementation of java.io. But the advantages of a capability-based system are lost: files must be referenced by name and problems similar to the Confused Deputy [30] are possible. An application can avoid the problems by using the (not JDK-compatible) capability-based file system interface.

In an multi-level security (MLS) system in which the file system is part of the TCB, the file system must be verified to work correctly - which may be a difficult task as file systems employ non-trivial algorithms. We used a configuration which eliminates the need for file system verification. Our system creates different instances of the file system for the different security levels, each file server being able to use a disjunct range of sectors of the disk. Assuring correct MLS operation can now be reduced to the problem of verifying that the disk driver works correctly; that is, it really writes the block to the correct position on the disk. The file system may run outside the TCB with a security label that is equivalent to the data it stores.

4 Discussion

In this section we analyze how well JX meets the Saltzer & Schroeder [52] requirements for a security architecture:

Economy of mechanism. *The security mechanisms must be simple and small to be subject to verification*. The microkernel is small and as simple as possible. The concept of stack inspection is no longer needed. Even untrusted code can obtain a capability to do useful work in a restricted way. JX relies on the type safety of the Java Bytecode language. If a flaw in the type system is found the whole system is compromised. We assume that Java is type-safe. There is a lot of ongoing and finished work on formally proving the type safety of Java. Using a simpler intermediate language could make this proof easier and require a simpler translator [45].

The trusted computing base must be as small as possible, because it must be verified to obtain high assurance. It must not only be small in size, but the whole system architecture must be simple and clean. One requirement for a security architecture is a small and modular TCB. Table 1 gives an estimate of the complexity of several systems by counting lines of code (kLOC = 1000 lines of code). When comparing the numbers one should keep in mind that different programming languages are used: the Translator and Verifier of JX are written in Java, the kernel of JX and all other systems are written in C and assembler. A number of programming errors that are possible in C and assembler are not possible in Java, such as memory management and pointer manipulation errors.Therefore we assume that Java programs contain less bugs per LOC.

System	Parts	kLOC	total kLOC
SecureJava [22]	Paramecium Kernel Java Nucleus	11 22	33
Linux 2.4.2	kernel mm ipc arch/i386/kernel arch/i386/mm fs fs/ext2 net/ipv4	13 15 3 25 1 23 5 34	119
Linux 2.4.2 drivers			1,711
Linux 1.0 & drivers			105
LOCK [57]	TCB	87	87
KeyKOS [49]	Kernel Domain code	25 25	50
JX integrity kernel (no drivers)	Microkernel Translator Verifier	25 40 12	77

Table 1: Operating system code sizes (from published sources or measured using wc and find)

All systems have between 30 and 120 kLOC. The largest part of the Linux source code are device drivers. But only few drivers are normally linked to the kernel statically or as a module. The Linux number only contains the absolutely necessary part of the sources. The number would be higher in one of the standard distributions where the kernel contains additional file systems, network protocols, or other services.

Using a Java processor the translator can be eliminated from the TCB. This would reduce the size of the integrity kernel to 37 kLOC.

Fail-safe defaults. *Access should be rejected if not explicitly granted*. Basing access decisions on fail-safe defaults is mainly the responsibility of the security manager. As an example, we implemented a security manager that allows communication between dedicated principals and automatically rejects all other communication attempts.

Complete mediation. *All accesses must be checked*. The reference monitor is automatically invoked when a portal is accessed.

Open design. *The system design must be published*. The design and implementation of JX is completely open

Separation of privilege. *Do not concentrate all privileges at one principal*. The microkernel-based architecture supports a system design where privileges are not centralized in one component but distributed through the system in separate domains. Domains do not trust each other; therefore breaking into one domain has a strictly limited effect for the overall system security.

Least privilege. *A system component should be granted the minimal privileges necessary to fulfil its task*. A domain starts with the privilege to lookup portals. What portals can be obtained by a domain is limited by the name service and also by the reference monitor that is consulted when a portal comes into a domain or is passed to another domain. If the file system is compromised file data can be modified and disclosed, but a database or file system that run in another domain can still be trusted - as long as it does not trust the compromised file system domain.

