Managing the life-cycle of Java Card applets in other Java virtual machines *

Michael Roland, Josef Langer, René Mayrhofer

University of Applied Sciences Upper Austria
Softwarepark 11, 4232 Hagenberg, Austria
E-mail: michael.roland@fh-hagenberg.at

Abstract

Purpose: Today, for developers, it is difficult to get access to an NFC secure element in current smart phones. Moreover, the security constraints of smartcards make in-circuit debugging of applications impractical. Therefore, it would be useful to have an environment that emulates a secure element for rapid prototyping and debugging. This paper addresses the design, implementation, performance and limitations of such an environment.

Design/methodology/approach: Our approach to such an environment is the emulation of Java Card applets on top of non-Java Card virtual machines (e.g. Android Dalvik VM) as this would facilitate the use of existing debugging tools. As the operation principle of the Java Card VM is based on persistent memory technology, the VM and applications running on top of it have a significantly different life-cycle compared to other Java VMs. We evaluate these differences and their impact on Java VM-based Java Card emulation. We compare possible strategies to overcome the problems caused by these differences, propose a possible solution and create a prototypical implementation in order to verify the practical feasibility of such an emulation environment.

Findings: While we found that the Java Card inbuilt persistent memory management is not available on other Java VMs, we present a strategy to model this persistence mechanism on other VMs in order to build a complete Java Card run-time environment on top of a non-Java Card VM. Our analysis of the performance degradation in a prototypical implementation caused by additional effort put into maintaining persistent application state revealed that the implementation of such an emulation environment is practically feasible.

Originality/value: This paper addresses the problem of emulating a complete Java Card run-time environment on top of non-Java Card virtual machines which could open and significantly ease the development of NFC secure element applications.

*This paper is an extended version of M. Roland, J. Langer, and R. Mayrhofer. (Ab)using foreign VMs: Running Java Card Applets in non-Java Card Virtual Machines. In *Proceedings of the 11th International Conference on Advances in Mobile Computing & Multimedia (MoMM2013)*. ACM, Vienna, Austria, December 2013 [16].

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

Keywords

Java Card, Emulator, Secure Element, Near Field Communication, Card emulation, Android, Persistence

1 Introduction

Smartcards are pervasive in our every-day lives. We use them as keys to buildings, as bank and credit cards, as bus tickets, as SIM cards in our mobile devices, or to descramble pay-TV. Also many of our identity documents (e.g. passports) contain smartcard microchips.

While some of these smartcards contain only memory that can be read and written using a smartcard reader, many smartcards contain a processor that executes complex software programs, thus the word "smart". In the past, smartcards typically contained pre-programmed application-specific software. Today, however, many smartcards follow an open approach: The smartcard operating system provides a standardized run-time environment consisting of a standardized programming interface and a standardized instruction set. Thus, it becomes possible to develop and run applications independent of the actual hardware and operating system implementations.

The most widespread platform for open and interoperable smartcard applications is Java Card. The Java Card platform uses a subset of the Java language and has been optimized for execution on smartcards. While the Java Card platform itself is open, smartcards are tightly controlled environments. Deployment of software onto smartcards requires credentials that are typically only known to the card issuer or a trusted third party. This closed nature of smartcard microchips is particularly an issue with Near Field Communication (NFC) in mobile devices. NFC card emulation mode, where a mobile device acts as a contactless smartcard, relies on a smartcard chip, the secure element, to perform the actual card emulation. The secure element typically belongs to the manufacturer of the mobile device or to the mobile network operator. Management of the secure element is often delegated to a trusted service manager (TSM). This closed and complicated environment usually makes it impossible for the average developer to deploy (even for prototyping and testing) applications to a secure element [14] – even though approaches towards opening parts of this ecosystem are starting to appear (cf. [9]).

Besides the issue of deploying application prototypes, another issue is in-place source-level debugging. In the context of the secure element in an NFC-enabled smartphone, in-place debugging would mean that the Java Card applications running on the secure element would be debuggable while they are executed on the smartcard chip and accessed by apps through regular secure element APIs, as well as by external smartcard readers through the contactless NFC interface. Thus, a developer could step through the executed code while the Java Card application interacts with external application components. However, due to their high security requirements, current secure element microchips cannot be attached to a debugger for in-circuit emulation. As a result, it would be helpful for developers to have an alternative to the secure element which provides an open environment for debugging and rapid prototyping of Java Card applications that could later be deployed to actual secure elements.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

In the context of Android-based mobile devices, a Java Card emulator (cf. [15]) could be built on top of the system-native Dalvik virtual machine – a VM that executes translated Java code. As the Java Card language is a subset of the Java language, it is possible to compile Java Card applications for other Java virtual machines. Only the Java Card specific APIs would need to be added to the Java run-time environment. The in-place debugging and rapid prototyping capabilities could be provided by interfacing the Java Card emulator to existing secure element APIs (e.g. the Open Mobile API) and to host card emulation mode APIs which are available on some Android-based devices (cf. [2, 14, 15]).

The main benefit of using a standard Java virtual machine for emulation is the seamless integration with existing debuggers. For instance, the Dalvik virtual machine already provides source-level debugging using the Android debugger. Similarly, a Java SE virtual machine also comes with ready-made debugger integration. When creating a Java Card emulator with a custom Java Card virtual machine, however, these capabilities would also have to be manually implemented.

Nevertheless, there is one big issue when running Java Card applications in non-Java Card virtual machines: A Java Card virtual machine has a significantly different life-cycle compared to other Java VMs. While a standard Java VM typically starts when the Java application starts and terminates when the Java application terminates its execution, the Java Card VM starts when the smartcard is manufactured and terminates when the smartcard is destroyed. Thus, the state of the Java Card VM and the applications that run on top of it is persistent across the whole lifetime of the card regardless of power cycles, etc. Besides the differences in the life-cycle, the Java Card run-time environment has a transaction mechanism that assures atomic modification of application data and rollback in case of torn transactions.

In this paper, we evaluate these differences in virtual machine and application life-cycles and their impact on Java VM-based Java Card emulation. We base these evaluations on specific scenarios for secure element emulators for the Android platform (cf. [15]). We discuss possible solutions to the problems that arise from these differences in the life-cycles and propose an implementation for persisting Java Card application state in non-Java Card virtual machines. Our solution uses the Java reflection API to collect information about classes and objects, and to store and later reconstruct the persistent objects of a Java Card application. We implement a working prototype of the persistence framework for Java SE and Android that fulfills all qualities that we need for persisting Java Card applications. We then integrate our persistence framework into the open-source Java Card simulator jCardSim running on top of the Java SE virtual machine. Finally, we evaluate the performance of our Java Card emulation environment comparing scenarios with and without the persistence mechanism.

