Table of Contents

A Mechanism for Secure, Fine-Grained Dynamic Provisioning of Applications on Small Devices

William R. Bush[1], Antony Ng[2] Doug Simon[1], and Bernd Mathiske[1]

[1] Sun Microsystems Laboratories, Mountain View CA 94043, USA,
bill.bush@sun.com,
WWW home page: http://research.sun.com/
[2] D'Crypt Pte. Ltd

Abstract. As small, secure devices become more powerful and more wide spread, it has become desirable to support the dynamic provisioning and updating of multiple applications on such devices. This paper presents a simple mechanism for performing such provisioning and updating, even if the applications are mutually distrustful. The mechanism extends CLDC Java^{TM}technology with a classfile attribute that carries the certificates necessary to enable the added security.

1 Background

The work described here was motivated by a number of developments and considerations:

- Small, secure devices, such as smart cards and cryptographic modules, are becoming more capable.
- Such devices are being used in more complex situations running multiple applications.
- Updating the software on such devices once deployed is highly desirable, to provide both new functionality and software fixes, but poses various security issues.
- A dynamic provisioning mechanism supporting such activity should be small and simple, because of device limitations and to aid in verification and certification.
- The Java platform has appeared on small devices and provides dynamic class loading and some basic security features.

The resulting solution presented here:

- supports the secure incremental replacement and extension of software on small devices,
- enables distinct trust communities developing distinct applications,
- accomplishes this by extending a common version of the Java platform, and
- enables additional support for capabilities and running untrusted code.

2 The Java Context

The Java platform was the starting point for this investigation because it already provides various features supporting security and dynamic provisioning. Unlike C and C++, for example, it guarantees type and pointer safety. Various Java versions provide different forms of application isolation. And all versions support some mechanism for dynamically installing classes.

2.1 The Connected Limited Device Configuration

The work described here is specifically aimed at the next generation of smart cards and other similar small devices. Thus the technology base used is the Connected Limited Device Configuration (CLDC) version of Java 2 Platform, Micro Edition (J2ME$^{\text{TM}}$technology) [1]. This version of the Java platform is the smallest one that supports most standard features of the Java language (in contrast to the much more restrictive Java Card$^{\text{TM}}$specification [4]). It outlines a basic set of libraries and Java virtual machine features. Compliance with the CLDC specification is demonstrated by passing the CLDC Technology Compatibility Kit (TCK) tests [10].

The heart of the configuration is the K Virtual Machine (KVM) [8]. The KVM is a virtual machine designed with the constraints of small devices in mind. Named to indicate that its size is measured in tens of kilobytes, the KVM is simple, in order to minimize memory footprint. This simplicity makes the KVM easy to understand and modify, important characteristics in the context of security.

2.2 CLDC Security

The CLDC security model [1, 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 $^{\text{TM}}$platform by the security manager [3]. 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 are 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 a more secure platform. 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.

2.3 MIDP Security

The Mobile Information Device Profile (MIDP) [MIDP] is a set of additions to the base CLDC platform that supports mobile phones. Among the additions is a security mechanism [11], [12]. The MIDP security mechanism is based on two concepts: protection domains and JAR file signing.

A protection domain is a set of permissions granted an application, and defines the application's sandbox (see [13] and [14] for descriptions of nuanced sandboxes). An application runs in a single protection domain. A MIDP platform may define various domains, but required domains include Manufacturer, Operator, Third-Party, and Untrusted. Some permissions may only be granted through interaction with the user of the device (confirming use of the permission).

JAR file signing is used to verify the authenticity of an application. MIDP requires that an application reside in a single JAR file, which is typically signed using X.509 PKI infrastructure, support for which is required by the MIDP standard. The MIDP device uses the certificates it possesses to authenticate the application.

This security model has limitations with respect to high security devices. Specifically:

- Permissions are coarse grained and set when the device is manufactured. A set of permissions is *a priori* bound to a domain, and an entire application then executes in one of those predefined domains.
- Support for X.509 PKI is required, which may not be appropriate and can be cumbersome.
- User interaction may be required to grant some permissions.

