A Microkernel Virtual Machine: Building Security with Clear Interfaces

Xiaoqi Lu Scott F. Smith

Department of Computer Science
The Johns Hopkins University
{xiaoqilu, scott}@cs.jhu.edu

Abstract

In this paper we propose a novel microkernel-based virtual machine (μKVM), a new code-based security framework with a simple and declarative security architecture. The main design goals of the μKVM are to put a clear, inviolable programming interface between different codebases or security components, and to limit the size of the trusted codebase in the spirit of a microkernel. Security policies are enforced solely on the interface because all data must explicitly pass through the inviolable interface. The architecture of the μKVM effectively removes the need for expensive runtime stack inspection, and applies the principle of least privilege to both library and application code elegantly and efficiently. We have implemented a prototype of the proposed μKVM. A series of benchmarks show that the prototype preserves the original functionality of Java and compares favorably with the J2SDK performance-wise.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks; D.3.4 [*Programming Languages*]: Processors—Run-time environment; D.2.11 [*Software Engineering*]: Software Architecture—Languages

General Terms Design, Languages, Security

Keywords Language-based Security, Virtual Machine, Frameworks, Java, Kernel, Interface, Access Control

1. Introduction

One important means by which software can be secured is via *code-based* security mechanisms, whereby different code components can be given different access rights. The interest in code-based security blossomed with Java; in fact, dealing with dynamically downloaded code of limited trustworthiness was the primary initial goal of the language. The Java J2SDK [LY99, GJSB05] and the Microsoft CLR [FJ03, MG01] both define security architectures that limit access rights of different blocks of code [Gon99, Gon, DW02, Bro].

1.1 Dimensions of Code-Based Security

The J2SDK code-based access control mechanism is centered around the set of `Permissions` needed to access various resources such as network connections, files, clipboard, or user-defined resources, and a policy that maps different codebases to the `Permissions` they have. The J2SDK access control system is *monolithic*, all code-blocks are treated as equal. Thus, individual applications running as e.g. Applets are just different codebases in the JVM, and system libraries are also treated as just another codebase.

We believe it is important to forego a monolithic approach and to factor code-based security into three distinct dimensions of interaction: *inter-application security* between largely independent applications running in a single VM, *system service security* for system resource access, and *intra-application security* for applications with multiple security domains. We cover each of these in turn.

One reason why the monolithic approach is lacking is for the case of applets: different applets should not be interacting with each other at all, and so a more natural architecture is one that completely isolates different applications running in the same VM. Several good solutions for application isolation have been developed [JSR, App, HCC$^+$98, BG98, CD01, DBC$^+$00, DC05]. We term this dimension of code-based security *inter-application security*.

The primary use of the J2SDK access control mechanism in practice is to limit access by applications to system resources. There are two mutually supporting reasons for limiting access: in the case of downloaded code, the trustworthiness of the code is partial. Additionally, even if the code producers are not malicious, the code may contain errors which are security vulnerabilities. The Principle of Least Privilege [SS75] states that only the permissions needed in practice should be given out, and so limiting the permissions given to an application to only what it should need will make a system more secure. We term this dimension *system service security*, and it is the primary focus of this paper.

An additional dimension of code-based security arises by applying the Principle of Least Privilege within a single application: the different components of a large application may not all need the full set of privileges given to the application as a whole. For example, a component of an application that is doing nothing but logging only needs access to the log file, and no other system resources. If the component was given all the privileges the application has, it could mistakenly, or because of a virus, misuse some of those permissions. We term this dimension *intra-application security* since it is within a single application.

1.2 Securing system services in the J2SDK

The primary focus of this paper is securing application access to system services. First we review the J2SDK approach to this problem, and point out some of its weaknesses.

PLAS'06 June 10, 2006, Ottawa, Ontario, Canada.

Java system libraries (java.*, javax.*) have privileges to all system resources. Those system libraries have to be guarded against being lured into being a "confused deputy" [Har88], in which libraries become victims of malicious applications and misuse their privileges to do things on behalf of attackers. The runtime stack inspection algorithm [Gon, WF98] is designed to address this problem. Upon initiation, the stack inspection traverses back on the call stack to check if each stack frame has the required permissions, and fails if a frame is found that is from untrusted code. In cases where some trusted code such as a library class needs to perform operations on behalf of untrusted code, it calls `doPrivileged()` to temporarily raise the security level of the untrusted code; stack inspection will then stop its traversal when encountering this `doPrivileged()` frame. The Java security architecture is mainly encapsulated in the `SecurityManager` class, whose `checkPermission()` method triggers the stack inspection.

Perhaps the most serious shortcoming of the J2SDK security architecture is the lack of a clear runtime interface between different code components: objects in one codebase can freely refer to objects in other codebases, and new references can be dynamically passed at runtime, there can be no precise division in the object reference graph, and there can thus be backdoor channels by which system services can be accessed. Guaranteeing system security becomes a formidable mission for library programmers because they have to completely understand the boundary between libraries and applications by carefully tracking where sensitive references are passed. Here is an obvious example: leaking a reference of an opened file to an application would give the application access to the file even if it does not have such permission by itself (recall that file permission is only explicitly checked upon opening files in the J2SDK, and not upon reading or writing).

Not only is there no clear runtime interface, there is also no clear compile-time interface: security-related code snippets in the form of `checkPermission()` and `doPrivileged()` are scattered throughout library code. Such a blurred security interface hinders application developers because they would have hard time to get a handle on the accurate security architecture on which they will run their applications. For instance, a method call from an application might either fail or succeed depending on whether there is a library `doPrivileged()` frame on the call stack. Without knowing how and where the `doPrivileged()` is located, the application cannot make assumptions about how its computation will proceed.

The fact that security decisions in Java are made by traversing the call stack brings tremendous subtleties to compiler optimization because optimizations usually incur changes to the call stack, which might alter the security semantics of the whole system.

Another major shortcoming in the J2SDK is that the Principle of Least Privilege has not been applied to Java system libraries: each library in fact only needs access to the relevant resources it is operating on, *e.g.* `java.io` library does not need network access. The J2SDK currently gives all system library all privileges. If privileges were doled out only to the libraries that needed them, a large runtime cost would be introduced because the stack inspection could not collapse many consecutive system stack frames. Researchers have developed static, declarative approaches to the stack inspection via type systems [SS00, TH03], but not all checks can be performed statically due to the fundamentally dynamic nature of components.

