A Secure Microkernel Virtual Machine

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

ABSTRACT

In this paper, we develop a novel microkernel-based virtual machine, the μKVM. It is a microkernel architecture because the size of the trusted system codebase is greatly reduced in comparison to VM's such as the Java Virtual Machine. The μKVM kernel manages sensitive resources such as I/O, and implements a minimal set of low-level system operations. System libraries are implemented outside the kernel, and run in user mode. All interactions between the kernel and applications are declared on explicit interfaces, and security policies are also enforced at these interfaces. We test our architecture in a μKVM prototype developed for Java and show how the microkernel architecture supports the existing functionality of the J2SDK. The prototype is benchmarked, and the results show that our implementation compares favorably with the J2SDK and so the architecture does not appear to be a burden on running time.

1. INTRODUCTION

Dynamic component composition is becoming essential in modern software construction [26], and often arises in the form of downloaded code components. Supporting this flexibility also introduces new challenges in securing language runtime systems. The Java J2SDK [20, 12] and the Microsoft CLR [22] both define security architectures that allow safe execution of untrusted components. The security mechanisms in the two language runtime systems are similar as far as our interests go [11, 10, 7, 3], and we somewhat arbitrarily have chosen the J2SDK to focus on in this paper.

Java incorporates a code-based access control policy, based on a set of `Permission`s for access to various resources, and a policy that maps from the different codebases to permissions. In the Java security architecture, all system libraries (`java.*`, `javax.*`) are highly trusted and have privileges that user classes usually do not have. However, this design choice that grants full privileges to all system libraries is potentially problematic. First of all, there is no absolute guarantee that programmers make no mistakes in programming libraries, and giving so many privileges to this large codebase is not a good design methodology. Secondly, even if the system libraries are correct, they have to be guarded from being attacked or lured into a "confused deputy" [13], in which libraries become victims of malicious applications and misuse their privileges to do things on behalf of attackers.

The runtime stack inspection algorithm [10, 28] solves the confused deputy problem that plagued the original Java security architecture, but it is a far from perfect remedy. Perhaps the most serious shortcoming is how the Java Security Architecture scatters security related code snippets throughout library code, which makes it hard to understand exactly what security policy is being enforced, and hard to see whether all gates are in fact guarded. Moreover, the principle of least privilege [24] has not been applied to Java libraries: each library in fact only needs access to the relevant resources it is operating on, *e.g.* `java.io` file libraries do not need network access. But, to enforce this in the J2SDK would introduce a large runtime cost because stack inspection could not collapse consecutive system stack frames any more. Researchers have developed static, declarative approaches to stack inspection via type systems [25, 27], but not all checks can be performed statically due to the fundamentally dynamic nature of components.

In this paper we design the μ-Kernel Virtual Machine (μKVM), a novel language runtime framework with a simple and declarative security architecture. The main design goals of the μKVM are to decrease the size of the core trusted codebase, and to put a clear, inviolable interface between the trusted codebase and less trusted code. The *kernel* is the small trusted system codebase; since it is small in contrast with the large J2SDK system libraries, we term the architecture a *microkernel VM* architecture. Kernel-application interactions and the system security policy enforcement are defined solely on the interfaces between the kernel and the application. *Connectors* and *services* are the two types of kernel-application interfaces we define, representing persistent heavy-weight 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 almost no change, a fact that further illustrates the power and generality of the approach. By running standard Java benchmark suites, we show the original Java code functionality is preserved, and some limited performance benchmarks indicate that the μKVM compares favorably with the J2SDK and so the microkernel does not appear to introduce unacceptable overhead. Indeed, when the Java Security Manager is running, the μKVM implementation is faster. The prototype at this point implements enough of the interfaces, namely those for file, network, and threads, to show feasibility of the approach; implementing GUI libraries and a few other features is ongoing work.

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

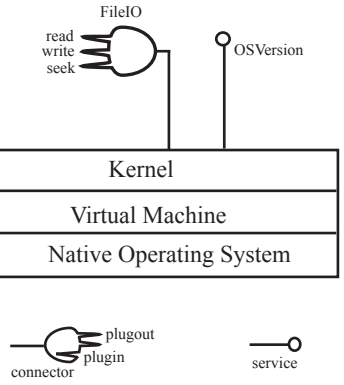

Figure 1: Building blocks of the μKVM

A microkernel virtual machine is analogous to a microkernel operating system [19, 9]: the size of the codebase with special privileges is held to a minimum. Features that must be in the kernel include the core language execution model, bytecode execution, threads, garbage collection, code loading, etc. 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.

Our μKVM has an additional important property beyond the small size of the trusted codebase. The kernel-application code boundary is clearly delineated via interfaces of interaction, and these interfaces are the *only* means for application code to interact with the kernel: there are no backdoors, callbacks, shared references, etc. Operating system kernels also have an interface since they are running in a distinct process; but our interfaces of interaction are more absolute than in an OS kernel: unlike operating systems there are no "shared pages" of memory in our architecture, and so *all* interactions with the kernel are via the interfaces and not by some indirect channels. Lack of backdoors makes the design more declarative, and less prone to errors in its definition and use.

The central component of our virtual machine is the *kernel*. 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 kernel defines two types of interfaces, *connectors* and *services*, for applications to communicate with it. Connectors are designed to be used for long term interactions with objects such as files, and services are for simple querying. Connectors 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 import *plugin* and export *plugout* operations. Runtime connections have to be established before connector plugins/plugouts can be used. For instance, the kernel in Fig. 1 allows a connection to a low-level file via connector "FileIO", which exports three plugouts, "read", "write" and "seek". Services are the interfaces for standard client/server style invocations. "OSVersion" in Fig. 1 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 service requests are not persistent and are more elegantly implemented as services; thus we provide both connectors and services. The notions of connector and service come from our previous work on component interfaces [23, 21].

Connections 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. 2 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 coupled connectors in dark shade to represent runtime connections as shown in Fig. 2, and plain connectors as pictured in Fig. 1 to symbolize static connector interfaces. We follow this convention throughout this paper.

As mentioned above, 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 con-

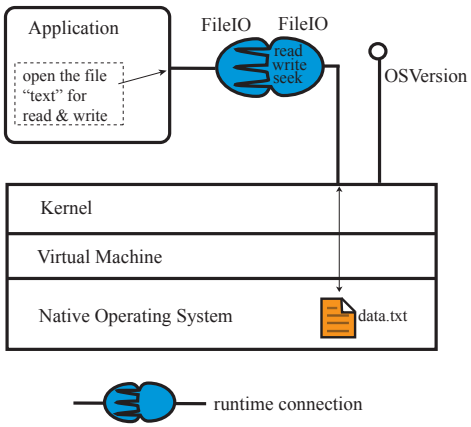

Figure 2: A connection on connectors

nection 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. 2, 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 itself, it can revoke the application's access to the file at any point by disconnecting. The revocability of connections makes it possible to put new security policies into effect instantly without the interruption of current running applications, which is not possible in the J2SDK. For example, if a file in the J2SDK is open with read permission, the application can always read that file as long as it does not close the file, even if the runtime `java.security.Policy` object is changed to withhold read permissions on the file.