In sum, the MIDP platform has been carefully tuned for mobile phones. High security devices are different.

2.4 Compatibility and Security Goals

The broad goal of this work is to develop a more secure version of the KVM (the Secure KVM, or SKVM), with a particular focus on dynamic provisioning. More specifically:

- Correct CLDC/KVM applications should not be able to distinguish the KVM from the SKVM on the basis of observed behavior.
- Correctly implemented secure applications operating normally should not be able to distinguish the KVM from the SKVM on the basis of observed behavior.
- Only malicious classes should elicit different behavior from the SKVM than they would from the KVM.
- The SKVM must pass the proper compatibility tests (the CLDC TCK).

Additionally, the SKVM should be capable of being validated as secure, specifically achieving FIPS 140-2 certification [2]. FIPS 140-2 is a specification drawn up by the National Institute of Standards and Technology, defining security requirements for cryptographic modules. 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. For example, the IBM 4758 has been certified at the highest level (of the predecessor FIPS 140-1 standard) [15].

3 Key Precepts

A number of key precepts guided the design of the SKVM architecture (see [16], [17], [18]).

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 FIPS 140-2 certification).

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 (some) 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 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.

4.1 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.

Components that are not part of a verifiable trust community are untrusted, and are not allowed to execute (subject to an optional feature discussed in Section 6.2).

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.

4.2 Trusted 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.

Intuitively, a trusted class is provided for some trusted community of owners. Trusted classes are the means by which sensitive information is encapsulated. 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, and will thus not be loaded by the SKVM.

All the certificates of a trusted class are bundled into a new class attribute called a trust attribute. (Class attributes are the classfile mechanism used to store extra information about a class, and are described in the Java Virtual Machine Specification [6, section 4.7].)

4.3 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 the installation of individual classes. Specifically, the SKVM enforces a subclassing privilege that enables one class to install itself 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.

Note that accesses to static and instance methods and fields are controlled through the Language's private, protected, and public tagging [5, section 6.6]. Note that the Java Virtual Machine Specification stipulates that accesses have to be checked at run time [6, sections 5.4.3 and 5.4.4]. Information about private, protected, and public access permissions are stored in the classfile [6, sections 4.5 and 4.6] to enable such run time checking.

The S privilege enables a precise definition of a trust community. Since all classes except `java.lang.Object` subclass some other class, the class hierarchy tree rooted at `java.lang.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:

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

Note that the ability to subclass a class does not imply the ability to subclass any parent of the class in the class hierarchy independently.

4.4 Trusted Classes and Packages

The Language employs the package construct to bundle groups of classfiles, not necessarily related in the class hierarchy, into a single name space [5, 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 access must be controlled in order to control access to protected fields and methods (see [19], page 189).

The mechanism used in the SKVM for controlling package access has three elements:

- A 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 class is in a package. If a class does not specify a package the SKVM puts it in the unnamed package.

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 (or, optionally, when installed on the device; see below). 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 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. A class that does not specify a package is automatically put into the unnamed package. Any package public key specified in the trust attribute 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 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 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.

4.5 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.

4.6 Inner Classes

The Language allows the definition of inner classes as members of other classes [5, 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 attribute, as any other trusted class would. It is the responsibility of the owner of the trusted inner class to generate this trust attribute. Of course, any tool that supports generation of trust attributes may wish to facilitate the construction of attributes 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:

- Private inner classes are implemented as package level classes.
- Protected inner classes are implemented as public level classes.
- 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. The problems due to weakened access permissions can be avoided by adopting the following guidelines:

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

The use of anonymous inner classes should be avoided due to the difficulties of managing their trust relationships.

The use of non-static inner classes should also be avoided since they are in a sense syntactic sugar for static inner classes and thus hide detail that makes security analysis harder.