1.3 A Microkernel Approach to Securing System Services

In this paper we propose the μ-Kernel Virtual Machine (μKVM), a new code-based security framework with a simple and declarative security architecture. The main design goals of the μKVM are to put a clear, inviolable programming interface between the trusted codebase and less trusted one, and to decrease the size of the core trusted codebase. The *kernel* in our VM is a small trusted system codebase; since it is small compared to the large J2SDK system libraries, we term the architecture a *microkernel VM* architecture. The kernel is the only component with system-wide privileges, which effectively removes the need for runtime stack inspection: system libraries are not part of the kernel and have no privileges, therefore no malicious code can trick them to misuse any privileges. Such a microkernel based architecture embodies the Principle of Least Privilege for its system libraries: they have only the privileges they need.

Kernel-application interactions and the system security policy enforcement are defined solely on the interfaces between the kernel and applications. *Connectors* and *services* are the two types of cross-domain interfaces defined in the μKVM, representing persistent heavyweight interactions and lightweight one-time invocations, respectively. The key to the interface design is to give enough expressiveness to allow full functionality, but to make sure there are no backdoors.

We test our ideas in a prototype implementation of the μKVM, built by modifying the Sun sources of the J2SDK to replace the J2SDK security architecture with our new microkernel-based architecture. This change in architecture is internal in the sense that existing Java applications can run in the μKVM with basically no change, a fact that further illustrates the power and generality of the approach. By running a series of benchmark suites, we show the original Java functionality is preserved, and performance benchmarks indicate that the μKVM compares favorably with the J2SDK. Indeed, the μKVM is faster on enforcing security policies. The prototype at this point implements enough of the interfaces, namely those for file, network, thread and core part of the GUI, to show feasibility of the approach.

1.4 Intra-Application and Inter-Application Security

The techniques used to secure system services can be used analogously within a single application: a single-codebase application can be refactored into multiple codebases, where each codebase has the minimal privileges it needs, and it interacts with the other parts of the application via connectors. We will show how the above ideas can also be directly applied to the intra-application case.

Inter-application security has been covered well by many projects such as [JSR, App, HCC+98, BG98, CD01, DBC+00, DC05] and so that is not our focus here. However, placing the security fence at the proper place with the proper conditions is the key to achieve security in a system. The connector/service interface draws a clear line between different security components, making it a suitable abstraction for securing system services, intra-application interaction, as well as inter-application interaction.

2. Design of the μKVM

In this section, we first give a high-level overview of the proposed microkernel virtual machine model, and the intuitions behind the design. Then, the overall design blueprint of the μKVM is introduced.

2.1 Architecture Overview

A microkernel virtual machine is analogous to a microkernel operating system [Lie95, GDFR90]: the size of the codebase with special privileges is held to a minimum. An operating system microkernel usually provides minimal OS-neutral abstractions, such as memory address spaces, threads, *etc*. Sophisticated servers can be built upon those abstractions and run in user space. A microkernel architecture brings benefits such as robustness and configurability. Additionally, it has strengthened security guarantees due to the fact that only small number of services need maximal privileges.

Features that must be in our virtual machine kernel are the minimal services for a language runtime, which include the core language execution model, bytecode execution, threads, garbage collection, and code loading. Additionally, any potentially sensitive operation needs to be in the kernel so its use can be controlled. Most of these operations are input/output operations such as opening files, network connections, window system operations, *etc*. In order to minimize the size of the kernel, it is not necessary to put the whole file I/O library in the kernel—only the most low-level system operations need to be in the kernel. Virtual machines such as the JVM and CLR are not microkernel VM's: all of the system library code is placed in the kernel in the sense that all the system libraries have full privileges.

The central component of our virtual machine is the *kernel*, as shown in Fig. 1. The kernel is a special component that is created when the virtual machine starts, and it stays resident in memory thereafter. It manages system resources and exposes uniform interfaces via which user applications running on the VM can interact with the kernel. The kernel runs in system mode as a privileged component, while user applications run outside the kernel in user mode. The system/user modes in the μKVM are in close analogy to the dual modes in operating systems. Specifically, execution in user mode cannot issue privileged instructions directly, but only indirectly through the kernel.

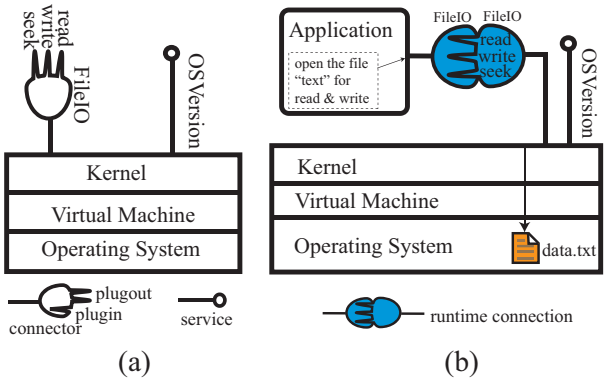


Figure 1. (a) Building blocks of the μKVM (b) Runtime connection on connectors

There are two types of interfaces in the μKVM, *connectors* and *services*. Connectors are designed to be used for long term interactions with objects such as files, and services are for simple querying. Connectors [RS02, LS04] are connection oriented in the sense that connections are required for communication on connectors. Services however are connection-less and therefore are one-time invocations. Connector interfaces are bidirectional: they import *plugin* and export *plugout* operations. Runtime connections have to be established before connector plugins/plugouts can be used. The kernel in Fig. 1(a) statically declares a connector "FileIO", which exports three plugouts, "read", "write" and "seek". Services are the interfaces for standard client/server style invocations. "OSVersion" is a service via which an application can query the version of the native operating system. Connectors do an excellent job of expressing persistent communication channels such as file operations and socket connections, but simple operation requests are not persistent and are more elegantly implemented as services; thus the μKVM has both connectors and services.

Connections on matching connectors are peer-to-peer persistent links constructed by mutual agreement between the two parties involved. Once a connection has been established, calls on a plugin are delegated to the corresponding plugout. For example, in order to read/write a file, the application in Fig. 1(b) first needs to request a connection with the kernel, which links a pair of matching "FileIO" connectors as shown in the figure. After this point, the application's call on its "read" plugin triggers the kernel's "read" plugout, *etc*. For each open file there is a different connection established, and the connection is maintained as long as the file is open. A connection can be disconnected by either party. We use plain connectors as pictured in Fig. 1(a) to symbolize static connector interfaces, and coupled connectors in dark shade to represent runtime connections as shown in Fig. 1(b). We follow this convention throughout this paper.