2 Java Card

Java Card technology is a subset of the Java programming language combined with a run-time environment that is optimized for tiny embedded devices like smartcards [19]. The run-time environment consists of a Java Card virtual machine (as defined in [20]), a Java Card specific API and Java Card specific security features.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

To assure that Java Card applications have a small footprint that matches the constrained resources of a smartcard, many features of the Java language are unavailable in the Java Card language. For instance, Java Card only supports the primitive data types *boolean*, *byte*, *short* and optionally *int*. Furthermore, most of the core API classes of the Java language are unsupported. Nevertheless, the Java Card language is a true subset of the Java language. Thus, all Java Card language constructs also exist in the Java language. However, the Java Card API provides several classes with smartcard-specific functionality like communication using smartcard commands, management of PIN codes and cryptographic keys, and execution of cryptographic operations that do not exist in the Java API.

The Java Card virtual machine is the abstraction layer between Java Card applications and different device platforms. Interoperability across different hardware platforms is provided through a common instruction set and a common application binary format. The Java Card VM lifetime is equivalent to the lifetime of the smartcard microchip [20]: The VM is installed and started during the manufacturing process and terminates when the chip is destroyed. In between, the VM lifetime spans even across any power cycles (though the VM appears to be inactive while power is removed). This means that Java Card applications that run inside the VM also run across power cycles. Applications, their data, and their state are preserved through storage in persistent memory (e.g. EEPROM).

The main entry point of a Java Card application is an applet instance. An applet provides several public methods for interaction with the Java Card runtime environment. The applet's *install* method is invoked to create and initialize an applet instance. After installation, applet instances remain in a suspended state until they are explicitly selected through a smartcard command. During selection, the applet instance's *select* method is invoked to prepare the applet for further processing. Once an applet is selected, all further smartcard commands are forwarded to that applet instance by triggering its *process* method. Upon selection of another applet instance, the current applet instance's *deselect* method is invoked and the applet instance returns into suspended state. Similarly to the Java Card VM, Java Card applets execute forever (or until they are explicitly uninstalled). However, applet instances return to the suspended state upon power loss.

In addition to storing application data associated with applet objects in persistent memory by default, the Java Card platform provides atomic transactions on this data. That is, certain modifications to persistent data can be guaranteed to be either performed completely or not to happen at all. Through that transaction mechanism, an applet can assure that data that belongs to one transaction is consistently stored to persistent memory. Thus, when a transaction is aborted (either explicitly or though power loss), any changes to persistent data are rolled back to the state before the transaction started. Only when a transaction completes successfully, changes to persistent data are committed to persistent memory.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

3 Secure Element Emulators

Several Java Card simulators exist which allow simulation and testing of Java Card applets without real smartcard hardware. The Java Card reference implementation, for instance, integrates with the Java ME secure element API. However, it can only process compiled Java Card applications and does not permit source-level debugging. Several smartcard manufacturers provide their custom Java Card simulation environments that simulate their specific smartcard architectures (e.g. G&D Java Card Simulation Suite [8] and Gemalto Simulation Suite [7]). Moreover, there exist Java Card simulators that run on top of standard Java virtual machines (e.g. Java Card Workstation Development Environment [18] and jCardSim, which is discussed in more detail below). Unfortunately, none of these simulators are known to integrate with current smartphone operating systems (specifically with the Android platform). Also, none of these simulators can be used for true prototyping of secure element applications.

As a consequence, we developed scenarios for integrating an open environment for debugging and rapid prototyping of secure element applications on Android devices and on the Android platform emulator (cf. [15]). The resulting Java Card emulator can be used as a drop-in replacement for a secure element on the Android platform and provides comparable functionality to a regular secure element. While the emulator operates at a much lower security level, it is an open platform that is available to all developers and that does not require the complicated and closed ecosystem of a regular secure element chip.

3.1 Java Card Emulator for the Android Emulator

The idea behind this concept is to use the existing open-source Java Card runtime environment simulator implementation *jCardSim* (see [5]) and to integrate it with the Android emulator as well as to interface it with card emulation hardware. The simulator runs on top of the Java SE virtual machine. Thus, it already runs in an environment that supports source-level debugging. jCardSim currently supports only two ways of interacting with the simulated applets: scripts that consist of smartcard commands and access through the Java Smart Card IO API.

Figure 1 shows our scenario for integrating the Java Card emulator with the Android emulator. The Open Mobile API-based secure element API of Android (cf. [9, 17]) is extended with a terminal interface that connects to jCardSim. Moreover, the card emulator hardware (e.g. an NFC reader in software card emulation mode) is attached to jCardSim so that smartcard commands can be passed between the card emulator hardware and the simulator.

3.2 Java Card Emulator for Android Devices

In the second scenario (see Fig. 2), the Java Card emulator is embedded into the mobile phone system as a middleware. Thus, in this scenario, jCardSim is ported to Android and runs on top of the Dalvik virtual machine. Again, an interface to access the emulator is added as a terminal interface to the Open Mobile API-based secure element API implementation of Android. Moreover, the emulator is connected to the host card emulation API (sometimes also called

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

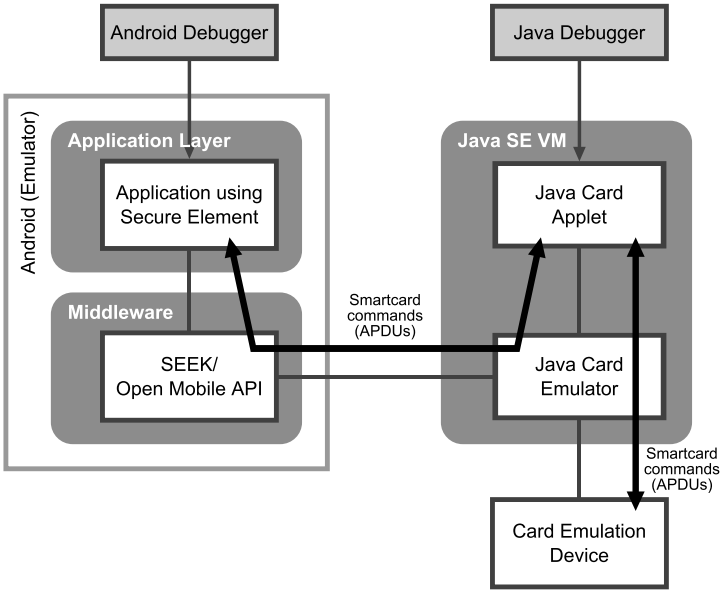


Figure 1: A Java Card emulator attached to the Android platform emulator [15].

