# The SKVM: A Secure Version of the KVM

William R. Bush, *Sun Microsystems Laboratories*

Antony Ng, *D'Crypt Pte. Ltd.*

Doug Simon, *Sun Microsystems Laboratories*

Version 1.0, 24 April 2002

## 1. Design Goals

This document describes the SKVM, a virtual machine architecture that satisfies the following primary goals:

1. The SKVM must provide a high level of security assurance when correctly implemented and used. It must be suitable for high-security applications solving security-specific problems.
2. The SKVM must be suitable for running on small, resource-constrained devices such as personalized cryptographic modules, next-generation smart cards, cellular phones, and PDAs.
3. The SKVM should be fully compliant with the Java™ 2 Platform, Micro Edition Connected Limited Device Configuration (CLDC) specification [CLDC], which defines a standard Java platform for small devices. Specifically,
    a. CLDC/KVM applications should not be able to distinguish the KVM from the SKVM on the basis of observed behavior.
    b. Correctly implemented secure applications operating normally should not be able to distinguish the KVM from the SKVM on the basis of observed behavior.
    c. Only malicious classes should elicit different behavior from the SKVM than they would from the KVM.

There are four other significant design goals.

1. The SKVM should be validated as secure. Implemented on a suitable hardware platform, it will be submitted for FIPS140-2 certification. FIPS140-2 is a specification drawn up by the National Institute of Standards and Technology, defining security requirements for cryptographic modules [FIPS]. In addition to being the stipulated requirement for any cryptographic module acquired by the US government, FIPS140-2 has become a de facto standard for cryptographic equipment and provides a level of assurance that the equipment was designed with adequate consideration of security. The standard spells out requirements in 11 different areas including physical security, hardware security, software security, and key management. A cryptographic module can be certified to any of four increasing levels of assurance.
2. The SKVM should qualify for Java Powered™ branding. To achieve this, the SKVM must pass the proper compatibility tests (the CLDC TCK) [TCK].
3. The SKVM should place minimal functional demands on its host operating system. It must run on simple, small operating systems designed for small devices. This goal is desirable in part because system security is a function of the security of the operating system, and a complex, pre-emptive multitasking operating system such as Windows is much harder to analyze and secure than a minimal operating system.
4. The SKVM should support soft real-time applications. Example applications include voice coding, soft modems, and short range RF communications such as Bluetooth. Such applications exhibit interrupt frequencies in the tens of milliseconds.

# 2. Assumptions

A device is only secure if all its components are. At some point the security of each component must be examined. In the limit, one could question whether the CPU executes the instructions it is fetching, or whether the memory has additional circuitry that modifies stored data maliciously. We assume here that the basic computing hardware can be trusted. We also assume that a secure operating system is available, controlling access to underlying hardware resources.

Any security device must have the ability to store secrets in a secure way. This typically means that there exists some specially identified region of non-volatile memory in which secret data can be stored. Such memory is usually protected against a variety of attacks, both physical and electronic. We assume that this memory exists, with hardware features that control access to it.

We assume that good cryptographic algorithms will be used for message digest (hashing), and digital signatures. Some examples of good cryptographic algorithms include MD5 for message digest, and RSA for digital signatures. Of course, any cryptographic algorithm can be broken given enough time and resources. However, it is not the objective of the SKVM architecture to protect against potential cryptographic algorithm weaknesses.

We also assume that the hardware will have the resources necessary to support the cryptographic algorithms. Such resources extend beyond conventional computational ability to include a source of random bits. While such resources are critical to the security of a good cryptographic token, they are beyond the scope of the SKVM architecture.

We also assume that the device will be physically secure. Such physical security can be implemented through mechanical means like tamper proofing, or through an integrated ability to detect and respond to tamper events. An example of a response to tamper would be the near-instant erasure of all secret information.

A cryptographic module (also referred to as a cryptographic token or a hardened token) is an example of a device with the above characteristics. Such modules store secrets, execute cryptograms, and interact in well-defined ways with the external environment. A correctly designed protocol working in concert with a cryptographic module can ensure that a security transaction is completed without secrets leaving the module; only information derived from secrets leaves the module. An example device has been built and described by IBM [IBM, 4758].

# 3. Key Precepts

A number of key precepts guided the design of the SKVM architecture [see S&L, AC, SE].

## 3.1 Simplicity

The overriding precept is simplicity. It has several important benefits.

- ? It minimizes bugs and possible points of compromise.
- ? It keeps the system's memory footprint small.
- ? It makes the system easier for application developers to understand, which in turn makes it easier for them to implement secure applications.
- ? It makes the system easier to validate for security. Such validation typically involves modeling security state transitions with a state machine (this is the paradigm required for FIPS140-2 certification). A simple system has fewer states and is thus more tractable. For example, a simple operating system without pre-emptive multitasking is far easier to analyze than a complex pre-emptive multitasking one, because an analysis of the latter must consider the multiplicity of states caused by the interaction of the pre-emption mechanism with normal execution.

## 3.2 Fail-Safe Design

When a fail-safe system encounters an unanticipated condition, it always lapses into a conservative, secure state. Such conditions can be genuinely unanticipated or can be a result of a partial malfunction of the system. No matter how comprehensively a system is analyzed, it is unlikely that all possible combinations of conditions have been anticipated. Fail-safe design ensures that the system makes conservative assumptions and lapses into a secure state when an unanticipated condition is encountered.

A watchdog signal in a battery-powered system is an example of fail-safe design. As long as periodic signals arrive from the watchdog, the CPU continues normal operation. If the battery runs low, the periodic watchdog signal is terminated, and the CPU shuts down. Note that if the watchdog signal is interrupted in any other way (for example, due to physical tampering), the CPU also shuts down – it lapses into a secure state.

## 3.3 Static Specification of Security Policy

The ability to modify policy dynamically is usually considered a desirable feature. For a secure device, it is also a major source of weaknesses. Dynamic modification of security policy is almost always a cause of subtle bugs. In contrast, a static security model forces the application developer to consider the security aspects of the software architecture earlier and more completely in the design process. A static model is also in general easier to analyze and vet because it is simpler and usually has fewer states and state transitions than a dynamic model.

## 3.4 Explicit Specification of Security

The combination of static security specification and fail-safe design dictates that security issues – specifically the granting of privileges – involve an explicit act on the part of an object, and that any privilege not explicitly granted is automatically denied. The SKVM implements this precept within the confines of the semantics of the Java programming language (hereinafter referred to as the Language).

As an example of how the Language semantics affects this precept, note that an object has the right to manipulate those parts of it (including protected data) that are inherited from its ancestors. When an object grants privileges to another object, through the Language semantics it automatically and implicitly grants access to all its ancestors in the inheritance tree.

## 3.5 Security at the VM Level

Implementing security policy using only classes is attractive for several reasons, including extensibility and uniformity. However, good security engineering suggests that the core of the security framework be implemented at the VM level. The challenge is to keep the bare minimum in the VM level and leave as much as possible at the language level so that a high level of security assurance can be established without compromising system flexibility.