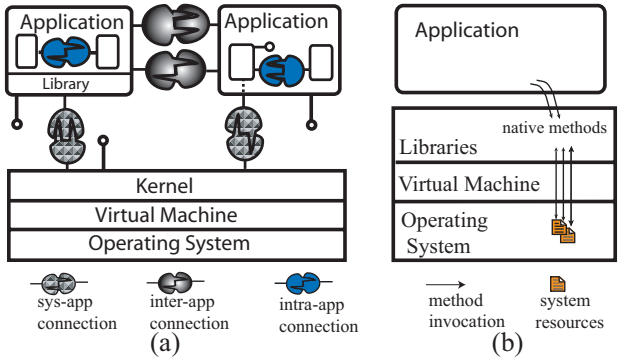


Figure 2. (a) The μKVM runtime overview and (b) J2SDK Runtime

Fig. 2(a) is a sketch of the μKVM runtime. The three different types of connections are shaded differently to emphasize their different uses: kernel-application, intra-application and inter-application connections.

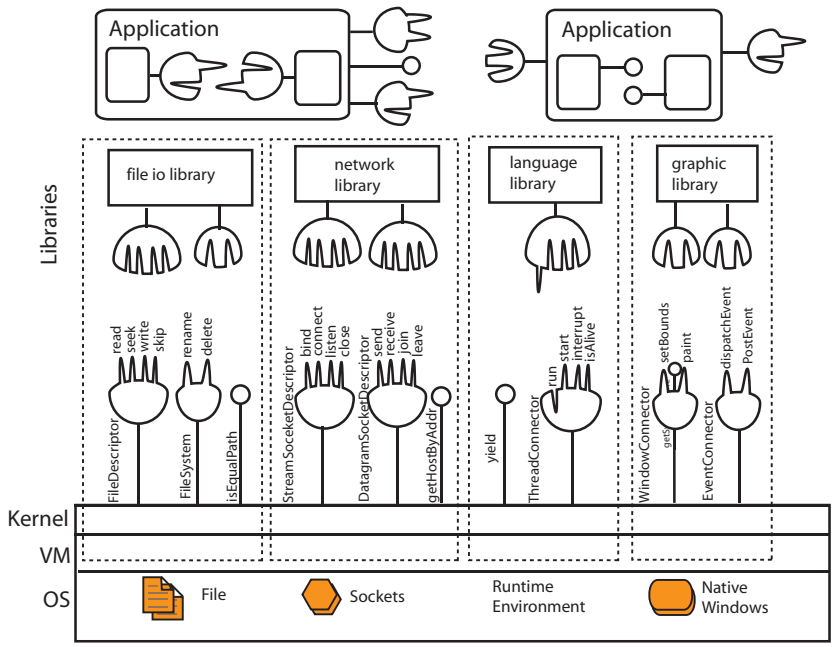
As mentioned before, one of the main reasons for defining a clear application-kernel boundary is so all access control checks can be made at this interface. To be precise, a connection to an application is established only if the kernel's security policy allows it. For services, proper permissions are checked each time before the service is invoked by user applications. In Fig. 1(b), the kernel grants the application a connection associated with the file "data.txt" only if the application has the permission to access the file. And, importantly, the kernel does not give the direct file handle to the application: the connection is just a channel through which the file may be accessed. Since only the kernel holds the file handle, it can revoke the application's access to the file at any point by disconnecting.

Unlike the Java J2SDK (Fig. 2(b)), libraries in the μKVM do not have any special system privileges and they interact with the kernel on services/connectors to fulfill their functionalities. Applications, with proper permissions, also have the option of directly interacting with kernel connectors/services without going through libraries. The two applications in Fig. 2(a) communicate with the kernel via libraries, or directly on a kernel connector.

A simple view of how the μKVM can be implemented is that we take a Java system library such as `java.io` and divide it into two parts, a non-privileged I/O library, and a core file operations in the kernel. The fact that such a split is possible is because a very small interface to the most low-level system routines can be constructed. The principle behind the success is that typically there are only a few, low-level channels of data in and out of an application.

2.2 Design Overview

Fig. 3 shows a static view of the μKVM, detailing the design of the interface between the kernel and system libraries. In the following



*not all services and plugins/plugout on connectors are shown due to lack of space

Figure 3. The μKVM design overview

sections, we first focus on the core part of the μKVM, the kernel-application interaction, and then in Sec. 6 we address how single applications can be divided into multiple security domains and secured with intra-application connectors.

There are four types of system resources that are crucial to a language runtime system: the file system, network sockets, the language runtime environment, and the graphical user interface. A secure system needs to protect these resources from unauthorized use. Fig. 3 illustrates how the μKVM kernel guards system resources and at the same time exposes a set of connectors/services so that applications are able to access resources with proper permissions.

The interface exported by the kernel represents the bare minimum set of connectors and services. The file system has two connectors, `FileDescriptor` and `FileSystem`, which plug out essential operations for accessing a file or the local file system, respectively. Operations exported on `StreamSocketDescriptor` and `DatagramSocketDescriptor` are for stream and datagram communication using network sockets. The Runtime Environment component in the diagram is the means whereby an application may influence its own execution, via `ThreadConnector`, *etc*. A GUI component of a system at the lowest level consists of interactions with local graphic devices and the event queue, modeled by the connectors `WindowConnector` and `EventConnector` in the μKVM.

Fig. 3 also helps illustrate the difference between connectors and services. A connector is used for persistent system resources that are dynamically allocated. For instance, a connection on `FileDescriptor` is a handle to an opened file. For a one-time interaction with the kernel, a service interface is more appropriate, e.g. the `getSystemTimeZone` service.

Every dotted square in Fig. 3 represents a module that has an equivalent J2SDK system component. For example, the network library in the diagram has matching interfaces of the kernel's network component interfaces. Together, they form a module that has the same functionality as the J2SDK `java.net` package.

The μKVM implementation so far includes complete file I/O, network, and thread implementations, and core GUI functionality. We have up to now left a few language features out of the design, including reflection and object serialization. These features are important but not essential to a language runtime and we decided to leave them out for future work. The design reflects the essence of the μKVM: only a handful of essential operations are performed by the kernel and in a well designed architecture it is possible to put a clear and declarative interface between the kernel and user applications.

3. Implementation of the Kernel

We have implemented a prototype of the μKVM by modifying the Java J2SDK source code tree[1].

To fit the design into the J2SDK, the μKVM connector/service architecture has been mapped on to Java classes. Such a mapping allows us to test our ideas, and in fact does not suffer significant performance overhead, as we show in Sec. 7.