Least common mechanism. *No unnecessary sharing between system components should be allowed*. JX allows controlled sharing between applications (domains) using portals. Domains do not share resources that are implemented by the microkernel. All resources, like files, sockets, and database tables, are implemented by domains and shared using portals. Domains have separate heaps and independent garbage collection.

Psychological acceptability. *When the security system communicates with the human user it must respect the mental model of the user and must not annoy the user with too many questions*. Whether the user accepts a security policy depends on the formulation and implementation of the policy and on the user interface. This is outside the scope of this paper.

Besides the requirements described by Saltzer & Schroeder there are additional requirements:

Separation of policy and enforcement. Separation of policy from mechanism is a software engineering principle that leads to a modular system structure with evolvable and exchangeable policies. Several security architectures follow this principle. The DTOS system [43] and its successor Flask [58] concentrated on policy flexibility in a microkernel-based OS. In some systems, security decisions are spread over the whole system, which makes it difficult to understand what kind of security policy the system as a whole actually enforces [37]. Centralizing the policy facilitates adaptations to new security requirements and enhances manageability. The policy can be changed without changes to the fundamental security and system architecture and without changes to the object servers. Furthermore, a central security manager is a requirement for the enforcement of complex security policies that are more than access decisions. The policy could, for example, state that all email must be encrypted, an alert must be activated for three unsuccessful login attempts, or that users of a certain clearance must store all their data in encrypted form. These policies can only be enforced by a security manager that has complete control over the system.

The security policy is not part of the servers. Even the enforcement is separated from the functional part of the servers.

Suitable programming language. The importance of the programming language for a secure system was recognized in early systems, such as KSOS [25]. A study of the Secure Computing Corporation evaluates five microkernel-based operating systems with respect to security [53]. This study contains a list ([53] pp. 24) of properties of a programming language that affect assurability of code. To improve assurability a programming language should allow a high abstraction level, support strong data typing, modularization, and prohibit pointer manipulation.

Many security flaws are due to language defiances, like buffer overflows, that simply cannot happen in a language like Java. We tried to keep the non-Java portion of the system as small as possible. As can be seen in Table 1 the microkernel, which is the only part of the system that is written in an unsafe language, is rather small (25 kLOC) compared to the other parts of the TCB.

Performance. Although current advances in processor performance will keep a moderate performance degradation below the perceptional threshold of a typical user, a slowdown of central mechanisms may have a dramatic effect for the performance of the whole system.

Security has an associated cost in terms of performance and resource usage. The performance overhead of JX has to causes: the use of a type-safe language and the use of a reference monitor. To measure the effect of type-safety we used two benchmarks and compare the JX performance with an equivalent Linux implementation: a web server and a Java implementation of the IOZone [65] benchmark.

As our JDK class library misses functionality that is required to run an off-the-shelf Java web server, such as tomcat [60], we wrote a simple Java web server. The JX web server accepts a connection, creates a either a new domain or a new thread and passes the portal that represents the TCP connection and a portal to the file system to the new domain/thread. Table 2 shows the performance of the JX web server. No reference monitor was installed in the system.

Benchmark	http request rate (req/sec)
JX web server using threads	459
JX web server using domains	142

Table 2: JX performance (mean of four runs each sending 1000 requests)

To see how well the JX web server performs we wrote an equivalent web server in C and measured its performance on Linux (Table 3). The Linux web server accepts a connection, and either forks a process that parses the http request, reads the requested file and sends a reply, or processes the request without forking a new process.

Benchmark	http request rate (req/sec)
Linux web server using fork	381
Linux webserver without fork	445

Table 3: Linux performance (mean of four runs each sending 1000 requests)

The Linux and JX/thread numbers are not much different. This indicates that even a TCP/IP stack that is written in Java can saturate a 100MBit/s network interface using a 500 MHz PIII processor. Creating a new domain to processes each request is considerably more expensive, but allows to execute arbitrary untrusted code to process the request.