## 3.6 Reliance on Data Authenticity, not Secrecy

Secrets stored on a device, such as symmetric or private keys, introduce potential vulnerabilities and complicate responses to security compromises. They also complicate the device and increase its cost since the device must now defend the secret against disclosure. In contrast, use of verifiably authentic data, such as certificates verifiable with public keys, does not create such weaknesses.

The SKVM is designed so that security assurance relies only on the ability of the device to keep data authentic. The SKVM does not require the device to keep a secret.

# 4. Platform Overview

The SKVM architecture has been designed with the following two platforms as the starting point.

## 4.1 The d'Cryptor™ PE Cryptographic Token

The d'Cryptor? PE cryptographic token is a high security assurance, commercially available cryptographic module, and is the first intended target platform for the SKVM. It is illustrative of the platforms on which the SKVM is designed to run. It was developed by D'Crypt Pte Ltd.

The d'Cryptor PE is a Type II PCMCIA card built around a programmable cryptographic core composed of a 206 MHz Intel StrongARM SA-1110 processor with 4MB of Flash ROM and 512KB of SRAM. A real-time clock, a hardware noise-source, and battery backup RAM provide necessary cryptographic support, while a PCMCIA interface, two UARTs, a USB slave interface, and other ports allow the d'Cryptor PE to interface with a variety of peripherals. The device includes a range of features that actively sense tamper-events. These sensing circuits are powered by an internal rechargeable battery and erase the battery backup RAM on detection of tamper.

The d'Cryptor PE runs a small, secure micro-operating system. The ?OS supports cooperative multitasking with up to 16 separate code-modules. The ?OS employs memory management techniques to ensure that modules remain memory-disjoint. Modules communicate either through inter-module procedure calls with pass-by-value arguments, or through explicitly declared shared memory regions.

More information on the d'Cryptor PE can be found at http://www.d-crypt.com/products/dcryptor.html.

## 4.2 The K Virtual Machine and the CLDC

The Java 2 Platform, Micro Edition (J2ME™ technology) spans a broad array of consumer and embedded electronics. Two basic J2ME configurations have been defined, one for devices that are mobile, and one for devices that typically are fixed. The hardware and network resources available to mobile devices tend to be more limited than those available to stationary devices with an ample supply of power. These configurations consist of core library sets and virtual machines optimized for characteristics typical of small devices.

The configuration for smaller, handheld devices is the J2ME Connected Limited Device Configuration (CLDC). It outlines the most basic set of libraries and Java virtual machine features that must be present in each implementation of J2ME. To form a complete environment for any given device, implementers add additional libraries that address API areas not dealt with in the low-level CLDC. Two-way pagers, mobile phones, personal digital assistants and point of sale terminals are some, but not all, of the devices that are supported by this configuration. As mentioned above, compliance with the CLDC specification is demonstrated by passing the CLDC Technology Compatibility Kit tests.

The heart of the configuration is the K virtual machine (KVM). The KVM is a new virtual machine designed with the constraints of inexpensive mobile devices in mind. Named to indicate that its size is measured in tens of kilobytes, the KVM is suitable for devices with:

? Total memory of between 128K and 512K, with up to 256K of ROM or flash memory and up to 256K of RAM; in most cases devices will have more ROM or flash memory than RAM;

? Limited power, often battery operated;

? Connectivity to some type of network, with possibly limited bandwidth (9600 bits per second or less); and

? User interfaces with varying degrees of sophistication, down to and including none.

As a result of the design goal to minimize footprint, the KVM is very simple. This simplicity makes the KVM easy to understand and modify, important characteristics in the context of security.

More information on the KVM can be found at http://java.sun.com/products/cldc.

# 5. Security Issues Addressed by Current Specifications

Current Java specifications deal with two primary security issues, the internal integrity of programs, and the external integrity of interactions between programs and the outside world.

## 5.1 Verification of Class Files

An important aspect of security is the verification by the VM of the well formedness of the classfiles loaded into the virtual machine for execution. This verification process has four phases [JVMS, section 4.9].

*Phase 1*: Check the basic integrity of the structure of the classfile.

*Phase 2*: Check the classes and references in the classfile (final classes are not subclassed, final methods are not overridden; every class but Object has a direct superclass; the constant pool meets static constraints; all field and method references in the constant pool have valid names, classes, and type descriptors).

*Phase 3*: Verify that the code in the classfile is safe. This is known as bytecode verification.

*Phase 4*: Perform run time type and method checks on first reference.

## 5.2 Bytecode Verification

The Language is strongly typed. To save time and space, the type information is encoded in the bytecodes, rather than in the objects they manipulate (as is the case, for example, in Smalltalk). In other words, the bytecodes specify the types of the objects they reference; the bytecode to load an integer, for example is `iload`, and the bytecode to store an object reference is `astore`. Not all bytecodes are typed, however (the bytecodes `dup` and `dup2` duplicate the top of the stack regardless of its type), and so the bytecodes themselves are not enough to guarantee type safety. Additional analysis is required. Furthermore, arbitrary sequences of bytecodes can be constructed independent of a compiler, expressing semantics not allowed in the Language.

The specific consequence of this is that, since the evaluation stack is not typed, there is no guarantee that an arbitrary sequence of bytecodes will manage the evaluation stack in a type-safe manner.

The solution is to perform an analysis of the bytecodes before they are executed [JLS, section 12.3.1][JVMS, section 4.9.2]. Before execution they are subjected to a verification step that checks for type-consistency and safety. The component that performs this check is known as the verifier.

## 5.3 Bytecode Verification and the CLDC

The standard verifier is a relatively large and complex program, unsuitable for deployment on small devices. As a result, the verification process mandated in the CLDC is slightly different [CLDC, section 5.2]. It is a two-stage process.

The first stage, based on the standard verifier and performed on a desktop or other large platform, analyzes the bytecodes for type consistency and generates type maps for the execution stack at various points in the code. This stage is called the preverifier.

The second stage runs on the target device. It makes a single pass over the bytecodes, emulating the operations of the bytecodes it encounters and verifying the resulting stack manipulations against the type maps for type safety. This second stage, also called the verifier, will not allow a classfile to be executed if it detects any inconsistency between the bytecodes and type-maps.

## 5.4 Bytecode Verification and the SKVM

The SKVM performs all four phases of classfile verification.

Bytecode verification is the most challenging. Bytecode integrity is clearly necessary for a secure VM, especially if classfiles are to be accepted and executed from potentially malicious sources. The SKVM is CLDC compliant and performs CLDC-style bytecode verification.

However, for the SKVM to obtain FIPS140-2 certification at a high security assurance level, relying on the current CLDC bytecode verification process, it will have to be formally proven that any unsafe bytecode sequence will be detected by the verifier, and the verifier implementation will have to be validated as equivalent to its formal specification. At this point there are no plans to perform these difficult tasks.