The `java.kernel` package implements the kernel functionality and provides service/connector interfaces. The system libraries, such as `library.io.*`, are re-implemented as non-privileged libraries that interact with the kernel on services/connectors. They are closely based on the analogous `java.*`, but have no special privileges and obtain system services from the kernel like any other

[1] The J2SDK1.4.2 source code is obtained through the Sun Community Source License

applications would do. The Java Virtual Machine is modified to accommodate these new components of the μKVM.

In this section, we elaborate how the kernel is constructed. Building libraries in the μKVM is discussed in Sec. 4.

3.1 The Kernel Runtime and the java.kernel Package

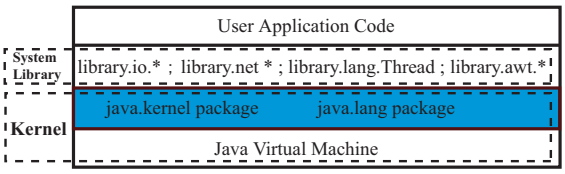


Figure 4. The μKVM kernel structure

We loosely used the term "kernel" in previous sections; there are in fact two closely related concepts, the kernel and the `java.kernel` package. The kernel is a runtime component, and as shown in Fig. 4 the kernel itself includes a Java package named `java.kernel`. We call this package the *kernel package*. We inject this package into the J2SDK as system code by giving it the prefix `java` and placing it in the system directory of the J2SDK.

The `java.lang` package, except the Thread class, is treated as part of the kernel in the current prototype. This is simply because we have at this point only implemented the `Thread` portion of this package; all potentially unsafe features of `java.lang` should eventually be implemented as connectors/services, and `java.lang.*` can then be replaced with a safe `library.lang.*`. Some examples of potentially unsafe classes in `java.lang` include `ClassLoader` and `System`.

3.2 Services and the java.kernel.President class

In the prototype, there is a special `President` class in the kernel, which is the interface used by applications to request new connections and access kernel services. The kernel exports services via the `President` in the form of public methods. A unique `President` object in the μKVM represents the handle to the kernel and is foreknown by all applications. Fig. 5 lists all kernel services currently declared in this class. Data passed in and out these services are subject to certain syntactic restrictions discussed in Sec. 3.4.

```
public  static  President   getPresident();
public  static  Connector   connect(Connector);
public  static  String      getProperty(String);
public  static  int         NetworkInterfaceGetNumber();
public  static  void        NetworkInterfaceGetByIndex
                            (int, NetworkInterface);
public  static  void        NetworkInterfaceGetByName
                            (String, NetworkInterface);
public  static  void        NetworkInterfaceGetByInetAddress
                            (byte[], NetworkInterface);
public  static  String      getHostByAddr(boolean, byte[])
public  static  String      getLocalHostName(boolean)
public  static  boolean     isEqualPath(String, String);
public  static  boolean     impliesIgnoreMask(String, String);
public  static  void        checkPermission(String);
public  static  void        yield();
public  static  void        sleep();
```

Figure 5. Kernel services exported via the `President`

3.3 Connectors

The kernel provides connectors for accessing persistent system resources like files and sockets. Since Java has no built-in notion of connector, each type of connector must be individually implemented in terms of Java classes.

A kernel connector is instantiated from a connector class in the `java.kernel` package. These kernel connector classes are declared package private so that only the kernel can access them. On the other hand, a kernel connector needs to export its operations to the application connector that is connected with it. Public connector interface classes for connectors are introduced for this purpose. Each established connection is constituted at runtime by two connector objects, with mutual references from each other. One of these connector objects belongs to the kernel, and the other is part of the application.

Even though two matching connectors are mirrors of each other, with reversed plugins and plugouts, a common connector interface that includes declaration of all plugins and plugouts is used for both. In the connector objects themselves, a plugout is implemented as a method of the connector class with a concrete implementation, and a plugin is implemented simply as a forwarder to its corresponding plugin.

Extensive changes have been made to the Java Virtual Machine for building the μKVM kernel. Take GUI as an example, Fig. 6 shows the detailed GUI architecture of the μKVM. The system-wide event dispatch thread used in the J2SDK is replaced with application-level threads. Events are delivered from the kernel to an application via `EventConnector`. Native windows are managed by the kernel and accessible to applications via connectors. Notice that the kernel only posts events to an application event queue, the application's private dispatch thread is responsible for invoking the event listeners. Similarly, applications may request to draw on corresponding native windows via `WindowConnector`, instead of operating on them directly using JNI. This architecture effectively removes privileges from the original Java AWT/Swing libraries and puts the kernel in full control of GUI resources.

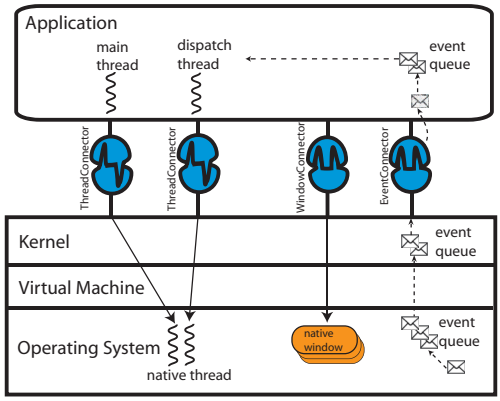


Figure 6. GUI model in the μKVM

Fig. 6 also illustrates the threading model of the μKVM: threads are built on connectors. A GUI application has at least two such thread connections at runtime: one is the connection on the main thread fired by the VM to launch the application and the other on the event dispatching thread instantiated from `library.lang.Thread` class, a μKVM library class. We will discuss libraries in the next section.

Corresponding to the kernel's fixed set of connectors, we provide matching application side connector classes for applications to directly import and use.

3.4 Syntactic Restrictions on the Kernel Interface

In order to maintain the invariant that all interactions with the kernel are solely through its services and connectors, the kernel package must follow some strict syntactic rules. First of all the ker-

nel package has no static fields visible to applications. Connector/service interfaces are the only public components that are accessible to applications. Moreover, we only allow the exchange of primitive data and certain immutable objects on these interfaces.