A benchmark that is more dominated by computation performance is IOZone. Figure 7 compares the IOZone performance of JX to Linux. In this benchmark JX performs considerable worse than Linux. We expect this problem to gradually disappear in future versions, because there are no optimization barriers in the JX architecture: components can be loaded privately in a domain and the translator has global information about all components loaded in a domain; no stack inspection is used, i.e. methods can be

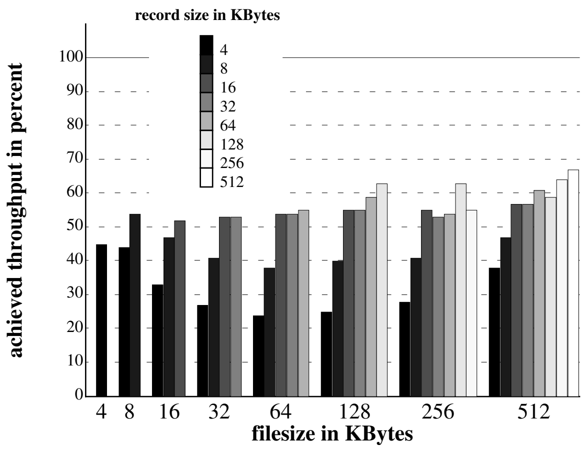

Figure 7: Multi-domain IOZone benchmark without reference monitor compared to Linux IOZone

inlined; a domain can use it's own non-preemptive scheduler and can be restricted to one processor, i.e. memory accesses can be liberally reordered, as long as they only affect the own heap of a domain.

Depending on the configuration, using a reference monitor causes an additional overhead. Figure 8 shows the overhead relative to the multi-domain configuration that is created by using a monitor that intercepts and allows all invocations.

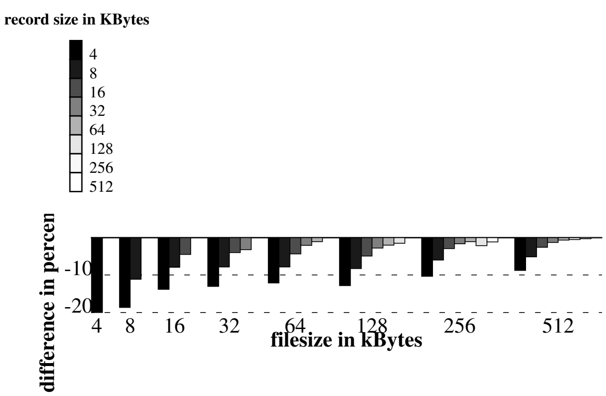

Figure 8: Client and file server in different domains; Monitor intercepts and allows all operations

We implemented a reference monitor which imitates the discretionary access policy of UNIX. Each domain is owned by a principal. The credentials of a principal consist of user ID and a list of group IDs. Each read and write access to the file portal is validated against the user credentials. During this check the security manager asks the file server for the file permissions. As can be seen in Figure 9 this configuration is expensive. Using a pure capability architecture is much faster, because only portal creation must be checked but not portal access. This creates, however, the problem of cached access decisions. When the security policy or the

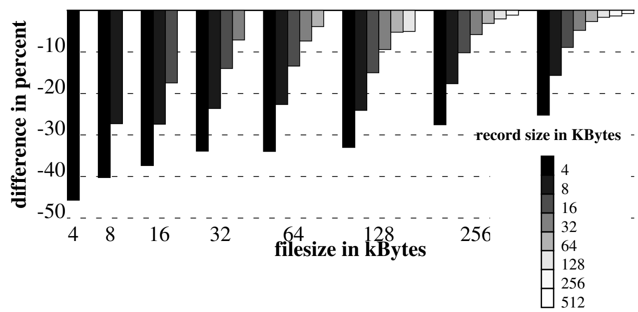

Figure 9: Client and file server in different domains; Monitor ch permissions before every read/write method

security attributes of an object are changed, the portal (capability) still allows access.

5 Related Work

Capability-based systems. Several operating systems are based on capabilities and use three different implementation techniques: partitioned memory, tagged memory [23], and password capabilities. Early capability systems used a tagged memory architecture (Burroughs 5000 [46], Symbolics Lisp Machine [44]), or partitioned memory in data-containing and capability-containing segments (KeyKOS [27] and EROS [54]). All these implementations relied on specific hardware features. To become hardware-independent, password capabilities [1] have been invented and are used in several systems (Mungi [32], Opal [14], Amoeba [59]). There is no definite agreement on how secure password capabilities really are. There is a non-zero chance that passwords can be guessed. Security relies on the strength of the cryptographic function. Password capabilities can be transferred over an external channel, for example, on a desktop system the user reads the capability in one window and types it in another window. Furthermore, using cryptographic methods adds an overhead when using password capabilities.