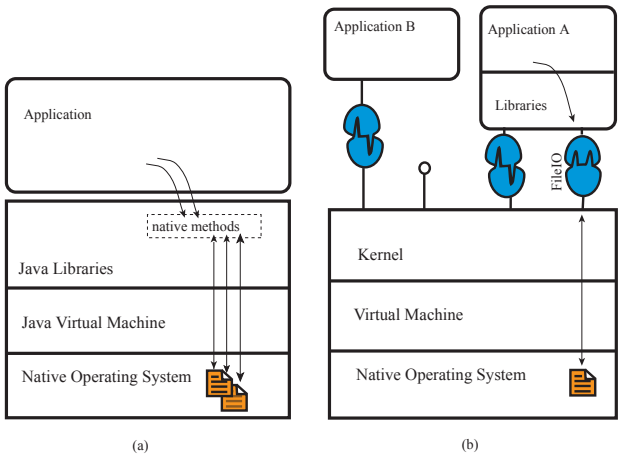

Figure 3: Comparison of the Java J2SDK (a) and the μKVM (b)

Unlike the Java J2SDK, libraries in the μKVM do not have any special system privileges, since the privileged code is all in the kernel, and the libraries will interact with the kernel. Programmers also have the option of gaining more control over system functionality by directly interacting with kernel connectors/services. Application A in Fig. 3 communicates with the kernel via libraries, while application B directly interacts with the kernel. A simple view of how the μKVM can be implemented is that we take Java system libraries such as `java.io` and divide it into two parts, the non-privileged `library.io`, and putting the core file operations in the kernel. It might seem unintuitive that such a split is possible, but in practice we found that a small interface to the very low-level system routines could be constructed, and so this division works in practice. The principle behind the success is that there are always only a few, low-level channels of data in and out of any application. Details of the implementation are described in Sections 3 and 4.

The J2SDK security architecture protects the system domain from being breached by means of stack inspection. But, there is no explicit boundary between different application domains on the same VM. Thus, it is difficult to enforce the fact that permissions from the more privileged domain will not leak into the less privileged one, since the domains can have many references between each other and so it is not clear where one begins and the other ends. In the μKVM, every application can be encapsulated in a distinct domain, as shown in Fig. 3, and these domains can interact with each other via connectors and services *only*. In the μKVM implemented for this paper, we for now simplify the task, and deal only with the case of one application domain; thus the only inter-domain interaction here is between the kernel domain and the (sole) application domain. A full-fledged multi-application μKVM is future work and is discussed in Sec. 7.

2.2 Design Overview

Fig. 4 shows an overview of the whole μKVM design. It is an abbreviated blueprint showing all the connectors, but leaving out some services for lack of space. There are four types of system resources that are crucial to any language runtime system: graphical user interface, file system, network socket and runtime environment resources. A secure system needs to protect these essential resources from unauthorized use. Fig. 4 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: only code that absolutely must be part of the kernel is there. For instance, a GUI component of a system at the lowest level consists of interactions with local graphic devices and an event system, modeled by the connectors `GraphicConnector` and `EventConnector` in the μKVM, as shown in Fig. 4. The graphics library outside of the kernel contains the bulk of the GUI system code. For the file system, the two essential connectors are `newFileDescriptor` and `FileSystem`, which plug out essential operations for accessing a file or the local file system, respectively. Socket operations are provided on `StreamSocketDescriptor` and `DatagramSocketDescriptor`,

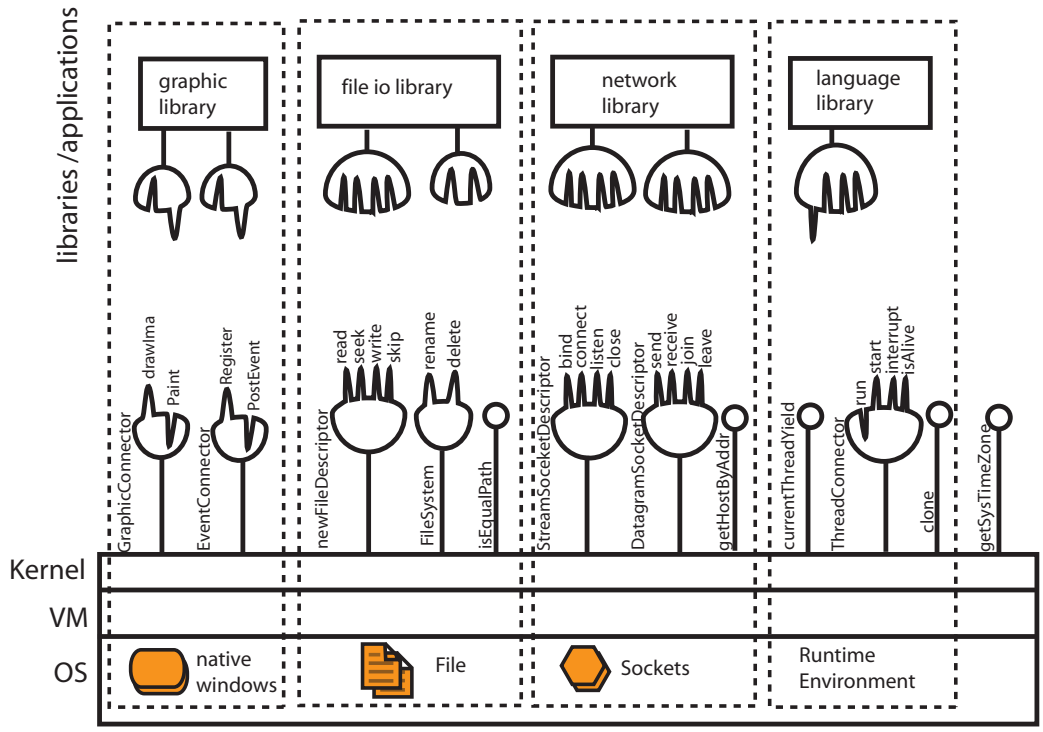

Figure 4: μKVM design overview

for stream and datagram communication. The Runtime Environment component in the diagram is the means whereby an application may effect its own execution, via threads, low-level `Object` protocols, *etc.* The `ThreadConnector` supports threads, forming a connection for each thread.