In lieu of such verifier validation, the SKVM must implement a typed evaluation stack or equivalent mechanism suitable for validation, so that bytecode integrity can be assured.

## 5.5 Implementation Optimizations

It may prove necessary to improve the performance of the SKVM. Two well-known techniques for improving performance are fast bytecode transformations and just-in-time compilation. These techniques dynamically replace or supplant bytecodes in a classfile. Obviously, any such modifications must be carefully analyzed for security weaknesses. This analysis will be particularly difficult for FIPS certification, because all possible changes in security state introduced by the optimizations will have to be enumerated and analyzed.

## 5.6 The CLDC Security Model

The CLDC security model [CLDC section 3.4] defines three types of security, low-level VM security, application-level security, and end-to-end security.

Low-level VM security is defined as the characteristic that "an application must not be able to harm the device in which it is running, or crash the virtual machine itself." In the context of the KVM this means that CLDC verification must be done on all classfiles.

End-to-end security refers to network-based solution-oriented security, which is outside the scope of the CLDC specification.

Application-level security is defined as controlling access to external resources, which is done on the larger J2SE™ platform by the security manager. The security manager was deemed to have too large a memory footprint for the CLDC, so a sandbox model is used instead. Specifically:

? Only a limited set of APIs is available (the CLDC libraries, profiles, and manufacturer-specific classes).

? Such system classes cannot be overridden.

? No user-defined class loaders area allowed.

? No native functions can be dynamically loaded onto the device.

? The class lookup order may not be manipulated.

? By default, an application may only load classes from its own JAR file.

In addition, a CLDC implementation need not support multiple concurrent applications.

The CLDC security model is a good starting point for the SKVM. It is small, simple, and relatively static, which is good both for small devices and for increased security. It has a static set of APIs, system classes, and native functions, and a single system class loader. It has a simple application model. It is possible to be compliant with the CLDC and provide greater security than the CLDC mandates.

# 6. The SKVM Security Architecture

This and following sections present the components of the SKVM security architecture:

? The notion of trust;

? The implementation of trust;

? The characteristics of other necessary VM features; and

? The application model.

## 6.1 Owners, Trust Relationships, and the Trust Community

In a well-designed, secure, closed system every class, interface, and package – every component – has an owner, an entity (nominally a human) with ultimate responsibility for it.

Systems are often assembled from components developed by different owners. One owner may or may not trust another, based on their relationship. Owners that do trust one another form a community, however informally.

In addition, in a connected, Java-enabled world, components may be loaded dynamically that have no verifiable owner, and that may be malicious. They cannot be trusted.

These basic observations are the foundation of the SKVM security architecture. It implements these notions of *ownership* and *trust* in the context of the CLDC.

## 6.2 Trusted and Untrusted Classes

The SKVM supports trust relationships and a trust community by providing the framework and features necessary to request and grant privileges. The SKVM itself maintains no explicit information on trust relationships and trust communities other than what each class brings in.

Since trust communities are embedded in applications together with untrusted and potentially malicious classes, the SKVM provides the means to express which parts of an application are trusted, and to what degree, and which parts are not. This is done by dividing all classes into two groups, trusted and untrusted, and by creating a privilege hierarchy among trusted classes.

Intuitively, a trusted class is provided for some trusted community of owners. Trusted classes implement those parts of an application that have to be secured. They are also the means by which sensitive information is encapsulated. They can control or deny access by untrusted classes to such information. Trusted classes also have privileges and can in turn grant privileges to other trusted classes.

As mentioned earlier, a trusted class has an owner. It is the responsibility of this owner to request and obtain the necessary privileges for the trusted class. If class X needs a particular privilege from class Y, the owner of class X will have to acquire this privilege from the owner of the class Y. These privileges come in the form of certificates signed by Y's owner and held by class X. They are verified by the SKVM when class X is loaded. In fact, the difference between a trusted class and an untrusted class is precisely this: a trusted class will carry with it certificates that prove that it has certain privileges, while an untrusted class has no such certificates.

Untrusted classes may also be loaded into the SKVM. Such classes can provide useful, CLDC-standard functionality. Untrusted classes are kept in a strict sandbox. However, trusted classes can grant various privileges to untrusted classes. The degree to which trusted classes are prepared to grant privileges to untrusted classes defines the extent of the sandbox ? the sandbox is not fixed or defined *a priori*.

## 6.3 Trusted Classes and the Privilege Hierarchy

The SKVM is designed to support multiple application components that may be mutually distrustful. Furthermore, while all classes in an application may manage security properly, the architecture should be fail-safe and contain any region of compromise.

Both these observations imply a need for varying degrees of trust, even among trusted components. The SKVM supports a fine-grained security model by controlling two types of access.

? The more powerful privilege is the power to subclass, to install a class as a subclass of another. In some sense this privilege is the privilege to modify, in a controlled way, the code on the device. It is referred to as the **S** privilege. Subclassing also includes the power to access the protected fields, methods, and constructors of all the superclasses of the subclassing class, regardless of package.

? The less powerful privilege is the power to create a new instance of an object and reference its static methods and fields. This power is in essence the power to create a capability. Good object-oriented and secure programming practice mandates the strict factoring and encapsulation of data, which, combined with the controlled creation of objects, enables capability-based design. This privilege is referred to as the **A** ("access")

privilege. There are two implicit trusted instantiation privileges granted for a given class X. The first is (trivially) granted to X itself and the second is granted to the superclasses of X.

Accesses to static and member methods and fields are controlled through the Language's private, protected, and public tagging [JLS, section 6.6]. Note that the Java Language Specification stipulates that accesses have to be checked at run time [JVMS, sections 5.4.3 and 5.4.4] Information about private, protected, and public access permissions are maintained in the constant pool to support such run time checking.

The S privilege enables a precise definition of a trust community. Since all classes except `Object` subclass some other class, the class hierarchy tree rooted at `Object` can be thought of as defining a trust relationship that is transitively closed downwards. More precisely, a *trust community* is defined recursively as follows:

- `Object` is a member of the trust community, and
- Any class that has requested and obtained the privilege to subclass a class in the trust community is itself a member of the trust community.

Thus `Object` has a trust relationship; either directly or indirectly, with all trusted classes.

The ability to subclass or to create an object implies the ability to subclass or create any parent of the object *as part of the act of subclassing or creating the object.* These are standard Language rules. Note however that the ability to subclass or instantiate an object does not imply the ability to subclass or instantiate any parent of the object in the class hierarchy independently. The operation (subclassing or instantiation) on the parent of the class in question can only happen as a direct and automatic result of the same operation on the class itself. For example, if class **X** has permission to instantiate class **B,** which subclasses **A,** then it does not necessarily follow that X could directly instantiate **A**. To do so requires that **X** have explicit permission for instantiation from **A**.