DataType defined in Fig. 7 represents types that can be exchanged on the kernel-application interface. *Datatypes* are safe to be passed across domains because intuitively they are pure data objects, without references to unrestricted objects. *DataType* includes *PrimitiveType* and *StructType*. *PrimitiveType* are Java primitive types such as `int`, `java.lang.String`, arrays of *PrimitiveType*, *etc*. *PrimitiveType*s are passed by copy. *StructType*s are simple classes that are similar to the `struct` in C. Their contents are restricted as follows: *StructType* classes can only have instance fields of *PrimitiveType*, and they can have neither static fields nor methods other than methods for getting field values. Only two *StructType* classes, `NetworkInterface` and `DatagramPacket` (appearing in Fig. 5), so far have been needed and implemented in the kernel package. This again demonstrates the how fundamental the kernel-application interface can be in practice.

We assert that the kernel-application boundary is sound by showing that the applications share no references to non-*DataType* objects with the kernel, except references to the `President` and kernel connector objects. Here is a proof sketch. The kernel package contains no public static fields, from where object references internal to the kernel can escape. The only way that data can be exchanged is via connectors and services, and this data must be *DataType*s only. *PrimitiveType*s are passing by copy so exchanging them does not introduce any reference sharing between the kernel and applications. *StructType* objects have only immutable *PrimitiveType* fields, an application cannot get any other object references from a *StructType* object.

Currently we check by hand to guarantee that the kernel interface obeys the restricted type grammar. This is a fairly easy job since the kernel-application interface only includes a handful of connectors/services: Fig. 3 illustrates that there are 7 kernel connectors and Fig. 5 shows that the kernel now has a total of 14 services. Eventually an automatic grammatical checker will be developed.

4. Library Implementation

The existing J2SDK libraries cannot be placed directly into the μKVM because the `java.*` libraries are privileged system code with all permissions. For example, every class in `java.io.*` has the privilege, no matter such privileged is actually needed for the class to perform its task, to bypass the virtual machine and interact with the local operating system directly via Java Native Interface (JNI)[Lia99]. Only the kernel in the μKVM have such privileges so we need to develop a new set of libraries specifically tailored for the μKVM.

We have so far developed `library.io.*`, `library.net.*`, `library.lang.Thread`, and `library.awt.*`. They have nearly identical functionality as their java counterparts, but do not need special system privileges to perform their jobs. Due to space limitation, we focus our discussion on the I/O and GUI libraries. The `library.net.*` is developed in a similar way as the I/O library, while the `library.lang.Thread` has been briefly covered in Sec. 3.3.

The most difficult part of the μKVM library implementation is to move a large amount of native code in original Java system libraries to the kernel. Native code is a back door through which an application can bypass the language runtime, thus libraries are not allowed to have native methods in the μKVM as in the current prototype. Any use of native code in μKVM should properly be viewed as a kernel extension since it is not allowed in application space. The restriction on native code does not mean that our model is less flexible than the J2SDK: when a secure runtime is required, the J2SDK also must disallow arbitrary native code with the help of the Java `SecurityManager`. A production μKVM should support native code kernel extensions, which of course must be programmed very defensively. The other major challenge is to remove privileges originally possessed by the Java system libraries, which means the μKVM libraries have to be able to fulfill their tasks without any `doPrivileged()` calls.

4.1 I/O Library Implementation

4.1.1 Review of java.io

For comparison reason, let us first review how the J2SDK `java.io` package works. Fig. 8(a) includes the major classes of the J2SDK `java.io` package. The dependence of these classes on native code is drawn as dotted lines between classes and the Java Native Interface box.

Instances of `java.io.FileDescriptor` serve as an opaque handle of an open file or socket. `FileDescriptor` itself does not perform any I/O operations, but mainly serves as data holder storing references to opened native file handlers. The `RandomAccessFile` class supports random file access. The `FileInputStream` and the `FileOutputStream` are for raw bytes read and written, respectively. Each of the three classes has its own set of native methods performing I/O operations such as open/read. To enforce access control on I/O, the Java SecurityManager has to be called to perform stack inspection.

`java.io.FileSystem` is a class for local file system abstraction. It defines machine-dependent operations such as file canonicalization. The `File` class is an abstract representation of file and directory path names. It delegates file operations to the `FileSystem`.

4.1.2 The library.io package in the μKVM

Fig. 8(b) shows the class structure of the `library.io` package for the μKVM. Notice that this package is pure java code without any dependence on JNI, in contrast to `java.io` package.

Almost every class in the `library.io.*` has a peer class with the same class name in the `java.io` package. The inheritance structures among classes are almost identical in both packages. However, despite the surface similarities, the internal details of the implementations differ greatly.

Different from their `java.io` counterparts, `RandomAccessFile`, `FileInputStream` and `FileOutputStream` in `library.io.*` no longer have their own native file I/O implementations. Instead, they uniformly use a kernel connector, the `FileDescriptor`, via which the kernel exports file operations. This structure adequately illustrates how the kernel can have absolute control over all privileged I/O operations and at the same time provide a very minimal interface with adequate support for library/application functionalities.

Let us now show how a connection with the kernel is built with an example. Fig. 9 shows a code snippet opening a file. When a `RandomAccessFile` object is instantiated, its constructor tries to create a `library.io.FileDescriptor` object. The `library.io.FileDescriptor` class constructor in turn consults the `President` to access the file `"data.txt"`. If the kernel grants this request, it returns a kernel `FileDescriptor` associated with the requested file. At this point, the two `FileDescriptor` objects, one from the library and the other from the kernel form a persistent connection via which the newly created `RandomAccessFile` object gets read/write access to the designated file. Disconnecting the connection happens when the file is closed or the kernel's new security policy requires the access to be revoked. Another connector implemented in the I/O library is the `library.io.FileSystem`.

Definition 1 (*DataType*)
DataType ::= *PrimitiveType* | *StructType*;

Definition 2 (*PrimitiveType*)
PrimitiveType ::= **int** | **short** | **byte** | **long** | **char** | **float** | **double** | **boolean** | **String** | **array of** *PrimitiveType*;

Definition 3 (*StructType*)
StructType ::= **public final class** *Identifier*† {*ClassBodyDeclarations*opt}
ClassBodyDeclarations ::= *ClassBodyDeclaration*
ClassBodyDeclarations ClassBodyDeclaration
ClassBodyDeclaration ::= *ConstructorDeclaration*
FieldDeclaration
MethodDeclaration
ConstructorDeclaration ::= **public** *Identifier*† (*FormalParameterList*opt) {*ConstructorBody*† }
FormalParameterList ::= *FormalParameter*
FormalParameterList, *FormalParameter*
FormalParameter ::= *PrimitiveType VariableDeclaratorId*†
FieldDeclaration ::= *FieldModifiers PrimitiveType VariableDeclarators*†
FieldModifiers ::= *FieldModifier*
FieldModifiers FieldModifier
FieldModifier ::= **public** | **protected** | **private** | **final** | **transient** | **volatile**
MethodDeclaration ::= *PublicAccessorMethod*
PrivateInternalMethod
PublicAccessorMethod ::= **public** *PrimitiveType* **get***Identifier*† (*FormalParameter*) {**return** *Identifier*; }
PrivateInternalMethod ::= **private** *ReturnType*† *Identifier*† (*FormalParameterList*) {*MethodBody*† }