This design 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 `newFileDescriptor` at runtime is a handle to an opened file. When an interaction with the kernel is one-time operation, a service interface is more appropriate. For example, a request for the currently running thread to yield is not a request to any particular thread connection, and is designed as a service, `currentThreadYield`.

Every dotted square in the Fig. 4 represents a module that has an equivalent java core package. For example, the network library in the diagram, with matching interfaces of the kernel's network component interfaces, forms a module that has the same functionality as the `java.net` package of the J2SDK.

Our implementation of the μKVM so far includes file I/O, network and thread. Details of the implementation are discussed in Sec. 3 and Sec. 4. Amongst the remaining implementation tasks, GUI component is the most complex one and would require more effort. So, we will discuss how to implement it in future work (Sec. 7).

At this point we have left a few language features out of our design, including reflection and object serialization. These features should be realizable, but are complex enough that we decided to leave them out until the core design is fully implemented and performs well in practice.

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, but does not give the cleanest view of the concept of connectors, nor the fastest implementation. It would be more elegant and efficient to build the μKVM ground-up, in which case domain isolation via connectors and services is enforceable at language level, and some data sharing among domains is possible in the implementation to achieve efficiency. Still, as is shown in Section 6, initial benchmarks indicate that the performance of the μKVM is competitive with the J2SDK.

In the prototype, we constructed a `java.kernel` package which implements the kernel services and connectors. We also re-implemented some of the system libraries, such as `library.io` and `library.net`, as non-privileged libraries that interact with the kernel on interfaces. They are closely based on the analogous `java.io` and `java.net`, but have no special privileges and obtain system services from the kernel on its interfaces like any other applications would do. The J2SDK VM is modified to accommodate these new components of the μKVM prototype.

[1]We obtained the source code of the J2SDK1.4.2 through the Sun Community Source License

Although this implementation is specific to Java, we believe the μKVM design can be mapped on to the Microsoft CLR and other secure VM architectures.

In the following sections, we elaborate how the kernel is constructed and how kernel connectors and services are implemented in Java. Building libraries in the μKVM is discussed in Sec. 4.

The current prototype is a partial implementation of the design, it includes network, file, and thread functionality but the GUI libraries and some `java.lang` features are works in progress at this point.

3.1 The Kernel Runtime and the java.kernel package

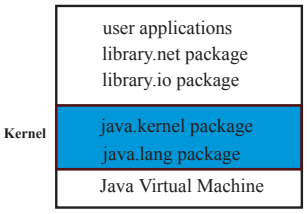

Figure 5: Kernel structure

In previous sections, we loosely used the term "kernel"; 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. 5 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 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, the kernel has a special `President` class 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 any application may request a reference to it. Fig. 6 lists all kernel services currently declared in this class.

3.3 Connectors

The kernel provides connectors for accessing persistent system resources such as 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.

```
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        currentThreadYield();
public  static  void        currentThreadSleep();
```

Figure 6: Services exported on the President

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

Interface classes for connectors are used to pair up connectors. Even though two matching connectors are mirrors of each other, with reversed plugins and plugouts, a common interface is used for both which includes all plugins and plugouts. 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.

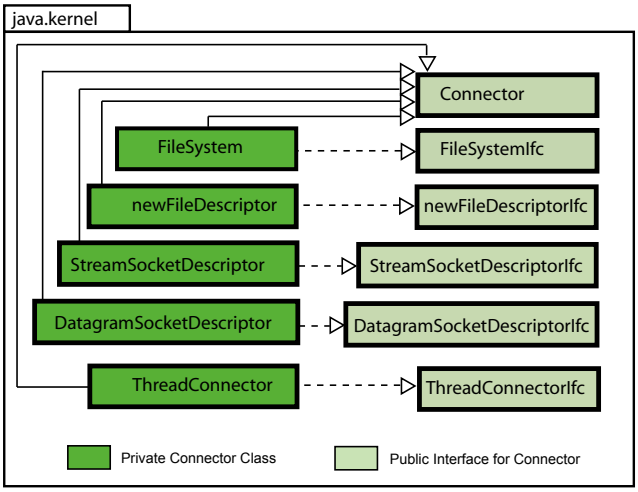

Figure 7: `java.kernel` class diagram

`java.kernel` as implemented in the current prototype provides five connector types , shown in Fig. 7. The interface class `newFileDescriptorIfc` declares basic operations allowed on a file descriptor, such as read and write. The `FileSystemIfc` defines file operations such as renaming a file. The `StreamSocketDescriptorIfc` support stream net-

working, while the `DatagramSocketDescriptorIfc` declares datagram communication. The `ThreadConnectorIfc` offers an abstraction on a thread of execution.

User applications and libraries need to have connectors following declarations of those interface classes in order to interact with kernel connector objects. Connector classes on the application side are always of a standard form, they are defined in a simple XML format and compiled down to Java classes. More details on constructing libraries in the μKVM can be found in Sec 4.

3.4 Syntactic Restrictions on the Kernel Interface

The key to securing the μKVM is to secure the kernel. The invariant we aim to maintain about the kernel is that all interactions with the kernel are solely through its services and connectors. To achieve this, we must design the architecture so that no object references from the user to the kernel space can bypass the kernel interfaces. In the prototype implementation, we achieve this through the use of copy-only data passing between the user and kernel spaces, and only allow for the exchange of primitive data and certain immutable objects on these interfaces. As discussed previously, connectors and services are mapped to Java classes in `java.kernel`. We now describe the syntactic restrictions on the form of these classes that will ensure the above invariants on object references always hold. Currently we are checking by hand whether our kernel library interfaces obey the restricted type discipline, but it is nothing more than a simple grammatical check which could be performed by an automated tool.

PrimitiveType represents the types that can be exchanged on the interfaces between the kernel and applications. Primitive types are safe to be passed across domains because intuitively they are pure data objects, without reference to non-primitive data objects.

PrimitiveType is defined in Fig. 8, and includes Java primitive types such as `int`, `java.lang.String`, arrays of *PrimitiveType* and the *DataType* defined subsequently. *DataType*s are simple classes that are similar to the `struct` of C, but *DataType* contents are restricted as follows: *DataType* classes can only have instance fields of *PrimitiveType*, and they can have neither static fields nor methods other than accessor methods.

Two *DataType* classes so far have been provided in the kernel package. The `DatagramPacket` class represents a datagram packet used in connection-less packet delivery. The `NetworkInterface` class represents a Network Interface made up of a name, and a list of IP addresses assigned to this interface.