## 6.4 Untrusted Classes and Privileges

Untrusted classes rely on trusted classes for all their privileges. Since untrusted classes have no certificates, from a security standpoint they are indistinguishable from one another. Privileges are granted uniformly to all untrusted classes. These privileges take three forms:

- In the capability-based style discussed above, and similar to a trusted class, an untrusted class may be allowed to subclass a trusted class. Since all untrusted classes are equivalent in privilege, this power when granted is given to all untrusted classes. Like all other privileges granted to untrusted classes, this is a privilege that the trusted class in question must grant explicitly.
- Analogously, an untrusted class may be granted the privilege to create a new instance of a trusted class. Again, this is a privilege that is explicitly granted by a trusted class to all untrusted classes uniformly.
- An untrusted class may be granted or denied the power to call a trusted method (usually static) or access a trusted field. This power is in addition to the Language's standard access control mechanisms, and is necessary for historical reasons.

Privileges granted to untrusted classes are specified in flags associated with the trusted class. These flags indicate if the trusted class has granted the privilege of subclassing or instance creation to all untrusted classes. Furthermore, flags associated with each method and field in the trusted class indicate whether the method can be called from, or the field accessed from, an untrusted class. This enables an application to run untrusted classes written to the standard CLDC API while retaining some measure of control.

It would be simpler to have a flag that indicated whether all methods and fields in a trusted class could be accessed from untrusted classes. This would be sufficient if the trusted aspects of an application were well factored into specific trusted classes. Although most of the CLDC library can be handled with such a flag, there are cases in CLDC that break this principle. It is in general desirable from a security factorization standpoint that SKVM applications be designed to use class-level security rather than relying on method-level or field-level control.

There have been attempts in the various releases of the Language to enumerate which functions in the core libraries are exposed to sandboxed classes. The flag mechanism provides a means by which such selective exposure can be accomplished. The flag mechanism also provides control over static methods and fields in classes like `system` that

cannot be instantiated. This is important since untrusted classes need access to some fields and methods (such as `system.out`) while other fields and methods (such as `system.exit`) should not be accessible.

## 6.5 Domains

A collection of classes may want to share the privilege to access one another's class resources (instantiation and access to static methods and fields), especially if the classes have been developed together, provide a coherent module of functionality, or are within a shared security perimeter. To reduce the burden of maintaining individual class access privileges, the SKVM supports domains.

A domain privilege is shared among a group of classes in a domain and allows each class to instantiate all other classes in the domain and reference their static methods and fields. With domains, individual access privileges for each class are no longer required. A class may be in only one domain. Domains, reflecting security concerns only (as opposed to name space issues), are distinct from packages, but may be made coincident with them.

Typically, domains allow groups of classes that reference each other frequently (whether through instantiation or static method or field access) to do so without needing to verify access privileges each time. Such privilege verification involves verifying a signature against a public key and can be costly in execution time as well as memory (to store the signature). With domains, an inter-class resource access is permitted as long as the domain membership keys of the accessing and accessed class are equal. In this way, domains simplify application design, application development, and SKVM implementation.

## 6.6 Trusted and Untrusted Packages

The Language employs the package construct to bundle groups of classfiles, not necessarily related in the class hierarchy, into a single name space [JLS, chapter 7]. Packages provide a natural way of organizing and referring to classes and methods. Significantly, classes within a package have access rights to each other's protected fields and methods. Each class is contained in exactly one package (possibly the unnamed package).

In the SKVM, package protected access must be controlled [SJ, page 189]. If it is not, an untrusted class can make itself part of a package containing trusted classes and gain protected access. The mechanism used in the SKVM for controlling package access has four elements:

? A trusted package has an owner and an owner-managed key-pair, analogous to that used for subclassing.

? A package's public key is part of the package's name. If someone tries to put a class in a package without the right public key, the class will be put in a different package. The SKVM simply uses the public key as part of the name space reference.

? Every trusted class is in a trusted package. If a trusted class does not specify a package the SKVM puts it in the unnamed trusted package.

? No untrusted class can be in a trusted package since it does not have a trust certificate.

Note that the VM does not have an *a priori* list of packages. The VM first knows about a package when a class belonging to the package is loaded. The first loaded class defines the package to the SKVM, and any subsequent classes belonging to the same package are checked for consistency. It is the responsibility of all trusted classes in a package to identify the package identically, by name as well as public key.

There may appear to be a security weakness because classes bring in both the package signature as well as the key with which the signature is verified. In fact, while a malicious class could generate a fake package key and a signature consistent with this fake key, it would not be able to join an existing package because it would not be able to replicate the signature associated with the package's private key; it would instead be put in a distinct package.

The Language defines an unnamed package and assigns all classes that do not specify a package name to this unnamed package. Since the SKVM must never mix trusted and untrusted classes in the same package, the SKVM implements separate unnamed packages for trusted and untrusted classes. A trusted class that does not specify a package is automatically put into the trusted unnamed package, while an untrusted class that does not specify a package is automatically put into the untrusted unnamed package. Any package public key specified in the trust

certificate of a class that does not specify a package name is ignored.

While the unnamed package is a convenience during code development, a secure application built for the SKVM should specify packages for all its trusted classes. To encourage such a practice and to provide higher levels of security assurance, the SKVM has a flag that, when set, prevents the loading of any trusted class that does not specify a package. The SKVM by default runs with this flag cleared. Once set, the flag cannot be cleared without restarting the SKVM.

## 6.7 Interfaces

With one notable exception, an interface specifies functionality without providing an implementation [JLS, chapter 9]. The exception is for static initialized fields. In such cases, the initializing expression may contain requests to instantiate objects, which may require SKVM privileges. It is therefore necessary to associate privileges with an interface.

As with a class, an interface has a nominal owner. The owner is responsible for securing all required privileges for the interface. A trusted interface X demonstrates that it can extend a trusted interface Y by presenting a certificate signed by Y. This certificate is analogous to the subclassing certificate and employs the same data structure and mechanisms.

Note that a class that implements an interface can be independently accessed and manipulated as a class in its own right. In such cases, normal Language semantics dictate what can be accessed. Additionally, however, in the specific case of an untrusted class invoking a method or referenceing a field in a trusted class through a trusted interface, both the trusted interface and the trusted class must allow the access.

## 6.8 Inner Classes

The Language allows the definition of inner classes as members of other classes [JLS, section 8.1.2]. These inner classes are implemented through compiler-introduced source code transformations and appear to the VM as distinct classfiles.

The SKVM requires a trusted inner class to present a trust certificate, as any other trusted class would. It is the responsibility of the owner of the trusted inner class to generate this trust certificate. Of course, any tool that supports generation of trust certificates may wish to facilitate the construction of certificates for inner classes. For instance, the tool might handle all name transformations transparently, and employ the same key-pairs for the inner class as it employs for the outer, enclosing class.