4.7 Exceptions and Trust Relationships

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 [5, section 11.3] [6, 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 class could install an exception handler and, at a later stage, a different class could throw an exception. This second class may not enjoy any trust relationship with the first class. As a result there is an unanticipated transfer of control that complicates security analysis and becomes a potential vulnerability.

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 static type of the reference may preclude any knowledge of the object's complete type. This can be achieved with a reference 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 to mask out or alter secure control flow transfer. Thus, a mechanism for preventing this interference is built into the SKVM.

Each class includes a flag within its trust attribute. 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.

5 The Trust Attribute

The trust attribute is a collection of data that is attached to each trusted class and that determines its privileges. The trust attribute is primarily composed of a number of public keys. The use of these keys to sign and verify privileges constitutes the crux of the SKVM. The trust attribute is understood by the SKVM and ignored by other VMs.

With each package P there is an associated key pair (PK_P, pk_P), generated by the owner of the package. With each class X there is an associated key pair, the subclassing key pair (SK_X, sk_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 (subclass and package being the same, for example).

In the following, class X belongs in package P and wishes to subclass Y. The trust attribute for X is described in Table 1.

The classfile described here refers to the CLDC classfile, which includes the traditional J2SE classfile and the stack-maps generated by the CLDC 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 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 keys of other classes and the class will end up in its own package. Since 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.

The EF_X flag in the trust attribute is designed to be fail-safe. Specifically, the false or **no** setting is the secure setting. 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 $\mathrm{SH}_Y(X|T|Cert)$. This same signature also establishes that the owner of Y has vouched for the

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 larger timestamps than older versions.
SK_X	The public key used for verifying attempts to subclass X.
$\mathrm{SH}_Y(X\|T\|Cert)$	A hash of class X, its timestamp T_X, and all the fields in the trust attribute *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 attribute.
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.
$\mathrm{PH}_P(X\|T)$	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 them 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).

Table 1. Components of a Trust Attribute

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 by subjecting the signed hash $\mathrm{PH}_P(X|T)$ to verification using the public key stored with the package. Membership in an existing package is demonstrated by using the same package public key as an existing class.

6 Contextual Issues

The SKVM as described above requires certain platform support to operate properly. In particular, classfiles stored on device must be handled correctly, and specific cryptographic functions must be available.

6.1 When Trust Attributes are Checked

In general, the arrival and storage of classfiles on a device will occur before the SKVM needs to load them (as is generally the case with CLDC platforms). In addition, some of the new classfiles arriving on a device may replace existing ones. These circumstances make it potentially desirable to check trust attributes at times other than class loading.

The CLDC specification [1, section 5.3] gives the platform implementor considerable freedom. The classfile lookup order is implementation dependent and a classpath is not required. It is required that the lookup order cannot be manipulated by the application programmer in any way. (Note that the platform must read classfiles and JAR files [1, section 5.3.1]. Applications that are "distributed publicly" on a network open to the public must be in JAR format. Also note that the JAR file loading boundary required by the CLDC specification is not necessary in the SKVM, since the SKVM uses a stronger security mechanism. Nonetheless, the SKVM supports this boundary in strict KVM mode.)

The SKVM follows the CLDC specification and thus does not impose an ordering or, therefore a particular time for checking trust attributes. The earliest attributes can be checked is when classfiles arrive on a device; the latest is when they are executed for the first time (per application), the traditional load time).

Various issues arise that affect order considerations. First, for devices that may install classfiles from a potentially malicious source, buffer overflow attacks via classfiles with bogus attributes are possible. A device could be flooded with apparently proper classfiles that can only be flushed when their attributes are checked. Second, newly arrived classfiles may dynamically replace a subset of the classes in an application, which requires versioning and the rechecking of attributes.

The way to deal with the buffer overflow problem is to check trust attributes when classfiles arrive on the platform. Otherwise, classfiles would have to be kept around indefinitely. Applications must therefore be engineered to install their classes in an order that allows them to be verified as trusted when they are installed. The burden of meeting this constraint is on the application designer: the superclass must be either already installed or immediately available.