Figs. 8 and 9 define the grammatical structure of `java.kernel`. From this structure we may observe the following properties of data coming in and out of `java.kernel`:

1. Only *PrimitiveType* objects are exchanged between the kernel and a non-kernel component, thus no complex object reference sharing is introduced. Specifically, in Fig. 9, *PublicKCMethodDeclaration* defines public kernel classes to take a *FormalParameterList*. According to the definition of *FormalParameterList* in Fig. 8, every parameter of such a public kernel class has to be a *PrimitiveType*. Moreover, return types of those kernel public classes are *PrimitiveType*s as well. One special exception is made to this policy, for the `connect` method of the `President`. It builds a connection by receiving a user connector and returns a kernel one, which is the bootstrap channel for kernel-application communication.

2. Data exchange through static fields is impossible since the kernel package has no static fields visible to applications. The **static** modifier is excluded from the *FieldModifer* defined in Fig. 8.

3. The interactions cross component boundaries are strictly via connector or service interfaces because they are the only public components that are visible to applications.

From these restrictions we may infer that callbacks from the kernel to applications will not arise, because only primitive data will be passed from the application to the kernel and so there are no objects passed by the application that the kernel can make a callback on: the *PrimitiveType* objects are solely public data.

4. IMPLEMENTATION OF LIBRARIES

The existing J2SDK libraries do not fit directly into the μKVM because the `java.*` libraries are privileged system code. For example, any `java.io` class has the privilege to interact with the operating system directly via Java Native Interface (JNI) [18]. So, a new set of libraries need to be developed especially for the μKVM.

Our `library.io` and `library.net` are such libraries. They have nearly identical functionality as the original `java.io` and `java.net` packages, but they do not need special system privileges to do their job because privileged operations are exclusively performed by the kernel. Consequently, they do not have any `doPrivileged` blocks.

Native code is a back door through which an application can bypass the language runtime. There is no guarantee of security in the presence of native code. So libraries are not allowed to have native methods in the μKVM as in the current prototype implementation. Any use of native code in μKVM should properly be viewed as a kernel extension since it is not allowed in application space. This 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 Security Manager. Eventually, we will have our own class loading mechanism as discussed in Sec. 7. Such functionality will be also on connectors/services upon which the kernel can decide if code, native or java, can be loaded.

4.1 I/O Library Implementation

In this section, we first briefly review how the J2SDK `java.io` package works, then discuss the design of the `library.io` package for the μKVM platform.

Definition 1 (PrimitiveType)

PrimitiveType	::=	**int** | **short** | **byte** | **long** | **char** | **float** | **double** | **boolean** | **String** | **array of PrimitiveType** | **DataType**;

Definition 2(DataType)

DataType	::=	**public final class** $Identifier*$ $\{ClassBodyDeclarations_{\text{opt}}\}$
ClassBodyDeclarations	::=	$ClassBodyDeclaration$ $ClassBodyDeclarations\ ClassBodyDeclaration$
ClassBodyDeclaration	::=	$ConstructorDeclaration$ $FieldDeclaration$ $MethodDeclaration$
ConstructorDeclaration	::=	**public** $Identifier*(FormalParamterList_{\text{opt}})$ $\{ConstructorBody*\}$
FormalParameterList	::=	$FormalParameter$ $FormalParameterList,\ FormalParameter$
FormalParameter	::=	$PrimitiveType\ VariableDeclartorId*$
FieldDeclaration	::=	$FieldModifiers\ PrimitivetType\ VariableDeclarators*$
FieldModifiers	::=	$FieldModifier$ $FieldModifiers\ FieldModifier$
FieldModifier	::=	**public** | **protected** | **private** | **final** | **transient** | **volatile**
MethodDeclaration	::=	$PublicAccessorMethod$ $PrivateInternalMethod$
PublicAccessorMethod	::=	**public** $PrimitiveType$ **get**$Identifier*(FormalParamter)$ $\{MethodBody*\}$
PrivateInternalMethod	::=	**private** $ReturnType*\ Identifier*(FormalParameterList)$ $\{MethodBody*\}$

$*$ The denoted term is defined in the Java Language Specification

Figure 8: *PrimitiveType* and *DataType* definitions

Definition (kernel package)

Kernel Package	::=	$ClassDeclarations$ $InterfaceDeclarations*$
ClassDeclarations	::=	$ClassDeclaration$ $ClassDeclarations\ ClassDeclaration$
ClassDeclaration	::=	$PublicClassDeclaration$ $PrivateClassDeclaration$
PublicClassDeclaration	::=	**public final class** $Identifier*$ $\{PubClassBodyDeclarations_{\text{opt}}\}$
PubClassBodyDeclarations	::=	$PubClassBodyDeclaration$ $PubClassBodyDeclarations\ PubClassBodyDeclaration$
PubClassBodyDeclaration	::=	$ConstructorDeclaration*$ $FieldDeclaration$ $KCMethodDeclaration$
KCMethodDeclaration	::=	$PrivateKCMethodDeclaration$ $PublicKCMethodDeclaration$
PublicKCMethodDeclaration	::=	**public** $MethodModifiers_{\text{opt}}*\ PrimitiveType\ Identifier*$ $(FormalParameterList_{\text{opt}})\ \{MethodBody*\}$
PrivateKCMethodDeclaration	::=	**private** $MethodModifiers_{\text{opt}}*\ ResultTyep*\ Identifier*$ $(FormalParameterList_{\text{opt}}*)\ \{MethodBody*\}$
PrivateClassDeclaration	::=	**class** $Identifier*$ $\{ClassBodyDeclarations_{\text{opt}}*\}$

$*$ The denoted term is defined in the Java Language Specification

Figure 9: java.kernel specification

4.1.1 Review on java.io

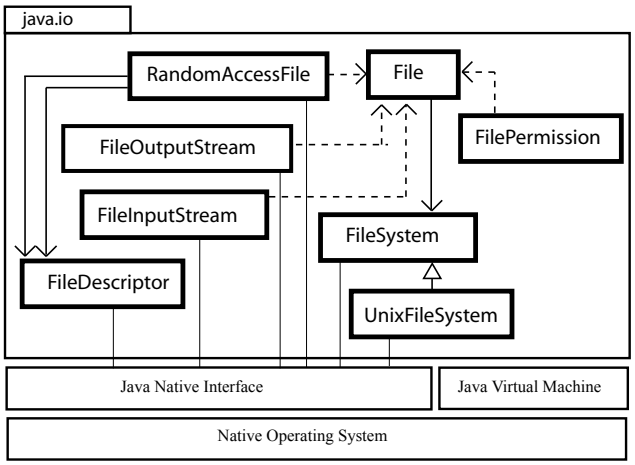

Figure 10: java.io class diagram

Fig. 10 includes the major classes of the J2SDK `java.io` package. It also illustrates dependence of these classes on native code, drawn as thin 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 `FileOutStream` are for raw bytes read and write respectively. Each of the three classes has its own set of native methods performing I/O operations such as open/read/close. To enforce access control, the Java Security Manager is called in to perform stack inspection, for example, when opening a file.

