Debugging and Rapid Prototyping of NFC Secure Element Applications

Michael Roland

NFC Research Lab Hagenberg
University of Applied Sciences Upper Austria
michael.roland@fh-hagenberg.at

Abstract. The ecosystem behind secure elements is complex and prevents average developers from creating secure element applications. In this paper we introduce concepts to overcome these issues. We develop two scenarios for open platforms emulating a secure element for the Android platform. Such an open emulator can be used for debugging and rapid prototyping of secure element applications. Moreover, by trading the secure element's security and trust for openness, such a platform can be used as a replacement for the secure element for long-term testing and for showcasing of applications.

Keywords: Near Field Communication, Secure Element, Java Card, Rapid prototyping, Debugging, Testing.

1 Introduction

Since Google added support for Near Field Communication (NFC) technology to their Android operating system and to their Nexus devices in 2010, the number of smart phones equipped with NFC functionality is rapidly increasing. This growing availability also boosts many developers' interest in developing NFC applications. Particularly card emulation is an often demanded feature. Card emulation is the ability of an NFC device to interact with existing contactless smartcard reader infrastructures. Therefore, an NFC-enabled smart phone in card emulation mode could replace smartcards in payment, access control, identification and ticketing applications. Especially payment use-cases and their potential for generating high revenues boost the developers' demand for NFC card emulation [8].

The central component of card emulation is the secure element. The secure element is a smartcard microchip that is embedded (fixed or removable) into the NFC device. This chip is a tamper proof hardware platform that provides highly secure storage and a trusted execution environment. Modern secure elements (just like many other modern smartcards) contain a Java Card run-time environment. Thus, developers can create applications independent of the actual secure element hardware using Java Card's subset of the Java programming language.

G. Memmi and U. Blanke (Eds.): MobiCASE 2013, LNICST 130, pp. 298–313, 2014.

While the secure element's high levels of security and trust are important requirements for many application scenarios, they also form an enormous obstacle for many developers [8]: The secure element ecosystem is complex and has several different players. Secure elements are usually controlled by their issuer (e.g. the handset manufacturer for an embedded secure element in a mobile phone or the mobile network operator for a UICC-based secure element). Issuers often delegate the management of their secure elements and the applications on them to a trusted service manager (TSM). The trusted service managers act as connectors between providers of secure element applications and the secure elements of a large number of users.

Besides acting as a connector, a TSM will typically also act as a filter: To maintain the security and the trusted state of the secure element, a TSM has to assure the quality and the security of applications deployed to the secure element. Thus, secure element applications will most likely have to undergo some form of (costly) security certification. Moreover, the available space on a secure element is very limited. Therefore, the TSM has to choose which applications to deploy to a particular secure element. This may result in a high cost for application developers and service providers to deploy their applications to secure elements. It may even cause applications that compete with services provided by a secure element's owner or by large service providers to be blocked completely from some secure elements (cf. [1, 2]). