(Ab)using foreign VMs: Running Java Card Applets in non-Java Card Virtual Machines

Michael Roland
NFC Research Lab Hagenberg
University of Applied Sciences Upper Austria
Softwarepark 11, 4232 Hagenberg, Austria
michael.roland@fh-hagenberg.at

Josef Langer
NFC Research Lab Hagenberg
University of Applied Sciences Upper Austria
Softwarepark 11, 4232 Hagenberg, Austria
josef.langer@fh-hagenberg.at

René Mayrhofer
Josef Ressel Center u'smile
University of Applied Sciences Upper Austria
Softwarepark 11, 4232 Hagenberg, Austria
rene.mayrhofer@fh-hagenberg.at

ABSTRACT

Creating Java Card applications for Near Field Communication's card emulation mode requires access to a secure smartcard chip (the secure element). Today, even for development purposes, it is difficult to get access to the secure element in most current smart phones. Therefore, it would be useful to have an environment that emulates a secure element for rapid prototyping and debugging. Our approach to such an environment is emulation of Java Card applets on top of non-Java Card virtual machines (e.g. Android's Dalvik VM). However, providing a Java Card run-time environment on top of another Java virtual machine faces one big problem: The Java Card virtual machine's operation principle is based on persistent memory technology. As a result, the VM and the applications that run on top of it have a significantly different life-cycle compared to other Java VMs. Based on specific scenarios for secure element emulators for the Android platform, we evaluate these differences and their impact on Java VM-based Java Card emulation. Further, we propose possible solutions to the problems that arise from these differences in the life-cycles.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.11 [**Software Engineering**]: Software Architectures; D.2.m [**Software Engineering**]: Miscellaneous

General Terms

Algorithms, Experimentation, Verification

MoMM2013, 2-4 December, 2013, Vienna, Austria

Keywords

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

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 application-specific software systems. Today, however, many smartcards follow an open approach: The smartcard's 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 (typically smartphones). NFC's 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 (cf. [4]).

Besides the issue of deploying application prototypes, another issuer 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 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, developers seek 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.

In the context of Android-based mobile devices, a Java Card emulator could be built on top of the system's native Dalvik Java virtual machine (cf. [5]). 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 the software card emulation mode which is available on some Android-based devices (cf. [4, 5]).

The main benefit of using a standard Java virtual machine for emulation is the seamless integration with existing debuggers. For instance, the Dalvik Java 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 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 life-time 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. We discuss possible solutions to the problems that arise from these differences in the life-cycles. Finally, we propose an implementation for persisting Java Card application state in non-Java Card virtual machines. Our solution uses Java's reflection API to collect information about classes and objects, and to store and later reconstruct the persistent objects of a Java Card application. We implemented a working prototype for Java SE and Android that fulfills all qualities that we need for persisting Java Card applications. The prototype uses a simple XML file as its database for storing object state.

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 [7]. The run-time environment consists of a Java Card virtual machine (as defined in [8]), a Java Card specific API and Java Card specific security features.

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's lifetime is equivalent to the lifetime of the smartcard microchip [8]: The VM is installed and started during the manufacturing process and terminates when the chip is destroyed. In between, the VM's 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).

An applet instance is the main entry point of a Java Card application. An applet provides several public methods for interaction with the Java Card run-time 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. Similar 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 guaran-

teed 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.

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's Java Card Simulation Suite and Gemalto's Simulation Suite). Moreover, there exist Java Card simulators that run on top of standard Java virtual machines (e.g. Java Card Workstation Development Environment and jCardSim). Unfortunately, non of these simulators are known to integrate with current smartphone operating systems (specifically with the Android platform). Also, non 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. [5]). 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 JC Emulator for the Android Emulator

The idea behind this concept is to use the existing open-source Java Card run-time environment simulator implementation *jCardSim*[1] 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.

Fig. 1 shows our scenario for integrating the Java Card emulator with the Android emulator. Android's Open Mobile API-based secure element API (cf. [6]) 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.

[1] `http://jcardsim.org/`

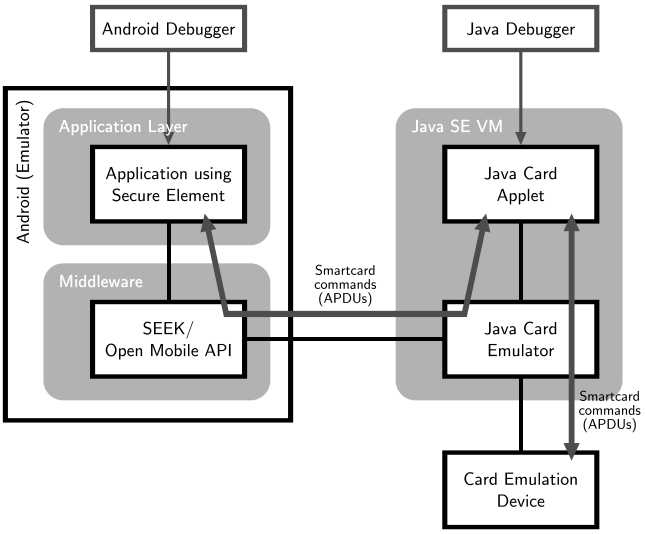

Figure 1: A Java Card emulator attached to the Android platform emulator [5].

3.2 JC 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 Java virtual machine. Again, an interface to access the emulator is added as a terminal interface to Android's Open Mobile API-based secure element API implementation. Moreover, the emulator is connected to the software card emulation API (sometimes also called "soft-SE" or "host card emulation") that is available for some Android devices (cf. [4, 10]).

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 Java application's last execution path 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, 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.

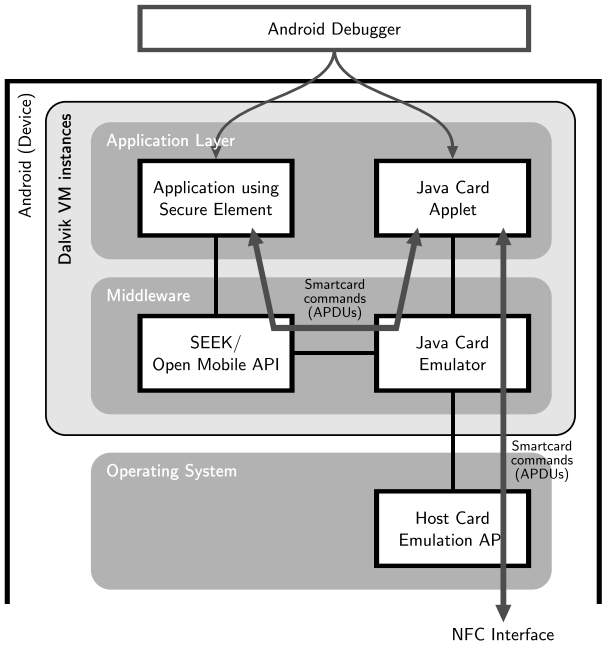

Figure 2: A Java Card emulator integrated into an Android device [5].

4. APPLICATION AND VM 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's 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 the 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 restarts of the host operating system or the Java Card emulator's Java VM would result into loss of Java Card applets' application state.

Nevertheless, for prototyping and long-term tests (particularly in combination with an external card emulation device), it would be interesting to preserve the Java Card applications' 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, Java's serialization mechanism typically requires modifications to the Java Card application's source code.

A 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 used 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 Java Card's 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 containing virtual machine 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, Android's state recovery is not comparable to the Java Card VM's 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.

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 [7], applet instances (i.e. objects instantiated

from an applet class) and objects referenced from a persistent object's field or from a class's static field are persistent.

However, preserving application state in non-Java Card virtual machines requires manual implementation. Typically, this would mean 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 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 [1]. 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. Also, it is not possible to automatically serialize or deserialize all static members of a class. It is also impossible to only include specific parts of an object graph into serialization. Moreover, Java requires serializable classes to be tagged as *serializable*. Thus, modifications to the classes' source code are necessary.

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 [3] describe how aspects can be used to add persistence to existing classes. However, their implementation requires these classes to fulfill 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.

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[2], Apache Cayenne[3], Hibernate[4] and ORMLite[5]). 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. [9]). Overall, ORM seems to require detailed knowledge of an application's 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 [1]. While ODMG bindings do not require modifications to the persistent classes' source code, 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 [2]. Thus, they would have a significant impact on source-level debug-ability.

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.
- No modifications to the source code of the Java Card classes should be required.
- No pre- or post-processing should be required in order to maintain source-level debug-ability with existing tools.
- The solution should work for both, Java SE and Android.

We, therefore, decided to create our own prototypical 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[6] library which permits instantiation of Java objects without calling a class's constructor.

Our implementation collects object state by starting at defined root nodes (the Java Card applet instances that are known to the Java Card run-time environment) and by iterating through every non-static field declared in the objects' classes and super-classes:

```
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();
```

[2] https://java.net/projects/activeobjects/
[3] http://cayenne.apache.org/
[4] http://www.hibernate.org/
[5] http://ormlite.com/
[6] http://objenesis.org/

```
      String fieldName = objectClass.getName() +
                         "#" + field.getName();

      ObjectState fieldState =
              process(fieldValue, fieldType);
      objectState.addField(fieldName, fieldState);
    }
    objectClass = objectClass.getSuperclass();
  }
}
```

For each discovered object, an *object state representation* is created. The object state representation indicates whether the field is a primitive value or a reference to an object or to an array. The state representation of an object maps each field's name to the object state representation of the field's value or referenced object. The state representation of an array maps each array entry to its object state representation. The state representation of a primitive value contains the primitive value.