`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 a `FileSystem` object. The `FilePermission` represents the access to a file or directory, consisting of a pathname and a set of actions valid for that pathname.

4.1.2 The library.io package for the μKVM

The `library.io` package wraps the basic I/O operations the kernel offers and provides programmers a high-level view of those operations via classes such as `RandomAccessFile`. Fig. 11 shows the class structure of this package. Notice that the `library.io` package is pure java code without any dependence on the Java Native Interfaces, by contrast with `java.io`. The bond between the kernel and the native code is also minimum.

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

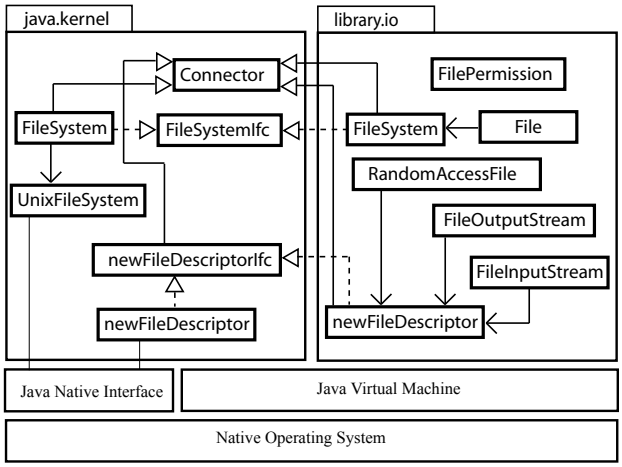

Figure 11: library.io for the μKVM

In `library.io`, `RandomAccessFile`, `FileInputStream` and `FileOutputStream` differ from their `java.io` counterparts in that they no longer have their own native I/O methods. Instead, they import those operations from the kernel's `newFileDescriptor` connector. As the result, real I/O operations are delegated to the kernel via connections on this connector. Creating a `library.io.newFileDescriptor` instance triggers the process of requesting such a connection with the kernel. If granted, the connection is associated with a specific file by the kernel. Thereafter, the connection may be used as a valid channel for operating on that file.

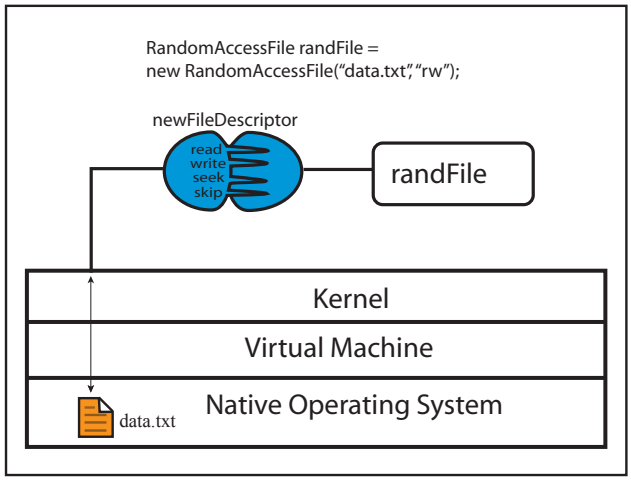

Figure 12: A RandomAccessFile example

Fig. 12 demonstrates a simple example. The statement `new RandomAccessFile ("data.txt","rw")` creates a new `RandomAccessFile` object. When the object is instantiated, the `RandomAccessFile` constructor demands the creation of a `library.io.newFileDescriptor`. The constructor for creating the `library.io.newFileDescriptor` object in turn consults the kernel `President` to get a connection on the

kernel `newFileDescriptor` connector associated with the file `"data.txt"`. If the kernel grants the connect request, then the application gets a valid `RandomAccessFile` object holding a persistent connection to the file.

The `library.io.File` class abstracts file and directory pathnames as it does in the standard `jave.io` package. But, it has no access to privileged data. For instance, one specific difference between `library.io.File` and `java.io.File` is the implementation of a method named `createTempFile`. The method in `java.io.File` uses `doPrivileged` to get the security-sensitive `java.io.tmpdir` property and creates a temporary file under the directory specified by this property. In contrast, the `library.io.File` delegates the creation of temporary files to the kernel via a connection to `FileSystem`. It doesn't need to query the protected `"java.io.tmpdir"` property, which is good because it doesn't have the privileges to do so. Similarly, the `library.io.FilePermission` relinquishes privileges the `java.io.FilePermission` has.

4.2 Net Library Implementation

The `library.net` package provides the same functionality as the J2SDK `java.net` package. The important classes of `library.net` are shown in Fig. 13.

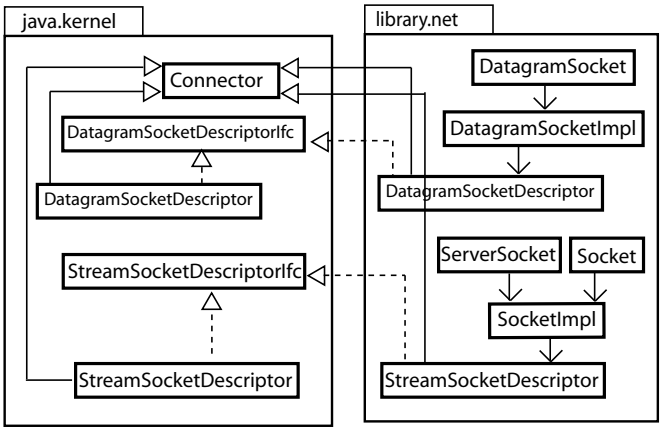

Figure 13: ibrary.net class diagram

The kernel and `library.net` each define two connectors, `StreamSocketDescriptor` and `DatagramSocketDescriptor`, for stream-based and datagram network communication, respectively. Platform-dependent network operations such as `getLocalHostName`, originally performed by native methods of `java.net`, are made part of the kernel public services in the μKVM.

DataType classes, `NetworkInterface` and `DatagramPacket` discussed in Sec. 3.4 are used by the `library.net`.

5. THE SECURITY ARCHITECTURE

The μKVM prototype reuses the Java protection domain framework since the resources being protected and the principals are the same. A Java protection domain conceptually encloses a set of classes whose instances are granted the same set of permissions. A security policy declares the allowable permissions of a protection domain. The Java application environment maintains a mapping from classes and instances to their protection domains and then to their permissions [10]. For policy enforcement, we gut the Java Security Manager and replace it with our own mechanism: Java stack inspection is disabled in our re-implementation of `checkPermission`, and `checkPermission` is invoked solely by the kernel at the start of connector/service invocation and checks that the immediately invoking code has the needed file/network/... permission.