One implementation of this checking is to use an arrival buffer. Classes arriving on the device are initially put in this buffer and are moved into classfile storage when their trust attributes are verified. JAR files are also unpacked in this buffer. When a classfile arrives its subclassing certificate is checked against its superclass, if extant. If the trust attribute or superclass is absent, the classfile is flushed. If the subclassing certificate is valid the classfile is moved to classfile storage; otherwise it is flushed.

With respect to the versioning issue, the SKVM requires the trust attribute of each classfile to include a time stamp. The trust attribute checker (whenever run) inspects the time stamps, validates the newest class, and discards the old class (or marks it as to be deleted if the old classfile is being used by a running application).

Rechecking of attributes can be done in one of two ways. If attribute checking is done at installation then the installation of a replacement class in turn requires that classes granted privileges by it must be rechecked, if anything in its attribute has changed. If checking is done dynamically, then no extra processing is necessary. (Note that binary compatibility is important but is not an SKVM issue; it is rather handled by the VM in due course.)

6.2 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 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 (described in Section 6.2) have to be maintained in the VM, and each such certificate requires 1024 bits (128 bytes) of storage.

If standard RSA certificate size is found to be unacceptable, either elliptic-curve RSA, which is secure with approximately 155bits ($\approx$ 20 bytes), or DSA, for which the signatures are $2 \times 160 = 320 bits$ (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. 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 attribute). 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 the 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.

6.3 Security-Related Exceptions

The SKVM throws an exception when a privilege verification fails. Depending on the circumstances, `IllegalSubclassException` or `IllegalPackageException` is thrown by the VM when a privilege certificate could not be verified successfully. When the VM throws one of these exceptions it uses an instance created at VM startup. Establishing whether or not an exception being propagated resulted from a security violation is thus reduced to a simple object pointer comparison.

6.4 SKVM Applications and Their Development

SKVM applications are CLDC applications: programs with a main method [1, 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 SKVM architecture does not require that the complete application be resident – components may be loaded dynamically over a communication channel – although an implementation may impose this restriction. The JAM starts the SKVM and indicates to it 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.

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 does not support multiple applications running in the same virtual machine. The goal of such support 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 [7], 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, the SKVM can be initiated in strict KVM mode. In this mode, SKVM security features are disabled. Trust attributes are not required for any of the classes. Strict KVM mode is enabled if the class containing the main method has no security attribute. The KVM default restriction that all classes in an application be in a single JAR file is enforced in strict KVM mode [1, section 3.4.2.3].

7 Additional Optional Functionality in the Trust Attribute

The trust attribute mechanism presented above can be extended with additional information to support other, optional, security-related features besides dynamic provisioning. The specific extensions explored in the SKVM involve implementing a form of capability-based control and enabling limited execution of untrusted classes.

7.1 Support for Capability-Oriented Design

Good object-oriented and secure programming practice mandates the factoring and encapsulation of data. This factoring enables a capability-based style of programming [20], in which a capability is represented by an object, and references to that object are controlled. Ideally, all references to a capability object would be monitored, and unauthorized uses prevented, but the overhead of checking all references is too great.

A simpler (and less secure) mechanism can be constructed to control the creation of objects (as capabilities) and references to static fields and methods. With this compromise scheme, the creation of capabilities is monitored but their use is not. Class-based references are also monitored.

This mechanism has been implemented in the SKVM via another trust attribute privilege: the privilege to create a new instance of (that is, instantiate) an object and reference its static methods and fields. This privilege is referred to as the **A** (“access”) privilege, with the sense of accessing the resources of a class. There are two implicit 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.

7.2 Domains

In practice the access privilege can be burdensome to administer. 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.

7.3 Additions to the Trust Attribute to Support Capabilities

With each class X there is the associated class resource access key pair (AK_X, ak_X).

With each domain D there is an associated key pair (DK_D, dk_D) generated by the owner of the domain.

Consider a class X that belongs in package P and domain D and wishes to access the resources in class Z (not in domain D). The additional components of the trust attribute for X relating to capabilities for this type of access are described in Table 2.