†The denoted term is defined in the Java Language Specification

Figure 7. Type Definitions

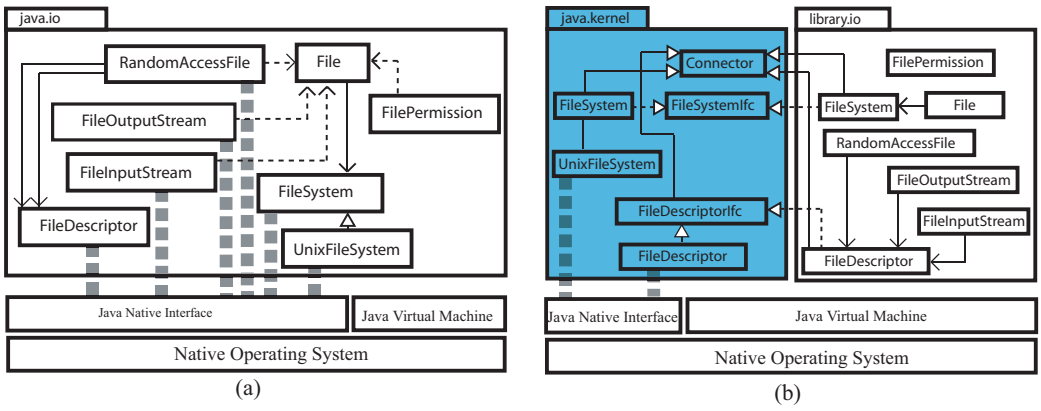


Figure 8. (a) Java io library (b) μKVM io library

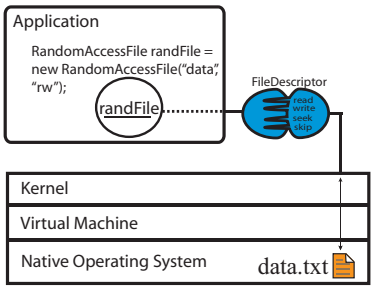


Figure 9. Access file using RandomAccessFile

4.2 GUI library

Implementing the μKVM GUI model, shown in Fig. 6, is a more challenging task. Java GUI consists of AWT and Swing. The `java.awt` package alone involves more than 400 classes tightly coupled with more than 100 platform-dependent classes. However, despite its tremendous size, the Java AWT has only a few core classes: `Component`, `Container`, `Windows`, `Frame` and `Dialog`. Those classes encapsulate the most fundamental functionalities such as the event dispatching mechanism. Swing is extended from the core AWT classes. With the core Java AWT classes implemented, building a Swing package for the μKVM is relatively easy.

Therefore, our focus is to refactor the core Java AWT classes into the μKVM GUI model: all platform-dependent code such as calls to native methods to create windows in those classes needed to be moved to the kernel, and a framework supporting the interactions between the kernel and the new `library.awt` classes on services/connectors has to be introduced.

At this stage we have implemented a primitive μKVM GUI component, which includes the kernel side graphic supports as discussed in Sec. 3.3 and a machine independent `library.awt` package consisting of `Component`, `Container`, `Window`, `Frame` and around 100 supporting classes. These two parts cooperate on `WindowConnector` and `EventConnector`.

Although the GUI component is still in its debugging stage, the fact that the prototype now supports essential AWT operations

such as showing a frame and posting a paint event to the frame demonstrates the feasibility of the proposed GUI model.

5. The Security Architecture

In this section we describe how access is secured in the μKVM, and briefly justify its correctness.

The μKVM prototype reuses the Java protection domain framework: we keep the J2SDK concepts of codebases, policies, and permissions [Gon]; we are only changing how the policy is enforced. For policy enforcement, we re-implement the `checkPermission()` of the Java `SecurityManager` to simply check if the caller has required permissions; no stack inspection is performed. This does not weaken the security properties of the μKVM because all protected resources in Java are protected by the kernel-application boundary of the μKVM. The difference is that Java performs such protection using stack inspection while the μKVM guards sensitive resources by placing them in the kernel, behind the kernel-application interface.

The clear interface defined on connectors and services means that all permissions can be checked at that gateway since there are no side channels or callbacks. The μKVM implementation of `checkPermission()` is invoked solely by the kernel at the start of service invocations and when connections are requested on kernel connectors. If the immediately invoking code has required permissions, it would succeed in these operations.

Object reference sharing breaks component isolation, so confining references across the kernel-application boundary is a key design requirement of the μKVM. Applications can obtain two types of references to the kernel-application interface: references to the `President` object (representing the kernel), and references to kernel connector objects. The `connect()` method of the `President` can only return kernel connectors to callers to form connections with the kernel. And, on these connections, only *DataType* can be passed in and out. Data passing on kernel services follows the same rule. In Sec. 3.4, we have proved that no uncontrolled references to kernel objects can be leaked out of the kernel-application interface. Moreover, the `java.kernel` package doesn't depend on any user classes. This prevents the kernel from invoking user class constructors, which eliminates direct callbacks from the kernel to applications.

Exceptions also need to be considered: they can traverse arbitrary domains, so a kernel exception may be propagated to applications. In order to protect the integrity of the kernel interface, the kernel only throws exceptions with at most string messages. Thus exceptions do not become a backdoor communication channel.

6. Securing Multiple Application Components

Large scale programs typically consist of several components, each of which has different responsibilities, and needs different permissions to accomplish its task. For instance, a I/O component requires permissions for file read/write, while a component conducting pure mathematical computation does not. If the application is configured into multiple code ownership domains, each component can be given only the permissions it needs, in line with the Principle of Least Privilege.

The application in Fig. 10 is divided into two components, Client and LogService. This division allows fewer privileges to be given to each component: the Client does not need I/O permissions and therefore has an empty set of permissions. LogService provides a logging service of the operations performed by the Client, and has permissions which allow it to access a local log file. In our model, we re-use the existing J2SDK notion of a codebase and define the two as separate codebases with their distinct permissions. This way, the Client may record its data into a log file, without having to have the explicit permission to access the file—the LogService intermediary is the only code needing access to the file. This division achieves the goal of separating permissions among components so that each component only gets the permissions it needs for completing its job.