We now informally justify why this architecture is a sound replacement for the security mechanism of the J2SDK.

Object reference sharing breaks component isolation, so confining references across the user-kernel boundary is especially important in the μKVM. Applications can only obtain two types of references into the kernel space: references to the `President` object, and references to kernel connectors.

In the `java.kernel` package, the only classes designed to be used by applications, for instance the simple *DataType* classes, are public. None of the classes for public use have static fields. Therefore, there is no route whereby reference to internal kernel objects can leak out, and no one outside the kernel package has access to the kernel's internal classes. User applications can thus only interact with the kernel on designed interfaces.

Every public class of the kernel has to be carefully examined to prevent undesirable object references from leaking out of the kernel accidentally. As presented in Sec. 3, only primitive types of data can be involved in inter-component interactions via the syntactic restriction enforced on service and connector class interfaces.

The kernel can also hold direct references to application spaces, but these references are limited to the user connectors and *PrimitiveType* objects. The connector and service interfaces contain no other type of object references except the two we just mentioned, so no others can be passed into the kernel. Moreover, the `java.kernel` package doesn't import any application classes. This prevents the kernel from invoking user class constructors. Therefore, we eliminate direct callbacks from the kernel to applications. The callback constraint is another fence guarding boundaries between the kernel and user applications.

We carefully map Java security policies to the security model of the μKVM prototype, so that protected resources in Java are also protected in our system. All the resources originally directly accessible to system libraries are now managed by the kernel, and accesses to such resources have to go through either connectors or services with proper permissions.

We reuse the standard Java exception mechanism. Exceptions can traverse arbitrary domains, so a kernel exception may be propagated to user spaces. In order to protect the integrity of the kernel interface, the kernel only throws exceptions with string messages at most so that exceptions do not become a backdoor communication channel.

In summary, we have shown how the μKVM security model can be faithfully implemented, and having a security mechanism based on a clear-cut kernel-application boundary is powerful, as strong as the J2SDK mechanism, and is fundamentally more declarative in nature since the checks all occur at a clearly declared interface. However, a system is secure only if it can survive attacks in a real world. Our ultimate goal is to put up the μKVM for public use once its implementation approaches completeness, so that its security can be tested in practice by real users.

6. μKVM EVALUATION

6.1 Functionality Evaluation

In order to verify that the user-level functionality of the J2SDK is preserved in the μKVM, we ran a series of test suites. We used the mauve test suite[2] for Java class libraries. No modification to the test suite programs for the μKVM needed to be made besides changing imports of `java.io` and `java.net` to `library.io` and `library.net`, respectively. The mauve test suite contains around 2000 tests for io and net packages. The following Table 1 shows the test results.

The first row in Table 1 shows numbers of failed, succeeded, and total test cases on I/O operations in the J2SDK and μKVM, respectively. The second row gives the test results for network operations. The last row summarizes across all test cases. This data shows that our implementation meets the mauve specification as well as the J2SDK implementation does[3]. Based on the tests, we can conclude that our prototype system indeed preserves the functionalities of the Java platform.

6.2 Performance Evaluation

Microkernel operating systems historically suffer runtime performance problems, which makes microkernel-based architectures potentially less appealing for practical systems. To find out whether the μKVM also has such a performance pitfall, we evaluated its performance via a set of performance benchmarks[4]. We had to design the benchmarks ourselves because we could find no existing benchmarks to test the speed of the I/O; on the mauve benchmarks above, the running time difference between the J2SDK and μKVM was insignificant because the applications did not do significant amounts of I/O. The benchmark tests of this section were run on the Sun platform, and security was switched on only for the cases where the security overhead was being profiled, noted below.

Test programs are run in identical settings in both the J2SDK and the μKVM, therefore, it is the relative performances really matter instead of absolute numbers. Comparisons between the J2SDK benchmarch results and the μKVM ones are reflected in percentages of their differences (Diff), which are calculated with the formula: Diff = (μKVM - J2SDK)/J2SDK * 100%. The Diff values are used in tables throughout this section.

6.2.1 File Open Tests

The file open tests aim to measure the time and memory consumption in file open operations. The test program simply opens 500 or 1000 files and records the total time spent on opening operations. The memory overhead is profiled by using the `-Xrunhprof:heaps=sites` option of the Java interpreter, which outputs the detailed information of heap usage. Opening a large amount of files is not a typical scenario in real applications. But we choose to benchmark it because it is the place that the μKVM mostly likely introduces overhead.

We executed the tests on a Sun Blade 50 running Solaris 2.5 with 650MHz processor and 256Mb memory. Table 2 shows the benchmark results.

From Table 2 we can see that for file opening the μKVM introduces a very small overhead. For example, the μKVM slowed down by 2.98% in opening 500 files and used 3.03% more memory. Part of this overhead came from library loading. In the μKVM, the io package is no longer system library and is not preloaded into the runtime as it is in the J2SDK. Therefore, the test program in the μKVM had to spend extra time in loading library classes. The memory overhead is due to mapping connections into delegations between two java connector objects, which causes more objects to be created. In any case, the overhead is reasonably small that would barely impact the performance of a real application.

6.2.2 File Read and Write Tests

Read/write tests are used to benchmach the kernel implementation because file I/O in the μKVM is managed solely by the kernel. In this group of tests, 5M data were read/written from/to a file.

As shown in Table 3, different block sizes were used to read/write the 5M data. When the block size was 64 bytes, the μKVM ran faster by 0.35% in read and 0.97% in write. In the 128 case, the J2SDK perfromed better by 0.71% and 0.01% in read and write respectively. Block size of 256, 512 and 1024 were also tested, and all results indicate that the difference of the μKVM and the J2SDK on file read/write is not statistically significant.

6.2.3 Network Benchmark

A typical TCP/IP network application, consisting of a client and a server, is used to assess network performance of the μKVM prototype. The client and the server were set up in a Local Area Network, running on Solaris 2.6 with 350MHz processor and Solaris 2.8 with 650MHz processor respectively, connected by a 100Mbps LAN. The client sent total of 1M data to the server and recorded the time elapses. We benchmarked 9 cases with different network packet sizes, ranging from 64 to 16384 bytes.