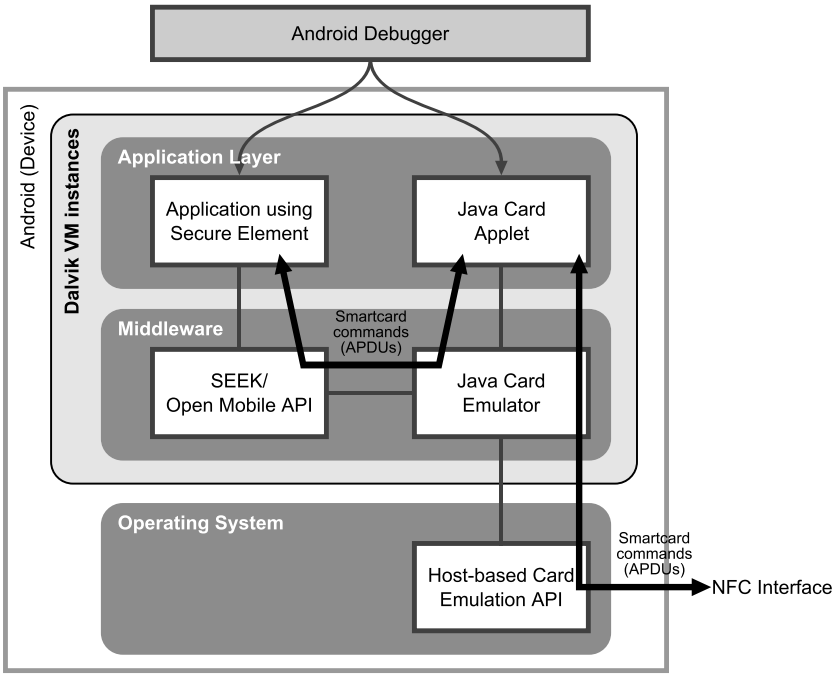


Figure 2: A Java Card emulator integrated into an Android device [15].

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

"soft-SE" or "software card emulation") that is available on CyanogenMod and on Android 4.4 devices (cf. [2, 14, 22]).

3.3 Deficiencies of these Scenarios

The main problem with these scenarios is the persistent nature of the Java Card VM. While the Java Card virtual machine and the objects it uses to represent applications and their data can persist throughout the whole lifetime of a smartcard, the Java VM and the Dalvik VM are both processes of the underlying operating system and only run while an application process is executing. Thus, the lifetime of the latter two VMs is bound to the lifetime of the Java applications that run on top of them. Both, the Java VM and the Dalvik VM, terminate when the last execution path of a Java application terminates, when the VM process is forced to terminate by the operating system or when the host system resets.

As a consequence, simply porting the Java Card API to the Java VM or the Dalvik VM does not provide the same level of capabilities as the Java Card run-time environment. Specifically, persistence of objects is not available by default. This means, however, that Java Card applets (as well as their data and their state) are lost when the virtual machine that runs the emulator terminates. This is usually not a problem for short term simulation and debugging sessions where the Java Card emulator is actively used. However, when emulating a secure element in a mobile device, we also want to perform long-term tests and prototyping. In these scenarios, an ideal secure element emulator would even persist any application state across power-cycles of the mobile device.

4 Application and Virtual Machine Life-cycles

Applications running on top of non-Java Card VMs have fundamentally different life-cycles as compared to Java Card applets on the Java Card VM. The Java Card VM lifetime spans across the whole smartcard lifetime. Due to use of persistent memory technologies for application and data memory, applications and their state persist from installation until uninstallation. Thus, an application appears to continue to run across power and reset cycles. Moreover, the Java Card run-time environment provides a transaction mechanism that guarantees certain operations to be atomic.

4.1 Java SE Virtual Machine

In our scenario of a Java Card emulator attached to the Android emulator, the Java Card emulator environment runs on top of the Java SE virtual machine on a PC system. The Java Card emulator is completely detached from the Android system and the Android emulator instance. As a consequence, power-cycles and reconfiguration of the Android system do not interfere with the life-cycle of the Java Card emulator. This makes it possible to persist the state of Java Card applications even across multiple boot-ups of the Android system inside the Android emulator. Only a restart of the host operating system or of the Java VM executing the Java Card emulator would result into loss of Java Card applet/application state.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

Nevertheless, for prototyping and long-term tests (particularly in combination with external smartcard readers through host card emulation), it would be necessary to preserve the Java Card application state across multiple executions of the Java Card emulator. A facility built into Java SE that could potentially help with persisting object state across Java VM life-cycles is serialization. However, the Java serialization mechanism typically requires modifications to the Java Card application source code.

Another significant difference between Java SE and Java Card is the transaction mechanism. Java SE does not support transaction atomicity. This may not be an issue for the Java Card emulator in many situations as the emulator would continue to run even if the connection between the emulator and the secure element API or the connection between the card emulation device and an external reader is torn. Thus, any remaining code would be executed because there is no power-loss upon disconnecting the emulator. However, a Java Card application could also intentionally abort a transaction. In that case, the emulator would need to rollback the application state to the state before the transaction started.

jCardSim, the open-source implementation that we use as the basis for our emulator scenarios, has been designed to run on top of a Java SE virtual machine. Consequently, it suffers from the above problems. In its current version it is neither capable of persisting state across simulation sessions nor of simulating the Java Card atomic transaction mechanism.

4.2 Android Dalvik Virtual Machine

Java applications on Android consist of windows (so-called *activities*), background *services*, broadcast message receivers and databases (so-called *content providers*). All components of one application typically share one Dalvik virtual machine that runs as an operating system process.

Activities are only active while they are visible and in the foreground. Broadcast receivers are only active while they process a received message. Therefore, the secure element emulator would typically run as a service in the background. As Android is designed for resource-constrained mobile devices, application components (like activities and services) and their virtual machine container processes may be shut down at any time in order to free resources. While a process with a foreground activity is unlikely to be terminated due to resource shortage, background services are significantly more likely subject to resource reuse.

In order to recover from situations where an Android application was terminated by the system, the Android platform already contains a mechanism for storing and recovering application state. However, Java objects representing application state and data are not persisted automatically. Instead, it is up to each application to manually store and recover data that is relevant to current application state. Therefore, the Android state recovery is not comparable to the Java Card VM persistent memory.

As a result, the situation for Java Card emulation inside the Dalvik VM is even worse than with the Java VM. The Dalvik VM and, consequently, the state of all Java Card applications running inside the emulator may be lost as a result of device reboots or as a result of the system automatically terminating idle or potentially unused processes. This would be an issue for long-term tests and for prototyping of Java Card applications.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

For the transaction mechanism, the same limitations as with the Java VM apply. Thus, the Dalvik VM and the Android platform do not support rollback of application state to a defined boundary.

4.3 Resulting Problem

According to the specification of the Java Card run-time environment [19], applet instances (i.e. objects instantiated from an applet class) and objects referenced from a persistent object field or from a class static field are persistent. In addition, the Java Card transaction mechanism requires that application state can be saved during application execution and that the application state can later be rolled back to these defined boundaries.

However, preserving application state in non-Java Card virtual machines requires manual implementation. Typically, this would mean that modifications to the Java classes of the Java Card applications would become necessary (e.g. to extract and implant the state of private member fields). However, simulation and debugging should be performed with the original applet source code in order to keep source code interoperable between the emulator and a real smartcard. Moreover, modifications could potentially cause mismatches to the execution on real smartcard hardware. Therefore, the emulator should be capable of extracting and implanting the state of the emulated Java Card applications without requiring modifications to their source code.