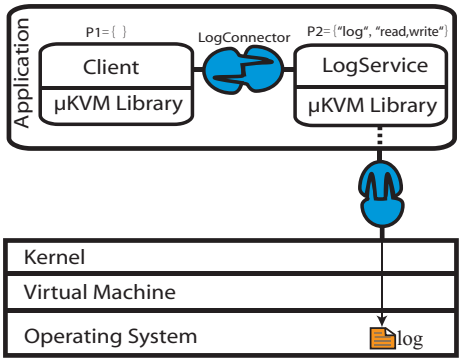


Figure 10. Application consists of code components with heterogeneous permissions

The J2SDK also supports multiple code ownership domains. However, the J2SDK architecture does not prevent references to resources from being passed from one codebase to another. The essential difference between our approach and the J2SDK approach is how such codebase boundary is enforced. We proceed analogously with how we enforce the kernel/library interface boundary, and require the components within an application to interact with each other on connectors/services *only*, which can pass primitive data only. In this example, the Client and the LogService build a connection on `LogConnector`.

The restriction on data passing on connector/service interface sounds severe in that the different code components cannot share references. The comparison is that the J2SDK allows more tightly coupled interaction between domains, but at the expense of a muddled interface which may not even be working. We believe that a clear component boundary is the essential foundation of a secure runtime system and it is a worthwhile tradeoff; it would be better to simply put both components of the application into one codebase rather than use the J2SDK model to lull one into believing there was a secure separation between the two.

Notice that both the Client and the LogService have their own copies of the μKVM libraries. This is because if the two components had shared the libraries, the library code would be operating on permissions coming from different sources, the root of confused deputy problem. The figure also accurately reflects that the μKVM libraries do not have system privileges – they only have the same permissions as the codebase components that use them, which illustrate that library code also gets the minimum permissions needed.

We have implemented the architecture in Fig. 10 by using a different class loader to load each codebase, and thus each component will load their own copy of the μKVM libraries. This is not an efficient implementation, but it does serve as a proof of concept.

7. Performance Evaluation

Microkernel operating systems historically suffer from runtime performance problems, which makes microkernel-based architectures potentially less appealing for practical systems. To show that the μKVM does not have such a performance pitfall, we benchmarked the performance of the μKVM on accessing system resources such as reading from files and transferring data on sockets. We designed

our own test programs[2] because we could find no existing benchmarks that met our need. We also benchmarked the functionality of the kernel and the system libraries using mauve test suite[3] to show that the observable functionality to the user was unchanged, in spite of fundamental changes in architecture.

The benchmark results show that the μKVM has performance comparable with the J2SDK. Most importantly, the μKVM has significant efficiency advantages in providing a secure runtime environment. For instance, when security was switched on in both the J2SDK and the μKVM, the μKVM outperformed the J2SDK in both speed and memory. In particular, the μKVM was 27% faster than Java when 500 files were opened and 19% faster when the file number increased to 1000. As for memory consumption, the μKVM consumed only 0.01% more memory in the 500 case while 1.63% less in the 1000 case. Details about the benchmark and results are omitted here for lack of space and can be found at the μKVM website[2].

8. Related Work

8.1 Application Isolation

Much research has been done to provide isolated application runtimes that allow multiple applications to run in a VM without interfering with each other's computation. The application domains of the .NET CLR [App, FJ03, MG01], JKernel [HCCDH97, HCC$^+$98, JKe], Isolates [JSR, BG98, Cza00, DBC$^+$00] and the Multitasking Virtual Machine (MVM) [CD01, CL01, DC05] fall into this category. These ideas could also be incorporated into the μKVM, but since that problem is mostly solved, our focus is on kernel-application and intra-application interaction.

8.2 Securing the System Domain

JOS is a JKernel extension in which file operations are put in a protected domain and can only be accessed via the JOS server. Theoretically, system libraries in the JKernel can be placed in separate privileged domains as the JOS does for file operations. However, they have to be carefully re-implemented so that only the most essential parts are in the privileged domains, otherwise cross-domain communication would slow down the whole system significantly.

The KaffeOS [BH99, BHL00] acknowledges the importance of the user/kernel distinction to maintain system integrity like we do in the μKVM. However, user/kernel modes in the KaffeOS indicate different environments with respect to termination and resource consumption. For instance, an application running in user mode should not hold kernel locks in order to terminate safely. In the μKVM, on the other hand, user/kernel modes represent domains with or without security privileges. Such difference in philosophy results in different system architectures. Specifically, part of the runtime libraries in the KaffeOS are fully trusted but run in the user mode, while the μKVM disallows any privileged code to exist in user space. The KaffeOS enables direct sharing between processes via a shared heap.

The SPIN operating system [BSP$^+$95] uses the safe language Modula-3 to ensure safety within the kernel. It allows safe extensions to be loaded into the kernel dynamically and then integrated themselves into the existing infrastructure. SPIN uses software protection to protect the OS kernel on dynamic extensions and implements protected communication using procedure calls. But it does not provide any code-based access control of the type found in architectures such as Java and .NET [Raz02].

The JX operating system [GFWK02] builds a complete operating system in Java. It has a small microkernel containing functionalities such as system initialization. Other components are pure Java code and may be loaded into different domains using different system configurations. JX components use RMI-style portals to communicate. The JX and the μKVM share a similar design philosophy in the sense that non-kernel components in μKVM cannot have native code to bypass the control of the kernel. But the two systems are otherwise addressing different problems. JX focuses on performance of low-level operations such as memory management, but not on high-level code-based access control policies.

8.3 Control of Intra-application interference

The Code-based security in the J2SDK gives fine-grained access control. But the lack of runtime enforcement for keeping different codebases from interfering with each other leads to problems such as authorizations implicitly being extended to client code. MARCO [PFKS05] uses static interprocedural analysis to detect such implicit leaks of authority and "tainted" variables which are untrusted values and might have been used by the high authority code without proper sanitization.

In the μKVM, code components with different permissions in an application are isolated in sub-domains and only can interact on connector/service interface. Consequently, the μKVM does not have the problem of implicit authority transfer. This illustrates how the μKVM improves on the J2SDK. The "tainted" variable detection performed by MARCO is still a useful utility for the μKVM since it is a fundamental property of any data transferred between security domains.