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

[3] The difference in the total number of tests performed on the J2SDK and our prototype implementation results from how the tests were grouped: the test suite groups tests together so that one test failure may cause skipping of the rest of tests in the same group. All the failed tests on our packages and those on peer Java packages are either the same or simply different on exception types. For example, writing to a closed socket in Java throws an `IOException`, while in our model it throws a null pointer exception because the connection associated with that socket has been disconnected and becomes null.

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

	J2SDK			μKVM		
	Failed	Succeeded	Total	Failed	Succeeded	Total
io tests	9	648	657	9	648	657
networking tests	9	365	374	8	376	384
total	18	1013	1031	17	1024	1041

Table 1: Results of mauve tests

File Number	File Open Time (ms)			Memory (byte)		
	J2SDK	μKVM	Dif(%)	J2SDK	μKVM	Diff(%)
500	395	407	2.98	2,385,752	2,458,072	3.03
1000	847	875	3.33	2,408,112	2,496,888	3.69

Table 2: File Open Benchmark

Table 4 shows our μKVM implementation runs faster than the J2SDK up to 10%, even though related code in the two platforms is technically identical. In-depth examination of the cpu profiles generated on the two systems revealed that Java spent more time synchronizing the field "fdUseCount" in its `PlainSocketImpl` class. This field is used for recording the number of threads using the corresponding socket and is updated synchronizely every time a write/read takes place on the socket. To find out the impact of such synchronization, we conducted another group of tests with the same setting as ones used in Table 4 but without synchronization on "fdUseCount". The results show that the μKVM and the J2SDK have comparable performance.

Memory overhead in the network benchmark was obtained via java option `-Xrunhprof:heaps=sizes`. Table 4 demonstrates that the μKVM only has slight overhead over the J2SDK, i.e. the difference is maximally 0.58%.

6.3 Security Evaluation

The benchmarks above focused on raw performance and therefore excluded the additional overhead incurred from access control checks. In this section we assess the J2SDK vs. μKVM security overhead. Note that we are only evaluating the performance of the checks, not other dimensions of the security model; that topic is addressed in Section 5.

Benchmark programs used in this section are the same as the ones in Sec. 6.2.1 but with security enabled. For Java, test programs were run with a Java Security Manager, while for the μKVM, a security mode was set and the kernel enforced access control on its interfaces. Test results then were compared with the results of the previous section.

Table 5 shows the overhead resulting from running the security architecture on the J2SDK and the μKVM respectively. On the J2SDK, performance suffered greatly. The speed for opening 500 or 1000 files slowed down by 136.45% and 77.33% respectively. Security also introduced overhead to the μKVM, but much less severe than it did to Java, i.e. 68.5% and 39.89% in the same two test cases.

Memory comsumption increased on both the J2SDK or the μKVM in the security overhead benchmarks, shown in Table 6. When 500 files were opened, Java had 24.44% memory overhead while the μKVM had 10.79%. In the 1000 case, Java experienced 43.28% memory increase while μKVM got 35.95%.

Table 7 is the head-to-head comparison between the security overheads of the J2SDK and the μKVM. It clearly shows the μKVM outperformed the J2SDK in both speed and memory. Precisely, the μKVM was 26.66% faster than Java when 500 files were opened and 18.51% faster when the file number increased to 1000. As for memory comsumption, the μKVM costed only 0.01% more memory in the 500 case while 1.63% less in the 1000 case.

We have shown in this section that the μKVM not only runs as fast as the J2SDK in file and network operations, but also has significant effiency advantages in providing a secure runtime environment. So far we have not yet done any optimization on the prototype implementation. We believe that our μKVM can be further toned to achieve better performance, with optimizations such as implementing parts of the kernel in native code. Based on the performance of the μKVM in these benchmarks, we are optimistic that good performance will be obtainable for a full implementation including GUI libraries, etc.

7. FUTURE WORK

We are in the process of completing the μKVM design of Fig. 4, in particular the GUI component. With a full GUI capability, we will be able to develop and test real-world applications on the μKVM. Fig. 14 shows a more detailed design for GUI. The kernel and applications execute in their own threads, and the system-wide event dispatch thread used in the J2SDK is replaced with application-level threads. Every application has its own event queue and dispatch thread. An application registers its interests to the kernel via its plugin `Register` and the kernel delivers proper events to the application via `PostEvent`. All the applications interested in the same event always get their own copies of that event. 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 GraphicConnector, instead of operating on them directly using JNI. This

BlockSize (byte)	Read (ms)			Write (ms)		
	J2SDK	μKVM	Diff(%)	J2SDK	μKVM	Diff(%)
64	877.5	874.4	-0.35	1,136.3	1,125.3	-0.97
128	466.1	469.4	0.71	688.1	688.2	0.01
256	264.8	266.0	0.45	488.8	484.5	-0.88
512	158.5	159.2	0.44	322.9	318.6	-1.33
1024	104.4	103.9	-0.48	234.0	230.7	-1.41

Table 3: File Read/Write Benchmark

Message Size*Num	Memory (byte)			Transfer Time (ms)		
	J2SDK	μKVM	Diff(%)	J2SDK	μKVM	Diff(%)
64*16384	10,777,328	10,789,736	0.12	3.33	3.10	-6.83
128*8192	9,723,656	9,735,328	0.12	5.60	5.01	-10.48
256*4096	9,213,296	9,220,456	0.08	10.73	9.59	-10.62
512*2048	8,957,792	8,980,544	0.25	22.48	20.18	-10.24
1024*1024	8,803,504	8,853,080	0.56	43.41	39.77	-8.39
2048*512	8,741,040	8,792,040	0.58	86.60	80.43	-7.12
4096*256	8,714,416	8,763,912	0.57	170.82	166.53	-2.51
8192*128	8,722,704	8,753,064	0.35	345.10	316.57	-8.27
16384*64	8,742,816	8,754,976	0.14	631.86	648.39	2.62

Table 4: Network Benchmark

File Number	J2SDK File Open Time (ms)			μKVM File Open Time (ms)		
	-Security	+Security	Diff(%)	-Security	+Security	Diff(%)
500	395	934	136.45	407	686	68.55
1000	847	1,502	77.33	875	1,224	39.89

Table 5: Security Overhead on File Open

File Number	J2SDK Memory (byte)			μKVM Memory (byte)		
	-Security	+Security	Diff(%)	-Security	+Security	Diff(%)
500	2,385,752	2,968,824	24.44	2,458,072	2,969,224	20.79
1000	2,408,112	3,450,360	43.28	2,496,888	3,394,200	35.94

Table 6: Security Overhead on Memory

File Number	File Open Time (ms)			Memory (byte)		
	J2SDK	μKVM	Diff(%)	J2SDK	μKVM	Diff(%)
500	934	686	-26.66	2,968,825	2,969,224	0.01
1000	1,502	1,224	-18.51	3,450,360	3,394,200	-1.63

Table 7: Security Overhead Comparison

architecture effectively removes privileges from the original Java AWT/Swing libraries and puts the kernel in full control of GUI resources.