5 Related Work and Approaches

A facility for persisting state in Java is serialization. Serialization permits a graph of connected objects to be converted into and restored from a byte stream [3]. Serialization, however, has several disadvantages: Serialization works only for one complete directed graph of connected objects that is generated by starting at one root node and iterating through all objects reachable from that node. Serializing and deserializing from multiple root nodes would result into completely separated object graphs after deserialization. Thus, even if two graphs referenced the same object before serialization, they would reference two separate instances after deserialization. While this could be overcome by merging all potential root nodes into a new object (e.g. a list that references all root nodes) and using that new object as the root node for serialization, it would be impossible to serialize or deserialize only a specific part of an object graph. Also, it is not possible to automatically serialize or deserialize the static members of a class. Moreover, Java requires serializable classes to be tagged as *serializable*. Thus, modifications to the source code of the classes are necessary. However, there also exist serialization frameworks (e.g. Kryo) that do not require tagging. Still, only serialization of an object graph as a whole would be possible.

Another technique that could potentially be used to introduce persistence to Java objects without modifying the actual classes is aspect-oriented programming (AOP). AOP is used to add functionality to a program on a level that is independent of a program's abstraction layers and modularization. Rashid and Chitchyan [13] describe how aspects can be used to add persistence to existing classes. However, their implementation requires these classes to fulfill

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

certain qualities. For instance, getter and setter methods are expected in order to retrieve and modify an object's fields. Moreover, AOP frameworks for Java usually need pre-processing of the application source code or post-processing of byte code in order to map the aspect-oriented Java code into code that runs on a regular Java VM. This is, for instance, the case with AspectJ for Java SE and Android. As a result, source-level debugging of Java Card applications would be severely hindered.

Similar to AOP, frameworks like Xposed for Android permit to hook into arbitrary method calls of an existing application and to execute customized code upon these hooks. However, Xposed currently only supports intercepting method calls and does not permit to directly hook into updates on an object's fields.

A programming technique that comes close to what we would like to achieve is object-relational mapping (ORM). There exist many ORM frameworks for Java (e.g. ActiveObjects, Apache Cayenne, Hibernate and ORMLite). With ORM, the objects of an application are mapped into a relational database. Thus, it becomes possible to store objects to and retrieve objects from a relational database. Nevertheless, typical ORM frameworks need modifications of the application source code (e.g. adding annotations, adding no-argument constructors or even adding getter/setter methods in order to access certain fields; cf. [21]). Overall, ORM seems to require detailed knowledge of application-specific data structures in order to implement the database mapping. However, this is not the case as we want our solution to be capable of handling arbitrary Java Card applications.

A technique comparable to ORM is object data management group (ODMG) binding. This technique maps Java objects to an object-oriented database. Therefore, the schema of the database that contains the persistent Java objects matches the Java classes [3]. While ODMG bindings do not require modifications to the source code of persistent classes, these bindings require either a pre-processor to add the bindings to the Java source code or a post-processor to add the bindings to the Java byte code [4]. Thus, they would have a significant impact on source-level debuggability.

6 Proposed Solution

While existing techniques already provide means to persist Java objects, our ideal solution should fulfill the following qualities:

- Networks of objects starting from one or more root nodes should be storable to and recoverable from persistent memory maintaining all references.
- Static fields of classes should be persistable by specifying a list of classes.
- Full or partial object graphs should be restorable to a defined previous state.
- No modifications to the source code of Java Card application classes should be required in order to be able to use the same code on a card and in the emulator.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

- No pre- or post-processing should be required in order to maintain source-level debuggability with existing tools.
- The solution should work for both Java SE and Android.

Therefore, we decided to create our own implementation of a persistence mechanism for Java. Our solution makes use of the Java reflection API which is available for Java SE as well as for Android. Reflection has the advantage that no knowledge of the classes and their fields is required at compile time. Moreover, we use the Objenesis library to instantiate Java objects without calling any constructor of a class in the process of deserialization.

Our implementation creates an in-memory copy of the object graphs of all Java Card applications installed and running in our emulator environment. For each object in the object graph, the in-memory copy consists of a wrapper object ("state representation") that references the original object and contains references to all its member field state representations. Similarly, for each array, the wrapper contains a reference to the original array as well as to each array element state representation. For primitive data types, the state representation contains a copy of the primitive value.

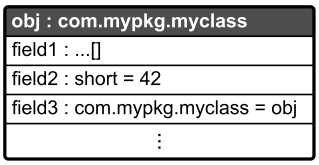
For initial collection of application state, our implementation starts at defined root nodes (the Java Card applet instances that are registered in the Java Card run-time environment and all static fields of classes in the installed Java Card application packages). It then iterates through the whole object graphs spanning from these nodes, creates an object state representation for each new object, records the object state representation in a list of references, and links the object state representations to each other to create the in-memory copy of the whole object graphs (see Fig. 3 for an example).

In order to update the state representation of a (potentially) modified object and its connected partial object graph, our implementation starts at the state representation of that object and checks if all fields still match the recorded references. If changes are discovered, links are updated and wrapper objects for new objects are created. This process is repeated for all parts of the object graph reachable from that object.

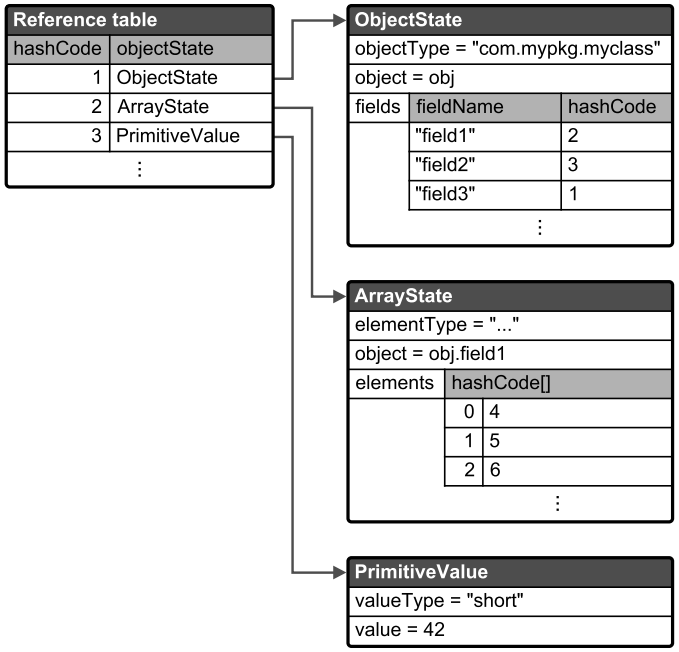
Similarly, in order to revert the state of an object and its connected partial object graph to the state that was previously stored in the in-memory copy of the object graph, our system starts at the state representation of that object and recursively restores all fields to the values and references stored in the object state representation.

The object state representation is designed so that the whole in-memory copy of the object graph can be easily serialized to and deserialized from an XML file. During serialization, each state representation recorded in the list of references is processed and its contained information (reference hash code, type information, fields and associated reference hash codes for objects, a list of array element hash codes for arrays or the primitive value for primitive data types) is written to the XML file. Similarly, a list of known root objects and classes keeps track of known root references and class static fields. Deserialization parses the XML file, recreates the in-memory copy of the object graph, and recursively re-instantiates all objects, arrays and primitive values based on the deserialized object state representation (see Fig. 4).