8.4 Capability-based Security

Capabilities have been used in both computer systems and languages for security. KeyKOS [Key] and its successor EROS [ERO] are pure capability operating systems. The E language[E L] is a capability-based distributed programming language, in which object references are treated as capabilities. The revocation aspect of connections on connectors relates the μKVM to capability revocation, But the similarity stops here as connections between two parties in the μKVM are not transferable, whereas capabilities are. The lack of transferability of connections prevents them from unauthorized use by external parties. We can enforce stronger constraints because we have a purely local model for securing code, and capabilities are a more general security model.

9. Conclusion

In this paper, we proposed a novel microkernel-based language VM, the μKVM, in which the kernel manages system resources and implements a core set of low-level system functionalities. System libraries such as I/O library are implemented outside the kernel so that stack inspection is no longer needed for protecting them. All kernel-application and intra-application interactions are declared on explicit connector/service interfaces. The μKVM has a small trusted codebase which includes only the kernel and the underlying virtual machine. It is well-known that the smaller the trusted codebase is, the lesser the impact of making programming mistakes.

The security architecture of the μKVM is simple and declarative. Accesses to all sensitive system resources are through publicly declared connectors and services. Permission checks are explicitly placed on those interfaces, and are controlled by the kernel. The simple, declarative nature of the μKVM security framework has obvious benefits: programmers can easily understand the security architecture and hence build their secure applications confidently. Moreover, the Principle of Least Privilege can be easily deployed in the μKVM via imposing fine-grained security schemas on interfaces.

[2] `www.cs.jhu.edu/~xiaoqilu/MicrokernelVM`

[3] `http://sources.redhat.com/mauve/`

Benchmarks show that our μKVM prototype can provide a secure runtime environment with less overhead than the J2SDK, thus avoid the runtime performance pitfall commonly found in microkernel-based operating systems.

References

[App] Application Domain FAQ. `http://www.gotdotnet.com/team/clr/AppdomainFAQ.aspx`.

[BG98] D. Balfanz and L. Gong. Experience with secure multiprocessing in Java. In *Proceeding of ICDCS'98*, Amsterdam, The Netherlands, May 1998.

[BH99] Godmar Back and Wilson C. Hsieh. Drawing the red line in Java. In *Proceedings of the 7th IEEE Workshop on Hot Topics in Operating Systems*, pages 116–121, Rio Rico, Arizona, March 1999.

[BHL00] G. Back, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000. USENIX.

[Bro] K. Brown. Security in .NET: Enforce code access rights with the Common Language Runtime. `http://msdn.microsoft.com/library/default.asp?url=/msdnmag/issues/01/02/cas/toc.asp`.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.

[CD01] G. Czajkowski and L. Daynes. Multitasking without compromise: a virtual machine evolution. In *OOPSLA 2001*, 2001.

[CL01] G. Czajkowski and M. Wolczko L.Daynes. Automated and portable native code isolation. Technical Report TR-2001-96, Sun Microsystems, Inc., 2001.

[Cza00] G. Czajkowski. Application isolation in the Java Virtual Machine. In *Proceedings of the 15th ACM SIGPLAN conference on OOPSLA 2000*, 2000.

[DBC+00] D. Dillenberge, R. Bordawekar, C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. St. John. Building a Java Virtual Machine for server applications: The JVM on OS/290. *IBM Systems Journal (Java Performance issue)*, 39, 2000.

[DC05] L. Daynes and G. Czajkowski. Sharing the runtime representation of classes across class loaders. In *ECOOP 2005*, 2005.

[DW02] S. Lange D. Watkins. An overview of security in the .NET framework, January 2002. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/netframesecover.asp`.

[E L] E language. `http://www.erights.org`.

[ERO] EROS: The extremely reliable operating system. www.eros-os.org/.

[FJ03] Adam Freeman and Allen Jones. *Programming .NET Security*. O'Reilly & Associates, Inc., Sebastopol, CA, June 2003.

[GDFR90] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–96, Anaheim, CA, June 1990.

[GFWK02] M. Golm, M. Felser, C. Wawersich, and J. Kleinoder. The JX operating system. In *Proceedings of the USENIX Annual Technical Conference*, pages 45–48, Monterey, June 2002. `http://www.jxos.org/publications.html`.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2005. `http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf`.

[Gon] L. Gong. Java 2 platform security architecture version 1.2. `http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html`.

[Gon99] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 1999.

[Har88] N. Hardy. The confused deputy. In *ACM Operating Systems Review*, volume 22(4), pages 36–38, October 1988. `http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html`.

[HCC+98] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. Eicken. Implementing multiple protection domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, 1998.

[HCCDH97] C. Hawblitzel, C. Chang, G. Czajkowsi, and T. Eicken D. Hu. SLK: A capability system based on safe language technology, 1997. `http://citeseer.ist.psu.edu/hawblitzel97slk.html`.

[JKe] JKernel and JServer. `http://www.cs.cornell.edu/slk/jkernel.html`.

[JSR] JSR 121: Application isolation API specification. `http://www.jcp.org/en/jsr/detail?id=121`.

[Key] The KeyKOS system. `http://www.cis.upenn.edu/~KeyKOS`.

[Lia99] S. Liang. *The Java Native Interface-Programmer's Guide and Specification*. Addison-Wesley, 1999. `http://java.sun.com/docs/books/jni/`.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, 1995.

[LS04] Y. Liu and S. Smith. Modules with interfaces for dynamic linking and communication. In *ECOOP 2004*, June 2004.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, April 1999.

[MG01] Erik Meijer and John Gough. A technical overview of the common language infrastructure, 2001.

[PFKS05] Marco Pistoia, Robert Flynn, Larry Koved, and Vugranam Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP 2005*, July 2005.

[Raz02] Valentine Razmov. Security in untrusted code environment:missing pieces of the puzzle, March 30 2002. `http://www.di.unipi.it/~giangi/CORSI/LPRA/LECTURES/Razmov02puzzle.pdf`.

[RS02] R. Rinat and S. Smith. Modular internet programming with Cells. In *Proceedings of the ECOOP 2002-Object-Oriented Programming: 16th European conference*, pages 256–280, Malaga, Spain, June 2002.

[SS75] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE 63(9)*, pages 1278–1308, September 1975.

[SS00] C. Skalka and S. Smith. Static enforcement of security with types. In *Proceeding of ICFP'00*, Montreal, Canada, 2000.

[TH03] A. Ohori T. Higuchi. A static type system for JVM access control. In *ICFP'03*, Uppsala, Sweden, August 2003.

[WF98] D. Wallach and E. Felten. Understanding Java stack inspection. In *Proceedings of Security and Privacy'98*, Oakland, California, May 1998.