As to efficiency, a major concern of any GUI implementation, we are confident that our model is able to achieve performance comparable with Java, based on the benchmarks of the file and network components in Section 6.

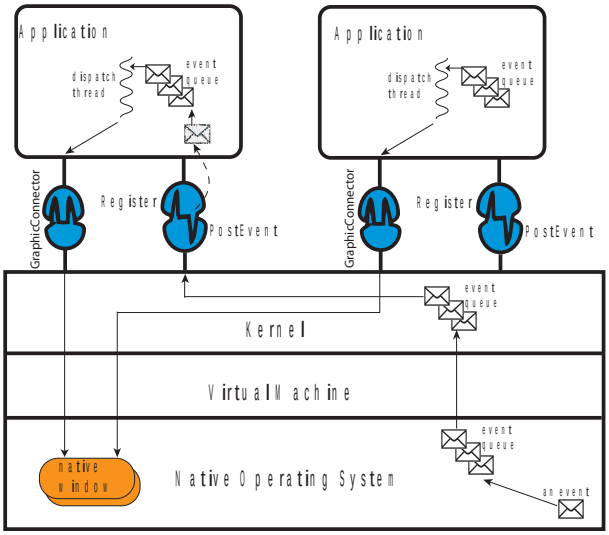

Figure 14: GUI Implementation Design

A full-fledged μKVM with the capability of running multiple user applications is one of our major future tasks. In the multi-application μKVM, every application is an isolated computation, and they interact with each other via interfaces as shown in Fig. 15, the same way as an application communicates with the kernel. Unlike the single user application model, resources are shared among applications in the multi-application μKVM. Multiplexing system resources properly is a key issue we have to address in order to guarantee application isolation.

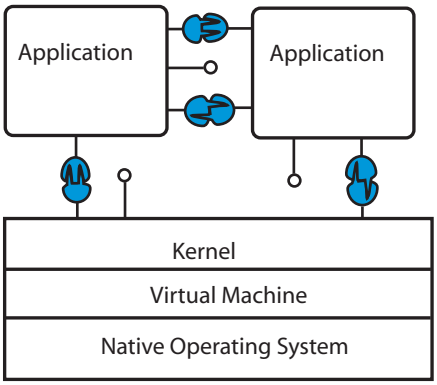

Figure 15: Multiple applications

To run multiple applications, a class loading mechanism must enforce application isolation. Classes shared by multiple applications need to be loaded in a way that they do not become a backdoor channel for inter-domain communication. Meanwhile, sharing code safely cross applications is essential to achieve efficiency and a smaller memory footprint. We would need to design and implement such class loading model because the Java Virtual Machine is essentially a single application platform whose class loaders do not have built-in capability for isolating multiple applications.

Isolated VM component domains have also been pursued in several other projects, so this is not as novel a topic as is the application-kernel dimension, and that is why this paper focuses on application-kernel and not application-application interaction. The Application Domains of the Microsoft CLR [22] guarantee isolation of multiple applications running in the CLR. Java Isolates [17] and related research [15, 2, 4, 8] are proposals to extend the JDK with an application isolation mechanism. Java Isolates concentrates on removing the potential for sharing via class static fields. The J-kernel [14] implements application isolation on top of an existing JVM and it focuses on using stub objects as capabilities for cross-domain communication. All of these systems use a copy-only communication mechanism for inter-component communication. Other approaches include the Multitasking Virtual Machine (MVM) [5, 6] and KaffeOS[1], which support primitive inter-application communication via sockets or a shared heap. The big difference between the μKVM and these other projects is how it also includes a specific domain for the key system services, the *kernel*.

8. CONCLUSION

In this paper, we proposed a novel microkernel-based language VM, the μKVM. In our architecture, the kernel manages system resources and implements a core set of low-level system functionalities. System libraries such as I/O library are implemented as a layer on top of the kernel, and run in user mode. All interactions between the kernel and applications are declared on explicit interfaces, the connectors and services.

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 μKVM does not need stack inspection to protect system libraries because those libraries have no more privileges than user applications. Because of this, programmers are free to replace system libraries with specialized libraries that better suit their needs, allowing for open extensions that will not introduce additional risk into the system.

The security architecture of the μKVM is simple and declarative. Accesses to all sensitive resources are through publicly declared interfaces, the 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, and implementing or modifying such a security model is not as demanding as the J2SDK security architecture.

The Principle of Least Privilege can be easily deployed in the μKVM via imposing fine-grained security schemas on interfaces. It is easy to see the permissions an application needs to execute successfully by checking the connectors/services it declares, and hence avoid granting extra permissions to it. At runtime, the kernel has the ability to revoke an already granted access to a system resource instantly, by disconnecting the connection associated with the resource. The J2SDK security model lacks such a revocation feature—an application can hold on to a previously obtained file descriptor even after such access is no longer allowed in a dynamically changed security policy.

Benchmarks of the μKVM prototype show that our model can provide a secure runtime environment with less overhead penalty than the Java standard platform, avoiding the runtime performance pitfall commonly found in microkernel-based operating systems.

9. REFERENCES

[1] 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.

[2] D. Balfanz and L. Gong. Experience with secure multi-processing in java. In *Proceeding of ICDCS'98*, Amsterdam, The Netherlands, May 1998.

[3] 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`.

[4] G. Czajkowski. Application isolation in the java virtual machine. In *Proceedings of the 15th ACM SIGPLAN conference on OOPSLA 2000*, 2000.

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

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

[7] S. L. 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`.

[8] D. Dillenberge, R. Bordawekar, C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. S. John. Building a java virtual machine for server applications: The jvm on os/290. *IBM Systems Journal (Java Performance issue)*, 39, 2000.

[9] 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.

[10] 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`.

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

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Boston, Mass., 2000.

[13] 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`.

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

[15] 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.

[16] C. Horstmann and G. Cornell. *Core Java Volume II: Advanced Features*. Sun Microsystems Press, Upper Saddle River, NJ, 2001.

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

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

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

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

[21] Y. Liu and S. Smith. Modules with interfaces for dyanmic linking and communication. In *ECOOP 2004*, June 2004. `http://www.cs.jhu.edu/~scott/pll/assemblage/ecoop04.pdf`.

[22] E. Meijer and J. Gough. A technical overview of the common language infrastructure, 2001.

[23] 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.

[24] 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.

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

[26] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[27] A. O. T. Higuchi. A static type system for jvm access control. In *ICFP'03*, Uppsala, Sweden, August 2003.

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