Type-safe instruction sets, such as the Java intermediate bytecode, are a fourth way of implementing capabilities. The main advantages of this technique are that it is hardware-independent, capability verification is performed at load time, access rights are encoded in the capability type and not stored as a bitmap in the capability, and capabilities can not be transferred over uncontrolled channels.

Virtual Machines. Virtual machines can be used to isolate systems that share the same hardware. The classic architecture is the IBM OS/360 [41]. Virtual machines experienced a recent revival with the VMWare PC emulator [64]. Using a VM to isolate untrusted systems requires that the underlying system (e.g., the control program in OS/360 and the host operating system of VMWare) is either secure or can not be attacked, because it is not connected to the network. Otherwise an intruder can break into the host system and read or modify the memory of the emulated system using interfaces, like /dev/kmem. VMs only work at a large granularity. VMWare instances consume a lot of resources to emulate a complete PC which makes it impossible to create fine-grained domains. Most applications require controlled information flow between classification levels; that is between VMWare instances. A virtual machine realizes a sandbox. The holes of the VMWare sandbox are the emulated devices. Thus, communication is rather expensive and stating a security policy in terms of an emulated device may be a difficult task.

Java security. Secure Java [22] aimed at reducing the TCB of a Java VM to its minimum. The bytecode verifier and just-in-time compiler are outside the TCB. The JIT can be inside the TCB to enable certain optimizations. The garbage collector is inside the TCB, but because the JIT and verifier are not trusted the integrity of the heap can not be guaranteed. We think that this is the main problem, because not relying on the integrity of the heap complicates the GC implementation and complex implementations should be avoided in a secure system. Heap integrity is important when reasoning about security of higher level applications. However, ideas from the Secure Java architecture could be used to build an additional protection ring inside our integrity kernel.

The J-Kernel [31] implements a capability architecture for Java. It is layered on top of a JVM, with the problems of a very large TCB and limited means of resource control. It uses classloaders to separate types. The capability system is not orthogonal to application code which makes reuse in a different context (using a different security policy) difficult.

The MVM [16], and KaffeOS [3] are systems that isolate applications that run in the same JVM. The MVM is an extension of Sun's HotSpot JVM that allows running many Java applications in one JVM and give the applications the illusion of having a JVM of their own. There are no means for resource control and no fast communication mechanisms for applications inside one MVM. KaffeOS is an extension of the Kaffe JVM. KaffeOS uses a process abstraction that is similar to UNIX, with kernel-mode code and user-mode code, whereas JX is more structured like a multi-server microkernel system. There needs to be no trusted Java code in JX. Communication between processes in KaffeOS is done using a shared heap. Our goal was to avoid sharing between domains as much as possible and we, therefore, use RPC for inter-domain communication.

6 Conclusion

We described the security architecture of the Java operating system JX, which can be seen as a hybrid of language-based protection and operating system protection. It reconciles the integrity of a type-safe protection mechanism with the strong isolation and complete mediation of operating systems. JX avoids typical Java security problems, such as native methods, execution of code of different trustworthiness in the same thread, and a huge trusted class library.

JX provides a number of security mechanisms of different invasiveness. The capability mechanism is inherent in the architecture and guarantees a minimal level of security. On a per-domain basis this mechanism can be supplemented by a monitor that controls propagation of capabilities between domains and, if necessary, a reference monitor that mediates access to these capabilities.

The measured performance overhead of the reference monitor indicates that this mechanism should not be used if not needed. We believe that for most applications the pure capability system, with proper interface design (e.g., a read-only interface), supplemented by the capability propagation monitor will provide sufficient security and guarantee confinement at a low cost.

7 References

[1] M. Anderson, R. Pose, and C. S. Wallace. A password-capability system. In *The Computer Journal, 29*, pp. 1-8, 1986.

[2] W. Arbaugh, D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. In *Proc. of IEEE Symposium on Security and Privacy*, pp. 65-71, May 1997.

[3] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proc. of 4th Symposium on Operating Systems Design & Implementation*, Oct. 2000.