The source code transformations introduced by the compiler to support inner classes implement a weakening of access permissions. This is necessary because there is no support in Java virtual machines for direct access to a private member of a class from another class. The specific instances of access permission weakening are:

1. Private inner classes are implemented as package level classes.
2. Protected inner classes are implemented as public level classes.
3. Private class members (fields or methods) that are visible between classes (due to the shared scoping relationship between inner and enclosing classes) are indirectly implemented with package level access. Note that sharing of private members between classes participating in an inner class relationship is achieved by a local protocol of access methods that reflect the mode of access expressed in the source. These methods have package scope and as such are open to any class within the same package.

Like other VMs, the SKVM cannot reliably identify inner classes and therefore cannot determine when such access permissions have been weakened. Therefore, developers for the SKVM platform must be aware of these issues and should avoid the use of inner classes where possible.

If the use of inner classes in a secure application is absolutely necessary, the problems due to weakened access permissions can be avoided by adopting the following guidelines:

1. Classes in an enclosing/inner class relationship should never rely on the shared scoping of their private members.

2. Inner classes should never be declared private or protected.

Following such guidelines will ensure that there is always a one-to-one correspondence between the source level access permissions of a class and its classfile implementation.

## 6.9 Exceptions

When an exception occurs in a running program, the VM unwinds the call stack until the most recently installed relevant exception handler is encountered, which then catches the exception [JLS, section 11.3][JVMS, sections 2.16.2 and 3.10]. The code that throws the exception is never resumed.

There are security issues in adopting such a model directly in the SKVM. For instance, a trusted class could install an exception handler. Control could then be transferred to an untrusted class, which could surreptitiously install another exception handler. Control could then proceed to another trusted class, which could throw an exception. The untrusted class that interposed the extra exception handler would now catch the exception, defeating the intent of the earlier exception handler installed by the trusted class. Such scenarios could threaten the integrity of the application. Alternatively, a trusted class could install an exception handler and, at a later stage, a different trusted class could throw an exception. This second trusted class may not enjoy any trust relationship with the first trusted class. Again, there is an unanticipated transfer of control that complicates security analysis and becomes a potential vulnerability.

It would of course be possible to require that methods in trusted classes that could be invoked from untrusted classes to install a full suite of exception handlers to catch all exceptions before they unwind to any exception handlers installed by the untrusted class. However, this is clearly too expensive and tedious a coding practice.

Note that exceptions are simply standard objects with the additional property that they are derived (indirectly or directly) from `java.lang.Throwable`. As such, package access semantics can be leveraged to prevent classes external to a package from catching exceptions thrown from within the package. A non-public exception (one whose class definition does not include the `public` access modifier) is invisible to all classes outside its package and therefore no handler in these external classes can explicitly declare to catch exceptions.

Unfortunately, package access semantics do not completely control exception handlers. It is legal to hold a reference to an object even though the scope of the reference may preclude any knowledge of the object's complete type. This can be achieved with references to publicly accessible base type (such as `java.lang.Object`). This means that exception handlers can catch exceptions via base class declarations. The lowest common base class for every exception is `java.lang.Throwable`. A handler declared to catch such an exception would catch package-restricted exceptions. While Language semantics prevent the handler's scope from using the exception as an instance of its complete type, the mere fact that it can been caught presents a means for an untrusted class to mask out or alter secure control flow transfer. Thus, a mechanism for preventing this interference is built into the SKVM.

Each trusted class includes a flag within its trust certificate. If the flag is cleared, then any package-restricted exception thrown by **X** can only be caught by exception handlers within the same package as the throwing class. If this flag is set in class **X**, then standard exception semantics are applied when an exception is thrown by any method in **X**.

Note that if all classes that potentially throw exceptions set this flag, exception handling in the SKVM will be identical to, and compatible with, the KVM.

### *6.9.1 Security-Related Exceptions*

The SKVM throws an exception when a privilege verification fails. The trusted throwable *IllegalSubclassException* is thrown by the VM when an attempt to subclass fails because the privilege certificate could not be verified successfully. Similarly, the VM throws *IllegalInstantiationException*, *IllegalPackageException*, *IllegalDomainException*, *IllegalMethodInvocationException* and *IllegalFieldReferenceException* in the appropriate circumstances. When the VM throws one of these exceptions it uses an instance created at VM startup. Establishing whether or not an exception being propogated resulted from a security violation it thus reduced to a simple object address comparison.

### *6.9.2 Error Handling*

One concern with untrusted classes is that they may attempt denial-of-service attacks by causing potentially fatal errors, such as OutOfMemory, to be thrown. Such errors, if not caught correctly, could result in abnormal termination of the SKVM. In order to handle this situation, it is necessary to distinguish whether errors originate from trusted or untrusted class execution. This is implemented by having the VM create an instance of an appropriate exception class, which is used to wrap any error that occurs in an untrusted class. The error thrown by the untrusted class is then presented as an exception. As it is propagated up the call stack, it can only be caught by handlers installed by trusted classes.

Errors thrown by trusted classes continue to elicit the conventional behavior ? that is, they terminate the VM.

## 6.10 Class Installation and Loading

The SKVM distinguishes class *installation*, when a classfile comes onto the platform, from class *loading*, when the class is actually loaded into the VM. It is during loading that the subclassing trust certificate is verified. As such, a class is known to be trusted only when it has loaded successfully and has demonstrated that it has a valid certificate.

A number of issues arise because of this separation between installation and loading:

1. A malicious application may mount a buffer overflow attack to deny service by installing a large number of bogus classfiles. This issue arises for both classfiles with trust certificates and ones without certificates, since trust certificates can only be verified in the context of class loading.
2. SKVM applications may need to be updated in place, and this means that newer classes will be installed that obsolete older classes. Such versioning issues need to be handled correctly.
3. The architecture of the SKVM assures the recipient of a classfile of its authenticity and security. However, it does not assure the sender that the classfile has reached its intended recipient. A malicious recipient may implement a virtual machine that accepts and executes all incoming classfiles, whether they have the requisite privileges or not.

The SKVM addresses these issues with three mechanisms.

Firstly, installation space is separated into trusted and untrusted partitions. Classes claiming to be trusted are installed into the trusted partition while untrusted classes are installed into the untrusted partition. This ensures that any buffer overflow attack mounted by installing untrusted classes is limited to the untrusted partition.

Secondly, classes claiming to be trusted (possessing a trust certificate) are initially put in the untrusted partition and are moved into the trusted partition only when their certificates are verified. How this verification is performed is platform dependent. One possible technique is to evaluate trust certificates statically. When a classfile arrives its subclassing privilege is checked against its superclass, if extant. If the superclass does not exist, the class is held in untrusted space until its putative superclass is successfully installed. If the subclassing privilege is valid the classfile is moved to the trusted partition. Another possible technique is to use installation certificates. Platform owners generate an installation key-pair and distribute the public key to all platforms for use in verification. Trusted classes that wish to be installed onto a platform obtain a certificate signed by the platform owner's installation private key. Classes are installed in trusted installation space on the platform only after this installation certificate is verified against the installation public key.

