

- 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).

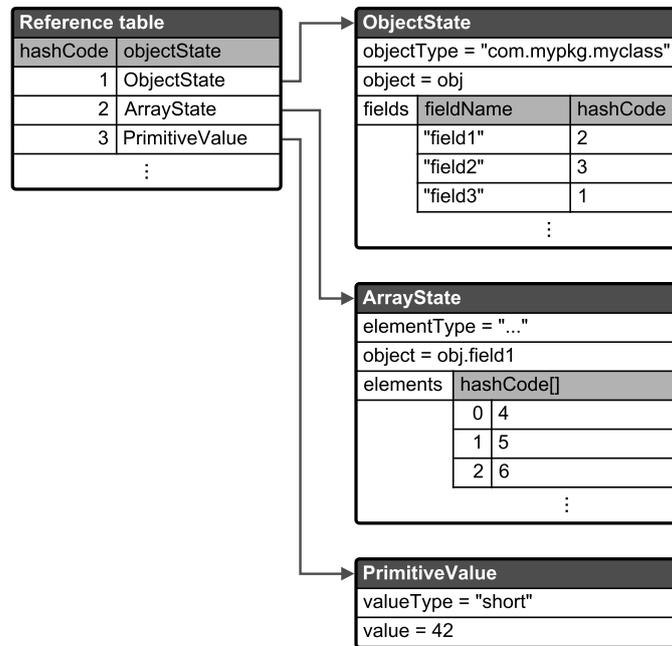
In order to update the state representation of a (potentially) modified object and its connected partial object graph, our implementation starts at the state representation of that object and checks if all fields still match the recorded references. If changes are discovered, links are updated and wrapper 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).

obj : com.mypkg.myclass	
field1	: ...[]
field2	: short = 42
field3	: com.mypkg.myclass = obj
	: ...

(a) Example object *obj*



(b) In-memory object state representation

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

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.

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).

```

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.

```

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

```

<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

```

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.

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.

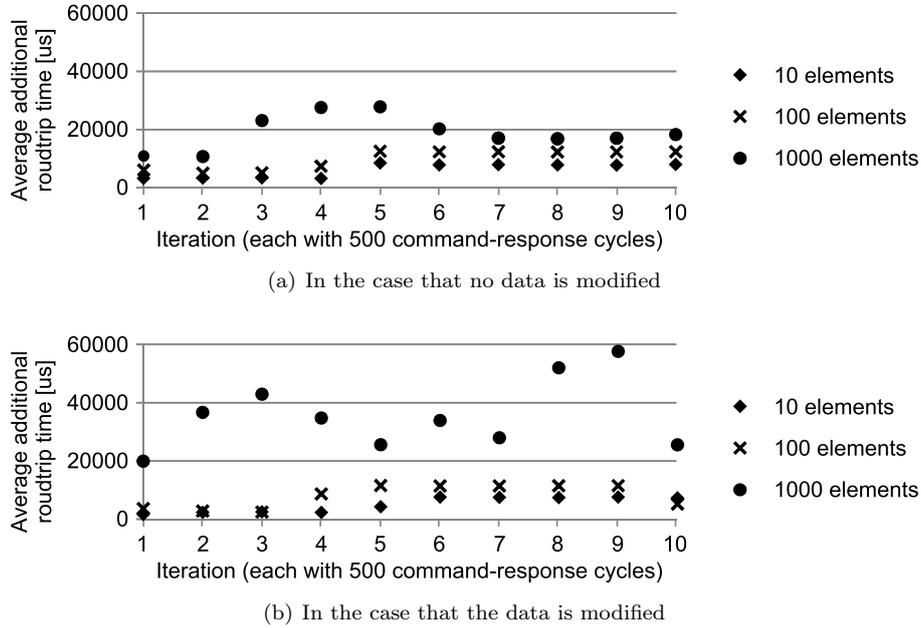


Figure 9: Average additional command-response delay (in microseconds) induced by the persistence mechanism for scenario 1 for array sizes of 10, 100 and 1000 elements.

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

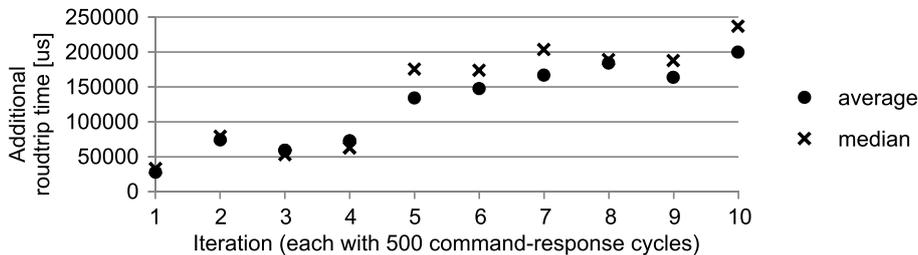


Figure 10: Average and median additional command-response delay (in microseconds) induced by the persistence mechanism for scenario 2.

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

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.

- [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.

- [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.

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.