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036



(a) Example object *obj*



(b) In-memory object state representation

Figure 3: In-memory object state representation generated for an example object *obj*.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

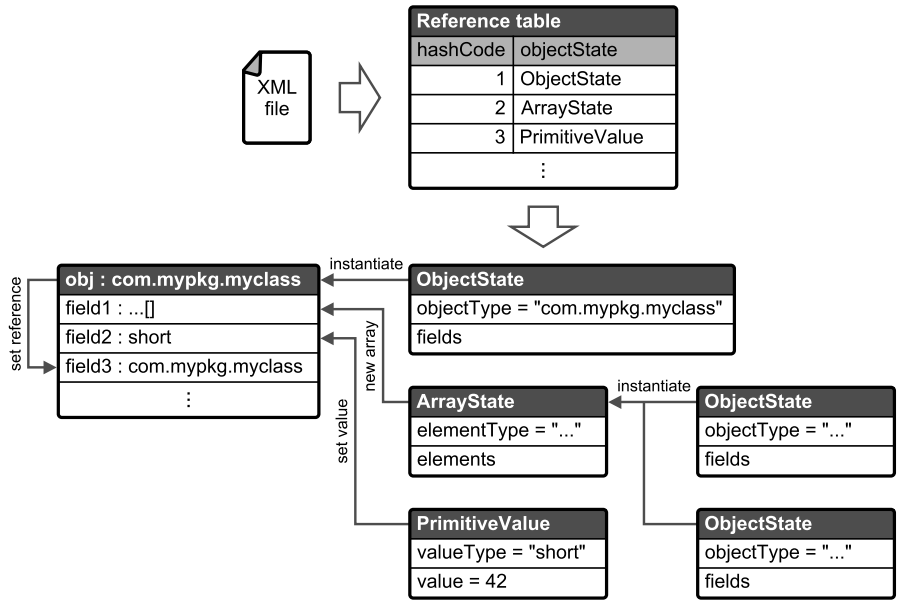


Figure 4: Deserialization of in-memory copy of object state and recreation of object graph.

7 Implementation

In this section we present the central parts of our implementation.

7.1 Uniquely Identifying Objects

The first obstacle when collecting information about the object graph is to uniquely identify objects. The Java *Object* base class provides a `hashCode()` method that returns a 32-bit integer value for all objects. This hash code has been designed for fast indexing and clustering in hash tables. The Java SE API [12] requires that this hash code is equal for any two objects that are equal according to the result of their `equals()` method. The default behavior of the `Object.equals()` method is to be only true if two compared references refer to the same object. However, classes may override the `equals()` method to introduce a more relaxed equality relationship between objects that could permit different objects to still be equal. As the contract for the `hashCode()` method requires the same hash code value to be returned for equal objects, all such objects would return the same hash code. In order to overcome this, the Java API provides a method `System.identityHashCode()` that returns the hash code value that would be returned by the default implementation of `Object.hashCode()` for any object. Unfortunately, even in that case, the Java SE API [12] requires only that the hash code value is equal for one and the same object, but it does not guarantee that unequal objects have different hash code values. Therefore, even using the `System.identityHashCode()` method, hash code collisions may occur. Consequently, the Java hash code implementation cannot be used to uniquely identify objects.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

Effectively, the only way to check that two references refer to the same object is a comparison with the == operator (`x == y`). This fact can be used to assign unique identifiers to objects. A map can be used to map a list of object references to a list containing their unique identifier values. Whenever a unique identifier is needed for an object, the reference is looked up in the list of object references using the == comparison operator. If a unique identifier has previously been recorded, the object reference will be found and will map to a unique identifier value. Otherwise a counter counting all registered objects is incremented by one and that value is registered in the map as the new unique identifier for that object reference. A similar approach has been used by the Kryo framework to map object references to their serialization position.

Moreover, once a unique identifier is no longer needed, the reference can be removed from the map and the unique identifier can be marked as reusable (in a list of reusable identifiers). This is useful in order to prevent an overrun of the overall object counter and to save memory used by the unique identifier map.

In addition, immutable numeric primitives can be considered equal if they are of the same primitive type and have the same primitive value. Therefore, their object identifier can be calculated based on their type and value information instead of using the map-based approach. This simplification significantly reduces the effort required to keep track of object references.

7.2 Reference List

The in-memory copy of the object graph is recorded in a list of references. The list of references maps unique object identifiers to object state representations. Whenever a new object is found, its object state representation is added to the list of references. When an object field refers to an object that has already been recorded, that reference can be resolved using the reference list. Thus, the in-memory copy of the object graph can be created without duplicating objects that are referenced from multiple places. Moreover, serialization of the whole object graph can be performed by linear iteration over the reference list.

7.3 Iterating through the Object Graph

In order to create an in-memory copy of the object graph, our implementation collects object state by starting at defined root nodes. Following the Java Card run-time environment specification [19], these root nodes are the Java Card applet instances that are known to the run-time environment as they and any objects referenced from their fields are supposed to be persistent.

For each object, it is determined if the object already exists in the list of references. If it does, the existing object state representation is used. Otherwise a new object state representation is created and registered. If the object is a primitive value, a state representation of this primitive value is created. If the object is an array, an array state representation is created and the object state collection is repeated for each array element. The object state representation of each element is then linked to the parent object state. Similarly, if the object is neither a primitive value nor an array, each non-static field that is declared by the object class or any superclass is processed and linked to the parent object. Consequently, the whole object graph is copied by recursive iteration through all fields/elements spanning from the root nodes (see Fig. 5).

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

```
ObjectState process(Object object, Class clazz) {
  if (exists(object)) {
    return getExistingObjectState(object);
  } else {
    ObjectState objectState;
    if (isPrimitive(object) || clazz.isPrimitive()) {
      objectState = new PrimitiveValue(object);
      register(objectState);
    } else if (isArray(object) || clazz.isArray()) {
      objectState = new ArrayState(object);
      register(objectState);
      for (Object element : toArray(object)) {
        ObjectState elementState =
                process(element,
                        getComponentClass(object));
        objectState.addElement(elementState);
      }
    } else {
      objectState = new ObjectState(object);
      register(objectState);
      Class objectClass = object.getClass();
      while (objectClass != null) {
        for (Field field : objectClass.getDeclaredFields()) {
          field.setAccessible(true);
          if (!Modifier.isStatic(field.getModifiers())) {
            Object fieldValue = field.get(object);
            Class fieldType = field.getType();
            String fieldName = objectClass.getName() +
                               "#" + field.getName();
            ObjectState fieldState =
                    process(fieldValue, fieldType);
            objectState.addField(fieldName, fieldState);
          }
          objectClass = objectClass.getSuperclass();
        }
      }
    }
    return objectState;
  }
}
```

Figure 5: Iterating through the object graph.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

```
for (Field field : clazz.getDeclaredFields()) {
  field.setAccessible(true);
  if (Modifier.isStatic(field.getModifiers())) {
    Class fieldType = field.getType();
    if (!(Modifier.isFinal(field.getModifiers()) &&
          (fieldType.isPrimitive() ||
           (fieldType.isArray() &&
            fieldType.getComponentType().isPrimitive())))) {
      Object fieldValue = field.get(null);
      String fieldName = clazz.getName() +
                         "#" + field.getName();
      ObjectState fieldState =
              process(fieldValue, fieldType);
      classState.addField(fieldName, fieldState);
    }
  }
}
```

Figure 6: Iterating through classes.