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

Class X proves that it has the privilege of belonging in domain D by subjecting the signed hash $\mathrm{DH}_D(X|T)$ to verification using the public key stored with the domain. Membership in an existing domain is demonstrated by using the same domain public key as an existing class.

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

AK_X	A public key used to verify access requests to the resources of class X.
$\mathrm{AH}_Z(X\|T)$	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_Z. Note that there are as many hashes of the form $\mathrm{AH}_Z(X\|T)$ 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.
$\mathrm{DH}_D(X\|T)$	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.

Table 2. Components of a Trust Attribute for supporting Capabilities

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.

As with packages, a rogue class cannot use the public key of another domain 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 domain keys of other classes and the class will end up in its own domain. Since domains 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 domain.

7.4 Loading and Executing Untrusted Classes

Untrusted classes are ones without any trust attributes (as distinguished from classes with invalid trust attributes, which are mistrusted). Such classes can provide useful, CLDC-standard functionality if their execution is strictly controlled.

In the SKVM this is done by keeping untrusted classes in a sandbox and allowing trusted classes to grant privileges to untrusted ones. 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*.

7.5 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:

- An untrusted class may be allowed to subclass a trusted class. Like all other privileges granted to untrusted classes, this is a privilege that the trusted class in question must grant explicitly.
- In the capability-based style discussed above, and similar to a trusted class, 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 with flags in the trust attribute of the trusted class. In addition, 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 `java.lang.System` that cannot be instantiated. This is important since untrusted classes may need access to some fields and methods (such as `java.lang.System.out`) while other fields and methods (such as `java.lang.System.exit`) should not be accessible.

7.6 Additions to the Trust Attribute to Support Untrusted Classes

For a class X, Table 3 describes the additional components of the trust attribute that relate to untrusted classes.

These flags 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

SF_X	A binary flag indicating if objects can subclass X without privileges. If the flag is false, then untrusted classes cannot subclass X and trusted classes 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).

Table 3. Components of a Trust Attribute for supporting Untrusted Classes

false or no setting is associated with the zeroed state in the platform (typically integer 0 or Boolean False).

8 Implementation

A version of the SKVM has been implemented using the KVM code base version 1.03. The implementation includes basic support for secure dynamic provisioning, support for capabilities, and support for untrusted classes.

In general, changes to the KVM were small and localized.

Note that support for capabilities and untrusted classes involves extra runtime checks and thus incurs a performance penalty. This penalty was not measured.

8.1 Secure Dynamic Provisioning

Seven files (out of 24) required modification, and one small file was added. The details of these changes are shown in Table 4. The total increase in size, in lines of code, is 4%.

File	LoC in base KVM	Additional LoC for SKVM	% Increase
class.c	1985	216	11%
collector.c	2096	54	3%
crypto.c	0	124	100%
frame.c	1149	6	1%
hashtable.c	718	76	11%
loader.c	2957	381	13%
nativeCore.c	1287	12	1%
pool.c	437	46	11%
total	*23759*	*915*	*4%*

Table 4. Code size overhead of Secure Dynamic Provisioning

This increase consists of enhancements to: identify and process trusted classes (loader); manage certificates (hashtable), privileges (collector), and other runtime structures (crypto); perform privilege checks (class, nativeCore, and pool); and handle exceptions (frame).

Additionally, a stand alone tool was written to annotate classfiles with properly constructed trust attributes.

8.2 Capabilities

Support for capabilities pushed the total code size increase to 6, to a total of 24272 lines. The details of these changes are shown in Table 5.

This further increase consists of enhancements to: process the capability-based privileges (loader); handle domain intersection and resource access checks (class); and handle exceptions (frame).

8.3 Untrusted Classes

Support for untrusted classes bumped the total code size increase to 7, to a total of 24538 lines.