Thirdly, the SKVM requires the trust certificate of each classfile to include a time stamp. The class loader inspects the time stamps and loads the most recent correctly validated class. This addresses versioning issues.

The SKVM also implements two optimizations. Firstly, when the most recent validated class has been loaded, all other installed classes with the same name are discarded. This ensures that versions that are either out of date or spoofing are purged. Secondly, if a class X being installed claims to be a trusted subclass of Y, and Y is in fact loaded when X seeks to be installed, the SKVM verifies the certificate of X against the Y and discards X immediately if the validation fails.

There are also other heuristics that can be employed to weed out badly formed or malicious classfiles at installation, helping to avoid potential buffer-overflow attacks. However, a correctly designed application can be engineered to

install its classes in an order that allows them to be verified as trusted when they are installed. Such a scheme would continue to work even if untrusted installation space has overflowed as a result of an attack, since trusted classes can be installed directly into trusted space without needing to rely on any of the heuristics discussed above.

Since the SKVM supports trusted class loading, there is no need for the standard CLDC JAR-file loading boundary [CLDC section 3.4.2.3]. Nevertheless, the SKVM supports this boundary in strict KVM mode.

## 6.11 Trust and Bytecode Verification

As described above, current Java implementations employ a bytecode verification process to ensure that classfile code is well formed. This process is not yet provably secure.

With respect to trusted classes, the SKVM instead puts the onus on the owner of the classfile to ensure that the classfile is well formed. This is reasonable, since the owner is trusted. Also, since the owner is additionally responsible for the application-level security of the class, the class will likely be validated. At that time the source code, and the bytecodes, can and should be examined.

Note that the owner could be prepared to trust the verifier to ensure type-safety ? this is acceptable because the SKVM still holds the owner (rather than the verifier) responsible for security assurance.

In the case of untrusted classes, bytecode verification is useful and is done for compatibility with the CLDC specification. However, as stated above, the SKVM cannot be certified relying on the verifier. As a result all such untrusted class files must be put in a strict sandbox, the integrity of which can be validated. This sandbox is described in detail below in the section on memory spaces.

# 7. Cryptographic Support

The SKVM requires cryptographic support to enforce security. Specifically, it requires two functions, one for digital signature support and one for cryptographic hash computation. The SKVM provides basic implementations of these functions, but allows deployment of custom versions.

The basic functions are RSA with a 1024-modulus key as the signature algorithm, and MD5 as the cryptographic hash function.

RSA is a public key algorithm. It operates by creating two keys, a public key and a private key. As the names suggest, the private key is kept private and used for signing certificates while the public key is made known to everyone for verifying certificate signatures. MD5 is a collision-free message digest, or hash function. It computes a 128-bit hash value of an array of data.

Note that RSA is a *de facto* standard and has the advantage of having very rapid signature verification times. However, this is at the expense of rather large signatures. This is not an issue for the subclass privilege, which is verified only on class loading and can be discarded afterwards. However, instantiation certificates have to be maintained in the VM, and each such certificate requires 1024 bits (128 bytes) of storage.

If space is a sufficiently severe constraint that the RSA certificate size is not acceptable, either elliptic-curve RSA, which is secure with approximately 155bits (? 20 bytes), or DSA, for which the signatures are 2?160 = 320bits (40 bytes), can be employed. However, there is a (running time) performance penalty in the use of either ECC or DSA. In addition, DSA optimizes signing at the expense of verification and can be up to 100 times slower than RSA.

Message Authentication Codes (MAC) are an alternative to public key based hash-and-sign signatures. While MACs are considerably less demanding in terms of storage and computation time, a MAC requires a symmetric secret key. The use of MACs should be avoided because they make SKVM integrity depend on secrets internal to the SKVM, violating the precept against secrecy (3.6). Additionally, they require infrastructure for the storage and management of secret keys, and require the secure transmission of classfiles (since keys appear in the clear in the trust certificate). Nevertheless, the choice remains with the platform owner.

Platform owners can integrate custom signature and hash algorithms into the SKVM. However, for security reasons, these must be integrated at the VM level, and not at the class level, in keeping with the precept of static specification.

Note that SKVM architecture does not specify how subclassing and instantiation verification keys are to be managed. Nor does it stipulate how the system protects itself against other forms of attack on public key systems such as spoofing and man-in-the-middle. Such issues depend on the requirements of the application and the trusted community. Note however that since all keys are public keys (unless MACs are used), confidentiality is not required and there is therefore no need to store secrets. All that is required is that the keys be authentic, and the mechanism by which classfiles are loaded ensures this.

# 8. The Trust Certificate

The trust certificate is a collection of data that is attached to each trusted class and that determines its privileges. The certificate is primarily composed of a number of public-private key pairs. The use of these key pairs to sign and verify privileges constitutes the crux of the SKVM. The certificate is packaged as an optional classfile attribute that is understood by the SKVM and ignored by other VMs [JVMS, section 4.7.1].

With each package *P* there is an associated key pair ($PK_P$, $pk_P$), generated by the owner of the package. Similarly, with each domain *D* there is an associated key pair ($DK_D$, $dk_D$) generated by the owner of the domain. With each class *X* there are two associated key pairs, the subclassing key pair ($SK_X$, $sk_X$) and the class resource access key pair ($AK_X$, $ak_X$). The SKVM does not require that all these key-pairs be distinct. Indeed, a key pair can be employed in multiple roles depending on the underlying security policy that the SKVM is enforcing (domain and package being the same, for example).

In the following, class *X* belongs in package *P* and domain *D* and wishes to subclass *Y* and access the resources in class *Z* (not in domain *D*). The trust certificate for *X* consists of these components:

| | |
|---|---|
| $T_X$ | A non-negative integral timestamp indicating the time of creation of *X*. This timestamp is used for version control on installation and loading, and it is assumed that newer versions have large timestamps than older versions. |
| $SF_X$ | A binary flag indicating if objects can subclass *X* without privileges. If the flag is false, then untrusted objects cannot subclass *X* and trusted objects need to present the appropriate certificate (signed hash) in order to subclass *X* successfully. If the flag is true, then all classes can subclass *X* as long as Language semantics are obeyed. |
| $NF_X$ | A flag specifying if objects can instantiate *X* without privilege. If the flag is false, then an object can only instantiate *X* by presenting the appropriate certificate. If the flag is true, then all classes can instantiate *X* as long as Language semantics are obeyed. |
| $MF_X$ | A constant specifying if all objects can invoke static methods in *X*. The constant takes on values **yes** (all static methods in *X* can be invoked by any object, subject to standard Language semantics), **no** (Static methods in *X* can only be invoked by an object that presents the appropriate certificate), and **byMethod** (flags associated with each static method determine if the method can be invoked without privilege). |
| $FF_X$ | A constant specifying if all objects can access static fields in *X*. The constant takes on values **yes** (all static fields in *X* can be accessed by any object, subject to standard Language semantics), **no** (static fields in *X* can only be accessed by an object that presents the appropriate certificate), and **byField** (flags associated with each static field determine if the field can be accessed without privilege). |
| $AK_X$ | A public key used to verify certificates requesting access to the resources of class *X*. |
| $AH_Z(X\|T)$ | A hash of class *X* and its timestamp $T_X$ signed by the class resource access private key of class *Z*. This signature is verified with the public key $AK_X$. Note that there are as many hashes of the form $AH_?(X\|T)$ as there are resources that *X* needs to access from different classes. |
| $DK_D$ | The public key of the domain *D* that *X* belongs in. This key determines the identity of the domain. |