7.4 Iterating through Classes

The Java Card run-time environment specification [19] defines that, besides the object graphs spanning from Java Card applet instances, all static fields of all classes within a Java Card application package are persistent. Therefore, also the object graphs spanning from all static fields of classes have to be recorded. In our prototype, we chose to manually define a list of classes for each Java Card application loaded into the emulator. Based on that list, our implementation iterates through each declared static field and collects the associated object state representation (see Fig. 6).

The reason why we chose to manually define a list of classes was simplicity during the development of our prototype. A good solution will depend on the framework that is provided to developers for adding application packages to our Java Card emulator middleware. The Java VM itself does not provide an easy means to get a list of all classes at run time. The origin of this limitation is that the Java class loader only loads classes on demand [11]. One possible approach could be that developers add their Java Card application in the form of JAR files. In that case, our emulator could traverse the list of class files in that JAR file and automatically build the list of classes.

7.5 Serialized Object Graph

The classes used for state representation of objects and class static fields were designed so that they can easily be serialized to and deserialized from XML. The resulting XML file (see Fig. 7) is used as the back-end database for persisting object state. Three lists are used for serialization:

- the list of references, which contains the state representation of each object in the object graph,
- the list of classes, which contains the state representation of all classes included in state recording, and

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

```
<References>
  <ObjectState hashCode="0" fieldType="null" />
  <ObjectState hashCode="1" fieldType="com.mypkg.myclass">
    <Fields>
      <Field name="com.mypkg.myclass#field1" hashCode="2" />
      <Field name="com.mypkg.myclass#field2" hashCode="3" />
      <Field name="com.mypkg.myclass#field3" hashCode="1" />
    </Fields>
  </ObjectState>
  <ArrayState hashCode="2" fieldType="[L...;">
    <Elements elementType="...">
      <Element hashCode="4" />
      <Element hashCode="5" />
      <Element hashCode="6" />
    </Elements>
  </ArrayState>
  <PrimitiveValue hashCode="3" fieldType="java.lang.Short">
    <Value primitiveType="eShort">42</Value>
  </PrimitiveValue>

  (...)

</References>

<Classes>
  <ClassState className="com.mypkg.myclass">
    <Fields>
      <Field name="com.mypkg.myclass#INSTANCE"
             hashCode="1" />
    </Fields>
  </ClassState>
</Classes>

<NamedInstances>
  <NamedInstance name="obj" hashCode="1" />
</NamedInstances>
```

Figure 7: Serialized object graph in XML.

- the list of root objects (i.e. object instances that can be referenced by a constant name across multiple executions).

7.6 Deserialization and Recreation of Object Graph

During the first step of deserialization, the three lists (references, classes and root objects) are recreated from the XML file. Based on the object identifiers, links between the state representations can be re-established.

As the next step, the original object graph is recreated. For each primitive value, a new instance of that value is created. For each array state representation a new array of the given type and length is created. The array elements are then filled with references to the respective objects. For each object, a new instance is created using the Objenesis library:

```
Object instance = ObjenesisHelper.newInstance(objectClass);
```

This avoids calling the constructor of a class for object creation and permits easy instantiation of classes that do not have a no-argument constructor. As

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

```
for (Entry<String, ObjectState> entry
          : fields.entrySet()) {
  String fieldName[] = entry.getKey().split("#", 2);
  ObjectState fieldState = entry.getValue();
  Class clazz = Class.forName(fieldName[0]);
  Field field = clazz.getDeclaredField(fieldName[1]);
  field.setAccessible(true);
  fieldState.restoreInstanceToField(field, instance);
}
```

Figure 8: Restoration of object fields.

the complete state of the object is restored from serialized data, processing of the constructor is not necessary.

Then, each field is restored with its primitive value or with a reference to the respective object (see Fig. 8). This process is performed recursively for all root objects and all class static fields in order to recreate all reachable objects in the object graph. Moreover, the unique object identifiers are mapped to the new objects.

7.7 Updating and Reverting Objects

Besides serialization and deserialization of the whole object graph, we also want to be able to update and revert subgraphs starting at any object.

During an update, the in-memory copy is updated to reflect modifications on the underlying objects. Starting at the node to be updated, we recursively iterate through all fields (and array elements) and check if the fields refer to the same objects as before. If a modification is detected, the object state representation is updated so that all fields map to the object state representations of the new references. If a new object is discovered, a state representation is created and added to the list of references.

During a reversal, the underlying objects are reverted to the state that was previously recorded in the in-memory copy. Similar to the update procedure, the objects are processed by recursively iterating through all fields (and array elements). Each field/array element is modified to refer to the object that it referenced at the previous update.

Moreover, for both update and reversal each processed object is marked as processed in order to prevent re-processing in case of loops in the object graph.

7.8 Garbage collection

So far, object state collection has only added objects but did not remove them if they are no longer used in the object graph. In order to permit garbage collection on such objects, as soon as an object state representation is no longer linked from any other object state representation and is none of the known root objects, it is removed from the list of references. Moreover, the unique object identifier that has been assigned to that object is freed for reuse.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

7.9 Integration with jCardSim

Several steps have been taken to integrate our persistence framework into jCardSim:

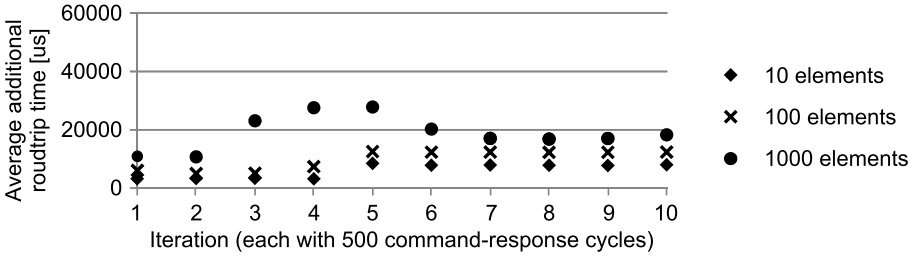
- When the emulator instance is started, existing serialized object state (XML file) is deserialized in order to recreate the state of a previously saved Java Card run-time environment.
- When applet packages are installed into the emulator environment, they and all the classes contained in the package are registered to our persistence mechanism.
- After each command-response sequence (application protocol data unit (APDU) exchange), the in-memory copy of the object state of the applet and all classes belonging to the application package containing the applet is updated.
- When an applet starts a new transaction, the in-memory copy of the object state of that applet and all classes belonging to the application package containing that applet is updated.
- When an applet aborts a transaction, that applet and all classes belonging to the application package containing that applet are reverted to the in-memory copy of the object state.
- When the emulator instance is terminated, the in-memory copy of the object state is serialized to persistent storage (XML file).