File	Additional LoC for Capabilities
class.c	243
frame.c	4
loader.c	266

Table 5. Code size overhead of Capabilites

File	Additional LoC for Untrusted Classes
loader.c	171
pool.c	95

Table 6. Code size overhead of Untrusted Classes

This final increase consists of enhancements to: process the untrusted class privileges (loader); and handle access checks (pool). The details of these changes are shown in Table 6.

9 Status and Further Work

As described above, a prototype version of the SKVM has been implemented based on the standard KVM. There are other, more recent CLDC implementations that are potentially better platforms on which to base the SKVM (such as [21]). Further work with the SKVM will likely be done using one of these implementations.

The next major step in demonstrating the feasibility and value of the SKVM will be porting it to a cryptographic module. This task may expose platform and deployment issues. It will also enable real-world testing of SKVM applications.

The subsequent step will be the FIPS certification of the SKVM. This effort will require precise definition of the operation and implementation of the SKVM.

A possible enhancement involves untrusted classes. If it is determined that they are truly useful on a secure platform, they can be completely isolated in their own execution environment [22]. They could be given their own heap, execution stack, and resource limits.

References

1. *Connected, Limited Device Configuration, Specification Version 1.1*; Sun Microsystems, May 2002; `http://java.sun.com/products/cldc`.

2. *Security Requirements for Cryptographic Modules*; NIST FIPS PUB 140-2, 25 May 2001.
3. *Inside Java 2 Platform Security*; Li Gong; Addison-Wesley; October 1999.
4. *Java Card Technology for Smart Cards*; Zhiqun Chen; Addison-Wesley; June 2000.
5. *The Java Language Specification, Second Edition*; James Gosling, Bill Joy, Guy Steele, Gilad Bracha; Addison-Wesley, June 2000.
6. *The Java Virtual Machine Specification, Second Edition*; Tim Lindholm, Frank Yellin; Addison-Wesley, April 1999.
7. *KVM Debug Wire Protocol (KDWP), Version 1.0*; Sun Microsystems; 26 February 2001.
8. Information on the KVM can be found at `http://java.sun.com/products/cldc`.
9. *Mobile Information Device Profile for Java 2 Micro Edition, Version 2.0*; Java Community Process, November 2002; `http://java.sun.com/products/midp`.
10. *CLDC Technology Compatibility Kit version 1.0a User's Guide*; Sun Microsystems; February 2001.
11. "MIDP 2.0 Security Enhancements"; Otto Kolsi, Teemupekka Virtanen; *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*; January 2004.
12. "Understanding MIDP 2.0's Security Architecture"; Jonathan Knudsen; February 2003; `http://developers.sun.com/techtopics/mobility/midp/articles/permissions/`
13. "MAPbox: Using Parameterized Behavior CLasses to Confine Untrusted Applications"; Anurag Acharya, Mandar Raje; *Proceedings of the 9th USENIX Security Symposium*; August 2000.
14. "A Flexible Containment Mechanism for Executing Untrusted Code"; David S. Peterson, Matt Bishop, Raju Pandey; *Proceedings of the 11th USENIX Security Symposium*; August 2002.
15. *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; October 2001; pp. **57-66**.
16. *Secrets and Lies*; Bruce Schneier; John Wiley and Sons, 2000.
17. *Applied Cryptography, Second Edition*; Bruce Schneier; John Wiley and Sons; 1996.
18. *Security Engineering: A Guide to Building Dependable Distributed Systems*; Ross Anderson; John Wiley and Sons; 2001.
19. *Securing Java*; Gary McGraw, Edward W. Felten; John Wiley and Sons, 1999.
20. "Programming Semantics for Multiprogrammed Computations"; Jack Dennis, Earl Van Horn; *Communications of the ACM*; March 1966; pp. **143-155**.
21. "A Java Virtual Machine Architecture for Very Small Devices"; Nik Shaylor, Doug Simon, Bill Bush; *Proceedings of the 2003 ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems*, June 2003, pp. **34-41**.
22. "A Secure Java Virtual Machine"; Leendert van Doorn; *Proceedings of the 9the USENIX Security Symposium*; August 2000.