In order to prevent duplication or duplicate processing of referenced objects, a map is created that contains a unique identifier for each reference as a key and the associated object state representation as its values. References that were previously found while iterating through the object graph are skipped from further processing. For each newly discovered object, an entry is added to that map. Consequently, the object graph is reconstructed as a graph of object state representations.

In order to get a list of all classes of the Java Card applications that could potentially contain static fields with data that needs to be persistent, a list of relevant classes has to be manually created. Based on that list, our implementation iterates through each declared static field and collects the associated object state representation:

```
for (Field field : clazz.getDeclaredFields()) {
  field.setAccessible(true);
  if (Modifier.isStatic(field.getModifiers())) {
    Class fieldType = field.getType();

    if (!(Modifier.isFinal(field.getModifiers()) &&
          fieldType.isPrimitive())) {
      Object fieldValue = field.get(object);
      String fieldName = objectClass.getName() +
                         "#" + field.getName();

      ObjectState fieldState =
              process(fieldValue, fieldType);
      classState.addField(fieldName, fieldState);
    }
  }
}
```

The classes used for state representation of objects and classes' static fields were designed so that they can easily be serialized to and deserialized from XML. The resulting XML file is used a the back-end database for persisting object state:

```
<References>
  <ObjectState hashCode="1106723384"
               fieldType="test.Car">
    <Fields>
      <Field name="test.Car#wheels"
             hashCode="1106727376" />
      (...)
    </Fields>
  </ObjectState>
  <ArrayState hashCode="1106727376"
              fieldType="[Ltest.Wheel;">
    <Elements elementType="test.Wheel">
      <Element hashCode="1106728104" />
      <Element hashCode="1106730936" />
      <Element hashCode="1106733208" />
      <Element hashCode="1106733208" />
    </Elements>
  </ArrayState>
  <ObjectState hashCode="1106728104"
               fieldType="test.FrontWheel">
    <Fields>
      <Field name="test.FrontWheel#typeCode"
             hashCode="1107456976" />
      <Field name="test.Wheel#pressure"
             hashCode="1107435728" />

      (...)

    </Fields>
  </ObjectState>
  <PrimitiveValue hashCode="1107456976"
                  fieldType="java.lang.Short">
    <Value primitiveType="eShortPrimitive">
      2
    </Value>
  </PrimitiveValue>

  (...)

</References>

<Classes>

  (...)

  <ClassState className="test.Wheel">
    <Fields />
  </ClassState>
  <ClassState className="test.FrontWheel">
    <Fields />
  </ClassState>
  <ClassState className="test.Car">
    <Fields>
      <Field name="test.Car#CAR"
             hashCode="1106723384" />
    </Fields>
  </ClassState>
</Classes>

<Objects>
  <Object name="testObject1"
          hashCode="1106723384" />
</Objects>
```

After deserialization of the state representation from the XML file, the original object network can be re-created. First, an instance of the class that is represented in the object state is created using the Objenesis library:

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

This avoids calling an object's constructor and permits easy instantiation of classes that do not have a no-argument constructor. As 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 an object:

```
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);
}
```

As a result, our implementation fulfills all five qualities that we expected from our solution.

However, our solution is still not ideal. Currently, it is only possible to save and restore application state as a whole. Moreover, serialization and deserialization using an XML file on the file system is relatively slow. This may result in a performance issue when emulating the Java Card transaction mechanism: Every time a transaction is started the whole application state needs to be captured. Similarly, rollback of a transaction requires implantation of the whole application state.

7. 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 by different virtual machine lifecycles by adding a persistence framework to the Java and Dalvik virtual machines. Our solution can extract the state of Java Card applications and store it to persistent memory. Later, this persisted state can be re-implanted into the application (recreating all objects, references and primitive values). Nevertheless, our solution is not ideal. Currently, our solution is only capable of extracting and reimplanting application state as a whole, does not perform any caching and is not capable of storing or reverting only modified fields. This, however, has a big impact on performance when it comes to modeling Java Card's transaction mechanism: Whenever a transaction is aborted, the state of the whole application has to be reverted. Therefore, future research should focus on optimization for this scenario.

We implemented and tested our concept using a simple class hierarchy and object network to model our application state. In the future, we intend to integrate our implementation together with jCardSim in our emulator scenarios.

8. 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.

9. REFERENCES

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

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

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

[4] 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.

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

[6] SIMalliance. *Open Mobile API specification*, June 2012.

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

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

[9] G. Watson. *ORMLite Package, Version 4.45*, Mar. 2013.

[10] 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.