| | |
|---|---|
| $DH_D(X\|T)$ | The hash of class *X* and its timestamp $T_X$ signed by the private key $dk_D$ of domain *D*. This signature is verified with the public key $DK_D$ and is the means by which the SKVM knows that class *X* belongs in domain *D*. |
| $PK_P$ | The public key of the package *P* (if any) that *X* belongs in. This is needed for identification of packages and exception processing. |
| $PH_P(X\|T)$ | The hash of class *X* and its timestamp $T_X$ signed by the private key $pk_p$ of package *P*. This signature is verified with the public key $PK_P$ and is the means by which the SKVM knows that class *X* belongs in package *P*. |
| $EF_X$ | A constant specifying how package-private exceptions are handled in the face of handlers declared to catch then via publicly accessible base classes. The constant takes on values **yes** (all classes can catch package-private exceptions thrown by this class) and **no** (only trusted classes in the same package as the class throwing the exception can catch it). |
| $SK_X$ | The public key for subclassing *X*. This key is always present since it serves the dual purpose of verifying subclassing (when $SF_X$ is true) as well as verifying the authenticity of the classfile. |
| $SH_Y(X\|T\|Cert)$ | A hash of class *X*, its timestamp $T_{X,}$ and all the fields in the trust certificate *minus this field*, signed by the subclassing private key of parent class *Y*. This hash is the signature that is used to validate the subclassing privilege, as well as the authenticity of the classfile and the trust certificate. |

The classfile described here refers to the KVM classfile, which includes the traditional J2SE classfile and the stack-maps generated by the KVM preverifier.

The timestamp above is part of the hash in order to validate the time the class was hashed. This guarantees the integrity of versioning, which is based on the timestamp.

A rogue class cannot use the public key of another domain or package because it will not be able to generate the necessary signed hash, since it does not have access to the private key. The rogue class could generate a separate key pair (in which case it would have the private key of that pair), but the public keys would not match the package or domain keys of other classes and the class will end up in its own domain or package. Since domains and packages are determined by equivalence classes defined on the relation of "equality of public keys", it is not possible for a rogue class to forge admission to a package or domain.

The flags in the trust certificate are designed to be fail-safe. Specifically, the false or **no** setting of the $SF_X$, $NF_X$, $MF_{X,}$, and $FF_X$ flags are secure settings. It is assumed that the false or **no** setting is associated with the zeroed state in the platform (typically integer 0 or Boolean False).

Now consider what happens when the SKVM receives class *X* and wishes to install it. As a trusted class, *X* subclasses *Y* and should be installed as its child. Class *X* demonstrates that it has this privilege by presenting $SH_y(X|T|Cert)$. This same signature also establishes that the owner of *Y* has vouched for the integrity of the contents of *X*. This can be verified with *Y*'s subclassing public key, which can be found in *Y*'s certificate. (Since *Y* is already installed, its certificate must have been previously validated.)

Class *X* proves that it has the privilege of belonging in package *P* and domain *D* by subjecting the signed hashes $PH_P(X|T)$ and $DH_D(X|T)$ to verification using the public keys stored with the package and domain respectively. Membership in an existing package or domain is demonstrated by using the same package or domain public key as an existing class.

If class *X* now wishes to create an instance of class *Z*, it demonstrates that it has this privilege by having the signed hash $AH_Z(X|T)$, which can be verified with $AK_Z$, the public key found in *Z*'s certificate.

Untrusted classes wishing to subclass, create instances, invoke methods, or access fields in *X* rely on the respective flags $SF_X$, $NF_X$, $MF_X$, and $FF_X$.

# 9. Trusted and Untrusted Space

The SKVM encapsulates the execution of untrusted classes in a strict sandbox, monitoring all references and calls out of the sandbox. This is done because the opportunities for compromise are reduced, FIPS certification is easier to obtain, and the constraints put on untrusted classes are reasonable given that the SKVM imposes greater security than the standard KVM.

SKVM users see a unified logical memory space. However, the SKVM maintains two separate memory spaces for trusted and untrusted classes. Trusted (T) space is for classes that come with a trust certificate, while untrusted (U) space is for those that come without a trust certificate.

## 9.1 Properties of Trusted and Untrusted Space

For security reasons, the SKVM implements a barrier between objects in trusted (T) and untrusted (U) space. This barrier is characterized by two properties:

? Transitions from T to U are method and data porous. Trusted objects can invoke methods and access public instance data in untrusted objects.

? Transitions from U to T are data-opaque but method-porous. Untrusted objects are allowed to invoke accessible methods in trusted objects. A method is accessible if the trusted object has explicitly made it available to untrusted objects. However, accessing data (even public data) in a trusted object is not allowed.

The barrier is enabled by turning off the $FF_X$ flags for all classes.

However, for compatibility with CLDC/KVM, the $FF_X$ flag allows a trusted class to permit access to its fields (subject of course to standard Language semantics) by untrusted classes. This feature should be used for compatibility only, since it allows access into T space without active monitoring by a trusted class. The preferred method of field access is via accessor methods, which allow explicit checking by the trusted class. Furthermore, for FIPS 140-2 certification, all security states and transitions must be enumerated. In this context, if an untrusted class accesses a field, then the state enumeration process must take into account the many possibilities that can result from arbitrary sequences of bytecodes.

The $FF_X$ flag for all classes is turned on in strict KVM mode.

## 9.2 Separation of Trusted and Untrusted Space

Each space has a stack and a heap, and both spaces are subject to garbage collection.

Stack frames for T-objects live on the T-stack, while stack frames for U-objects live on the U-stack. Logically the stack is continuous, though as execution meanders its way across the T-U boundary the physical stack becomes disjoint and references cross from one stack to the other. Such an approach does not present a security problem because these references are under the exclusive control of the VM, unseen by programs.

The T-U boundary is crossed under the following conditions:

? A T-method calls a U-method. This is allowed as long as Language semantics are obeyed.

? A T-method accesses a field or static in a U-object. This is allowed as long as Language semantics are obeyed.

? A U-method calls a T-method. This will be checked when the `invoke` bytecode is executed. Only T-methods that have been flagged as accessible from U-space can be called. If the check fails, an exception is thrown.