[4] CERT/CC. *VU#16532: BIND T_NXT record processing may cause buffer overflow* . Nov. 1999.

[5] CERT/CC. *VU#5648: Buffer Overflows in various email clients*. 1998.

[6] CERT/CC. *VU#970472: Network Time Protocol ([x]ntpd) daemon contains buffer overflow in ntp_control:ctl_getitem() function*. Apr. 2001.

[7] CERT/CC. *VU#745371: Multiple vendor telnet daemons vulnerable to buffer overflow via crafted protocol options*. July 2001.

[8] CERT/CC. *VU#28934: Sun Solaris sadmind buffer overflow in amsl_verify when requesting NETMGT_PROC_SERVICE*. Dec. 1999.

[9] CERT/CC. *VU#952336: Microsoft Index Server/Indexing Service used by IIS 4.0/ 5.0 contains unchecked buffer used when encoding double-byte characters*. June 2001.

[10] CERT/CC. *VU#29823: Format string input validation error in wu-ftpd site_exec() function*. June 2000.

[11] CERT/CC. *VU#789543: IIS decodes filenames superfluously after applying security checks*. May 2001.

[12] CERT/CC. *VU#17215: SGI systems may execute commands embedded in mail messages*. Apr. 1998.

[13] CERT/CC. *VU#945216: SSH CRC32 attack detection code contains remote integer overflow*. Feb. 2001.

[14] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single Address Space Operating System. In *ACM Trans. on Computer Systems, 12(4)*, pp. 271-307, Nov. 1994.

[15] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, Jan. 2000.

[16] G. Czajkowski and L. Daynes. Multitasking without Compromise: A Virtual Machine Evolution. In *Proc. of the OOPSLA*, pp. 125-138, Oct. 2001.

[17] G. Czajkowski and T. von Eicken. JRes: A Resource Accounting Interface for Java. In *Proc. of Conference on Object-Oriented Programming Systems, Languages, and Applications 98'*, pp. 21-35, ACM Press, 1998.

[18] D. Dean, E. W. Felten, D. S. Wallach, D. Balfanz, and P. J. Denning. Java security: Web browsers and beyond. In D. E. Denning (ed.) *Internet Beseiged: Countering Cyberspace Scofflaws*. pp. 241-269, ACM Press, 1998.

[19] J. B. Dennis and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. In *Communications of the ACM, 9(3)*, pp. 143-155, Mar. 1966.

[20] E. Denti, A. Omicini, and A. Ricci. tuProlog: A Light-weight Prolog for Internet Applications and Infrastructures. In Ramakrishnan, I.V. (ed.) *Practical Aspects of Declarative Languages*. In *3rd International Symposium (PADL 2001)*, Lecture Notes in Computer Science 1990, pp. 184-198, Springer-Verlag, 2001.

[21] Department of Defense. *Trusted computer system evaluation criteria (Orange Book)*. DOD 5200.28-STD, Dec. 1985.

[22] L. v. Doorn. A Secure Java Virtual Machine . In *Proc. of the 9th USENIX Security Symposium* , pp. 19-34, Aug. 2000.

[23] R. S. Fabry. Capability-based addressing . In *Communications of the ACM, 17(7)*, pp. 403-412 , July 1974.

[24] D. Ferraiolo and R. Kuhn. Role-based access controls. In *Proc. of the 15th National Computer Security Conference*, pp. 554-563, Oct. 1992.

[25] Ford Aerospace. *Secure Minicomputer Operating System (KSOS) Executive Summary: Phase I: Design of the Department of Defense Kernelized Secure Operating System*. Technical Report WDL-781, Palo Alto, CA, 1978.

[26] Franco Gasperoni and Gary Dismukes. Multilanguage Programming on the JVM: The Ada 95 Benefits.2002.

[27] B. Frantz. KeyKOS - a secure, high-performance environment for S/370. In *Proc. of SHARE 70*, pp. 465-471, Feb. 1988.

[28] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *Proc. of the 9th SIGOPS European Workshop*, Sep. 2000.

[29] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Aug. 1996.

[30] N. Hardy. The confused deputy. In *Operating Systems Review, 22(4)*, pp. 36-38, Oct. 1988.