8 Performance

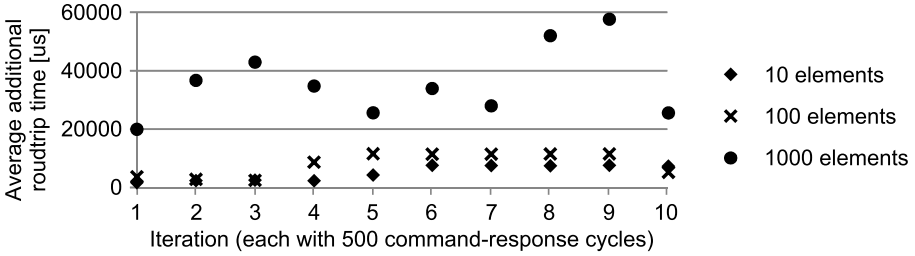
In order to measure the performance of our persistence mechanism, we created two test scenarios and run them on the Java SE 7 (r51) virtual machine on Windows 7 64bit on an Intel i5 (2.67 GHz):

1. In the first scenario, we use an applet that contains a simple data structure consisting of a numeric value of type *short*, a reference to the applet itself and an array of objects, where each object contains two numerical value fields (one of type *short* and one of type *byte*). The applet has two commands, one that simply responds with status "success" and one that changes the applet-internal state by changing the numeric value fields and by replacing elements of the array with new object instances. Thus, this scenario shows the performance of the state extraction with and without modifications of the application state.
2. In the second scenario, an applet that contains various cryptographic keys is used. For this scenario, various cryptographic operations (key generation, encryption, decryption, etc.) are performed within the applet. After each run of the cryptographic operations (causing significant changes to the internal application state), the performance of a simple ping-pong command-response sequence that does not cause any internal state change is measured. Thus, this scenario shows how the performance of the overall system varies after recording significant changes of the application state.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036



(a) In the case that no data is modified



(b) In the case that the data is modified

Figure 9: Average additional command-response delay (in microseconds) induced by the persistence mechanism for scenario 1 for array sizes of 10, 100 and 1000 elements.

Both scenarios are performed with and without the persistence mechanism being enabled.

Figure 9 shows a comparison of the command-response delay in scenario 1 for array sizes of 10, 100 and 1000 elements over 10 repetitions with 500 command-response cycles each. While the average command-response delay without the persistence mechanism is around 7.7 ms for all cases, the average command-response delay increases to around 13.6 ms for 10 array elements, to around 15.8 ms (no state changes) and 16.4 ms (with state changes) for 100 elements, and to around 27.8 ms (no state changes) and 43.4 ms (with state changes) for 1000 elements. Thus, the average additional command-response delay induced by the persistence mechanism is about

1. 5.9 ms for 10 array elements regardless of modifications to application state,
2. 8.1 ms for 100 array elements (with an additional 0.6 ms if the application state is modified upon every command), and
3. 20.1 ms for 1000 array elements (with an additional 15.6 ms if the application state is modified upon every command).

Figure 10 shows the average and median additional command-response delay in scenario 2 for the simple ping-pong sequence over 10 repetitions with 500 command-response cycles each. Between each of the 10 repetitions, a series of commands triggering various cryptographic operations is issued, causing significant changes to the applet-internal state. The diagram in Fig. 10 reveals a

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

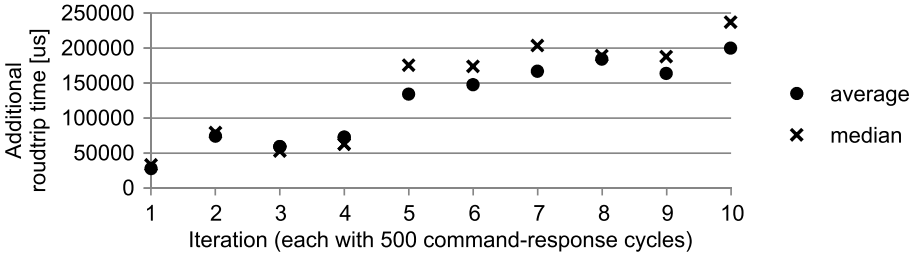


Figure 10: Average and median additional command-response delay (in microseconds) induced by the persistence mechanism for scenario 2.

significant increase of the additional round-trip time caused by the persistence mechanism over time (on average 19.1 ms and 22.7 ms for average and median respectively). Therefore, frequent massive changes of applet state cause a significant degradation of performance over time. The source of this performance loss is the increased memory consumption due to frequent changes of the in-memory copy of the persistent application state that cause many new object state representation objects to be created. Moreover, the unused object state representation objects that were created in preceding persistent state collection cycles likely result into additional effort being put into garbage collection by the Java virtual machine. While this may lead to problems with long debugging sessions, it would not cause problems for typical prototyping use-cases of secure element applications. In these cases, it can be expected that actual command-response sequences with Java Card applications are short (compared to the hundreds of queries performed in this test) and that the virtual machine container is stopped and restarted in between many of the command-response sequences causing a cleanup of the VMs memory.

For rapid-prototyping and debugging sessions of typical NFC secure element applications (like payment applets, password managers or password verifiers) with only a low number of exchanged command-response pairs, additional delays in the range of 10 to 100 ms per command can be expected. This may accumulate to several hundred milliseconds per transaction. As contactless transactions in the context of payment and ticketing applications (cf. [1,6]) may usually take 300 to 500 ms and contactless smartcard standards (cf. [10]) allow for delays up to several seconds, we assume that such an additional delay will be acceptable in the context of rapid-prototyping and debugging. Moreover, it can be expected that many operations carried out by smartcard applications (e.g. complex calculations, iterations over arrays, etc.) perform significantly better on a mobile device CPU (or a PC CPU) than on a resource-constrained smartcard micro-controller.

9 Conclusion

In this paper, we showed that there are significant differences between the Java Card virtual machine and other VMs. These differences cause problems with scenarios where Java Card applications are emulated on top of non-Java Card VMs. However, we found that it is possible to overcome the problems caused

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

by different virtual machine life-cycles by adding a persistence framework to the Java and Dalvik virtual machines. Our solution extracts the state of Java Card applications and creates an in-memory snapshot of the application state. This snapshot can be stored to persistent memory and later be re-implanted into the application (recreating all objects, references and primitive values). Moreover, it can be used to roll back application state to the previously saved snapshot version. Both capabilities were not present in jCardSim.