? A U-method attempts to access a field or static in a T-object. This will be checked when the `getfield`, `putfield`, `getstatic`, or `putstatic` bytecode is executed. Only T-fields that have been flagged as accessible from U-space can be accessed. If the check fails, an exception is thrown.

## 9.3 Allocation of Computing Resources

An untrusted class may mount a denial of service attack by attempting to consume all computing resources available on the platform. The SKVM addresses this issue by limiting the resources available (notably CPU time and memory space) available to untrusted classes. This is most easily done with command line arguments that specify limits on the resources available to untrusted classes.

## 9.4 Garbage Collection

Garbage collection runs separately in T and U space. Pointers from one space to the other are tracked and identified during garbage collection, and modified as necessary during compaction.

The type maps generated by the preverifier are also used by the KVM for exact garbage collection of T space. Since garbage collection requires knowledge of all pointers to objects, it is necessary at all times to be able to ascertain whether local variables and the stack contain pointers into the heap.

Determining which of the local variables and stack entries are pointers is relatively easy in T-space given the type maps. Note that the owners of the trusted classes have vouched for the correctness of these type maps. However, the type maps for U classes cannot be relied on. An ancillary type stack is therefore maintained, updated with every stack operation. When garbage collection occurs the type stack identifies the items on the execution stack that are pointers.

There is an obvious performance penalty of maintaining an ancillary type stack. The SKVM therefore provides a flag that allows the platform owner to enable this type stack. The flag can be accessed through a command line argument. For high security assurance applications that require U-side support and are prepared to tolerate a performance penalty in the interests of security, the type stack can be enabled. Note that with the type stack disabled, the SKVM relies on the correctness of the type-maps (and hence the preverifier-verifier) for security.

## 9.5 Error Handling

Since U space is distinct from T space, the errors that are thrown when the stack and heap in each space are exhausted must also be distinct. As a result, a trusted class can recognize that an error occurred in U space and continue normal T space execution. Given this, and the fact that a trusted application will always be able to catch such an error (because the application starts execution in T space), untrusted classes are prevented from mounting successful denial of service attacks against a trusted application. A well designed, robust, secure application that employs untrusted classes should be architected to respond to these untrusted errors.

# 10. SKVM Applications and Their Development

SKVM applications are CLDC applications: programs with a main method [CLDC, section 3.2]. An application's component classes are loaded when necessary. When the class containing the main method is loaded, the application is registered and is then run. Class loading is controlled by the security mechanisms described above.

The SKVM employs a Java Application Manager, or JAM, similar to JAMs used with the KVM. The JAM assumes that there is local storage (typically a file system or a local database) that stores installed classes and from which classes can be loaded. The complete application may be resident, or components may be loaded dynamically over a communication channel. The JAM starts the SKVM and indicates which application (which classfile with a main method) should be run.

For the purposes of defining trusted communities and establishing the initial trust relationships, the CLDC library can be thought of as owned by the platform. Applications wishing to execute on the platform will have to request and obtain privileges to subclass the CLDC library. Alternatively, the entire application can run untrusted.

If the platform is one of many being issued by an authority (such as a payment token being issued by a credit card company) then all platforms may share a common CLDC owner and hence have identical subclassing and instance-creation public keys.

For development purposes, a platform authority can release a version of the platform with a different CLDC signing key-pair created purely for application development, with the private key released to developers. This allows the developers to sign their classes each time they are changed, without having to request the authority to do so. When the application is complete, it is installed on the production platform with a different set of keys, with the CLDC signing key kept private.

The SKVM architecture explicitly does not support multiple applications running in the same virtual machine. This type of support is usually a requirement of systems such as application servers or applets running within a browser. The goal in these environments is to protect applications from each other to a degree comparable with process isolation on a standard operating system, less protection than the SKVM aims to provide.

The KVM supports the KVM Debug Wire Protocol [KDWP], a debugging protocol that is a subset of the JDWP standard. The SKVM implements the KDWP, but only during debugging. It obviously must be removed for deployment. The KDWP may be enhanced to display the additional information present in the SKVM, such as that pertaining to trusted and untrusted space.

The security support in the SKVM is enabled by configuration settings. Given fail-safe design principles, these settings default to secure modes. However, they can be set so that all security features in the SKVM visible to developers are disabled, with the result that the SKVM is identical in function to the KVM.

For backward compatibility reasons, the SKVM can be initiated in strict KVM mode. In this mode, SKVM security features are disabled and all code runs in untrusted space. Trust certificates are not required for any of the classes. Strict KVM mode is enabled if the class containing the `main` method is untrusted. The KVM default restriction that all classes in an application be in a single JAR file is enforced in strict KVM mode [CLDC section 3.4.2.3].

# 11. References

## 11.1 Specifications Referenced


[CLDC] *Connected, Limited Device Configuration, Specification Version 1.1*; Sun Microsystems, <May> 2002; http://java.sun.com/products/cldc.

[FIPS] *Security Requirements for Cryptographic Modules*, NIST FIPS PUB 140-2, 25 May 2001.

[ICS] *Inner Classes Specification*; Sun Microsystems, 4 Febuary 1997; http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html

[J2PS] *Inside Java 2 Platform Security*; Li Gong; Addison-Wesley; October 1999.

[JC] *Java Card Technology for Smart Cards*; Zhiqun Chen; Addison-Wesley; June 2000.

[JLS] *The Java Language Specification, Second Edition*; James Gosling, Bill Joy, Guy Steele, Gilad Bracha; Addison-Wesley, June 2000.

[JVMS] *The Java Virtual Machine Specification, Second Edition*; Tim Lindholm, Frank Yellin; Addison-Wesley, April 1999.

[KDWP] *KVM Debug Wire Protocol (KDWP), Version 1.0*; Sun Microsystems; 26 February 2001.

[TCK] *CLDC Technology Compatibility Kit version 1.0a User's Guide*; Sun Microsystems; February 2001.


## 11.2 Other References


[4758] *Building the IBM 4758 Secure Coprocessor*; Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, Steve Weingart; IEEE Computer; Oct 2001; pg 57-66.

[AC] *Applied Cryptography, Second Edition*; Bruce Schneier; John Wiley and Sons; 1996.

[IBM] *Building a High-Performance, Programmable Secure Coprocessor*; Sean Smith, Steve Weingart; IBM Research Division, T.J. Watson Research Center; RC 21102 (94393), 19 February 1998.

[PE] *Software Development Environment, Programmer's Reference Manual V1.0*; D'Crypt Pte Ltd; Sep 2001

[S&L] *Secrets and Lies*; Bruce Schneier; John Wiley and Sons, 2000.

[SE] *Security Engineering: A Guide to Building Dependable Distributed Systems*; Ross Anderson; John Wiley and Sons; 2001.

[SJ] *Securing Java*; Gary McGraw, Edward W. Felten; John Wiley and Sons, 1999.