[31] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. v. Eicken. Implementing Multiple Protection Domains in Java. In *Proc. of the USENIX Annual Technical Conference*, pp. 259-270, June 1998.

[32] G. Heiser, K. Elphinstone, S. Russel, and J. Vochteloo. Mungi: A Distributed Single Address-Space Operating System. In *17th Australiasion Computer Science Conference*, pp. 271-280, Jan. 1994.

[33] T. Jaeger, J. Tidswell, A. Gefflaut, Y. Park, J. Liedtke, and K. Elphinstone. Synchronous IPC over Transparent Monitors. In *9th SIGOPS European Workshop*, Sep. 2000.

[34] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*, June 2002.

[35] R. Johnson. *TCL and Java Integration*. Technical Report , Sun Microsystems Laboratory, Jan. 1998.

[36] B. W. Lampson. A Note on the Confinement Problem. In *Communications of the ACM, 16(10)*, pp. 613-615, Oct. 1973.

[37] C. E. Landwehr, C. L. Heitmeyer, and J. McLean. A Security Model for Military Message Systems. In *ACM Trans. on Computer Systems, 2(3)*, pp. 198-222, Aug. 1984.

[38] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Sep. 1996.

[39] P. A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *21st National Information Systems Security Conference*, pp. 303-314, Oct. 1998.

[40] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Usenix 2001 Freenix Track*, 2001.

[41] G. Mealy, B. Witt, and W. Clark. The Functional Structure of OS/360. In *IBM Systems Journal, 5(1)*, pp. 3-51, Jan. 1966.

[42] S. G. Miller. *SISC: A Complete Scheme Interpreter in Java*. Technical Report, Jan. 2002.

[43] S. E. Minear. Providing Policy Control Over Object Operations in a Mach Based System. In *Proc. of the 5th USENIX Security Symposium* , June 1995.

[44] D. A. Moon. Symbolics Architecture. In *IEEE Computer, 20(1)*, pp. 43-52, IEEE, Jan. 1987.

[45] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *Conference Record 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 85-97, 1998.

[46] E. I. Organick. *Computer System Organization: The B5700/B6700 Series*. Academic Press, Inc., New York, 1973.

[47] PERCobol, http://www.legacyj.com/.

[48] J. Potter, J. Noble, and R. Shelswell. Project Bruce: Translating from Eiffel to Java. In *TOOLS 97*, 7.

[49] S. Rajunas, N. Hardy, A. Bomberger, W. Frantz, and C. Landau. Security in KeyKOS. In *Proc. of the 1986 IEEE Symposium on Security and Privacy*, Apr. 1986.

[50] N. Rappin and S. Pedroni. *Jython Essentials*. OReilly.,2002.'

[51] J. Rushby. Design and Verification of Secure Systems. In *Proc. of the 8th Symposium on Operating System Principles*, pp. 12-21, 1981.

[52] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE , 63(9)*, pp. 1278-1308 , Sep. 1975.

[53] Secure Computing Corporation. *DTOS General System Security and Assurability Assessment Report*. 1997.

[54] J. S. Shapiro, J. M. Smith, and D. J. Farber. *EROS: a fast capability system*. In *Symposium on Operating Systems Principles*, pp. 170-185, 1999.

[55] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *ICDCS 1986*, pp. 198-204, 1986.

[56] SmalltalkJVM, http://www.smalltalkJVM.com/.

[57] R. E. Smith. *Cost Profile of a Highly Assured, Secure Operating System*. Sep. 1999.

[58] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proc. of the 8th USENIX Security Symposium* , Aug. 1999.

[59] A. Tanenbaum. Chapter 7. In *Distributed Operating Systems*. Prentice Hall, 1995.

[60] The Jakarta Project, http://jakarta.apache.org/tomcat/.

[61] K. Thompson. Reflections on trusting trust. In *Communications of the ACM, 27(8)*, pp. 761-763, Aug. 1984.

[62] W. E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proc. of the 7th DoD/NBS Computer Security Conference*, pp. 291-293, Sep. 1984.

[63] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *16th Symp. on Operating System Principles*, pp. 116-128, Apr. 1997.

[64] Webpage of VMWare, http://www.vmware.com/.

[65] Webpage of the IOZone filesystem benchmark, http://www.iozone.org/.