We integrated our persistence framework with the open-source Java Card simulator jCardSim. Based on this simulator environment, we measured the performance of our in-memory state extraction mechanism for both a minimal Java Card applet example and a typical Java Card applet. We found that, while the persistence framework induces a significant overhead in processing time and memory consumption, the extra delays are still sufficiently low enough to use the framework in our Java Card emulator environment for in-place testing, debugging and rapid prototyping (cf. [15]).

Nevertheless, while our approach adds a first implementation of Java Cards transaction roll-back mechanism to jCardSim, we have not verified this transaction mechanism for all corner-cases and expect that some of these cases are not covered by our current implementation. Therefore, further research needs to analyze and address those corner cases.

Acknowledgments

This work is part of the project "High Speed RFID" within the EU program "Regionale Wettbewerbsfähigkeit OÖ 2007–2013 (Regio 13)" funded by the European regional development fund (ERDF) and the Province of Upper Austria (Land Oberösterreich).

Moreover, this work has been carried out in cooperation with "u'smile", the Josef Ressel Center for User-Friendly Secure Mobile Environments, funded by the Christian Doppler Gesellschaft, A1 Telekom Austria AG, Drei-Banken-EDV GmbH, LG Nexera Business Solutions AG, and NXP Semiconductors Austria GmbH.

References

[1] Transit and Contactless Open Payments: An Emerging Approach for Fare Collection. White Paper TC-11002, Smart Card Alliance Transportation Council, Nov. 2011.

[2] Android Open Source Project. *Host-based Card Emulation*, 2013. `http://developer.android.com/guide/topics/connectivity/nfc/hce.html`.

[3] D. Barry and T. Stanienda. Solving the Java Object Storage Problem. *Computer*, 31(11):33–40, Nov. 1998.

[4] R. G. G. Cattell and D. K. Barry, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, 1997.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

[5] M. Dudarev. jCardSim – Java Card is simple! Presentation at JavaOne Russia, Apr. 2013. http://jcardsim.org/sites/default/files/CON1160.pdf.

[6] EMVCo. *EMV Contactless Specifications for Payment Systems – Book A: Architecture and General Requirements*, Mar. 2011.

[7] Gemalto. *Simulation Suite V2.6 – Getting Started*, Oct. 2010. http://www.gemalto.com/products/simulation_suite/download/GettingStarted.pdf.

[8] Gisecke & Devrient. *JCS Suite.* Technical white paper, July 2010. http://www.gi-de.com/gd_media/media/en/documents/brochures/mobile_security_2/cspa/Technical_white_paper_JCS_Suite_v30.pdf.

[9] M. Hölzl, R. Mayrhofer, and M. Roland. Requirements for an Open Ecosystem for Embedded Tamper Resistant Hardware on Mobile Devices. In *Proceedings of the 11th International Conference on Advances in Mobile Computing & Multimedia (MoMM2013)*, pages 249–252. ACM, Vienna, Austria, Dec. 2013.

[10] International Organization for Standardization. *ISO/IEC 14443-3: Identification cards – Contactless integrated circuit cards – Proximity cards – Part 3: Initialization and anticollision*, 2011.

[11] C. McManis. The basics of Java class loaders. *Java Indepth*, Oct. 1996. http://www.javaworld.com/article/2077260/learn-java/the-basics-of-java-class-loaders.html.

[12] Oracle. *Java Platform, Standard Edition 7: API Specification*, 2013.

[13] A. Rashid and R. Chitchyan. Persistence as an aspect. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development (AOSD)*, pages 120–129. Boston, MA, USA, 2003.

[14] M. Roland. Software Card Emulation in NFC-enabled Mobile Phones: Great Advantage or Security Nightmare? In *4th International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Phone Use*. Newcastle, UK, June 2012.

[15] M. Roland. Debugging and Rapid Prototyping of NFC Secure Element Applications. In *Mobile Computing, Applications, and Services*, LNICST. Springer, Paris, France, Nov. 2013.

[16] M. Roland, J. Langer, and R. Mayrhofer. (Ab)using foreign VMs: Running Java Card Applets in non-Java Card Virtual Machines. In *Proceedings of the 11th International Conference on Advances in Mobile Computing & Multimedia (MoMM2013)*, pages 286–292. ACM, Vienna, Austria, Dec. 2013.

[17] SIMalliance. *Open Mobile API specification V2.03*, June 2012.

[18] Sun Microsystems, Inc. *Java Card Platform: Development Kit User's Guide, Version 2.2.2*, Mar. 2006.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

[19] Sun Microsystems, Inc. *Java Card Platform: Runtime Environment Specification, Version 2.2.2*, Mar. 2006.

[20] Sun Microsystems, Inc. *Java Card Platform: Virtual Machine Specification, Version 2.2.2*, Mar. 2006.

[21] G. Watson. *ORMLite Package, Version 4.45*, Mar. 2013. `http://ormlite.com/docs/ormlite.pdf`.

[22] D. Yeager. Added NFC Reader support for two new tag types: ISO PCD type A and ISO PCD type B. Patches to the CyanogenMod aftermarket-firmware for Android devices, Jan. 2012. `https://github.com/CyanogenMod/android_packages_apps_Nfc/commit/d41edfd794d4d0fedd91d561114308f0d5f83878`.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036

Author biographies

Michael Roland is a post-doc researcher at the NFC Research Lab Hagenberg (University of Applied Sciences Upper Austria), Austria. His main research interests are NFC, security and Android. He is the creator of NFC TagInfo, one of the most successful NFC developer tools for Android devices, and co-author of the book "Anwendungen und Technik von Near Field Communication (NFC)". He holds a B.Sc. and a M.Sc. degree in Embedded Systems Design (University of Applied Sciences Upper Austria, 2007 and 2009) and a Ph.D. (Dr. techn.) degree in Computer Science (Johannes Kepler University Linz, Austria, 2013).

Josef Langer has been working in the smartcard, RFID and NFC industry area for more than 15 years. He studied electrical engineering at Vienna University of Technology, Austria, and RWTH Aachen, Germany, and received his Ph.D. (Dr. techn.) in Computer Sciences from the Johannes Kepler University in Linz, Austria. Langer has been professor for microprocessor engineering at the University of Applied Sciences Upper Austria since January 2003. He is head of the NFC Research Lab Hagenberg and head of the Research Group Embedded Systems at his university. He is author of more than 40 publications and co-author of the first NFC book.

René Mayrhofer currently holds a full professorship for Mobile Computing at Upper Austria University of Applied Sciences, Campus Hagenberg, Austria. Previously, he held a guest professorship for Mobile Computing at University of Vienna, Austria, during which he received his venia docendi for Applied Computer Science. His research interests include computer security, ubiquitous computing, and machine learning, which he brings together in his research on intuitive and unobtrusive techniques for securing spontaneous interaction. He received his Dipl.-Ing. (MSc) and Dr. techn. (PhD) degrees from Johannes Kepler University Linz, Austria, and subsequently held a Marie Curie Fellowship at Lancaster University, UK.

International Journal of Pervasive Computing and Communications, 10 (3), pp. 291–312, 2014.
https://doi.org/10.1108/IJPCC-06-2014-0036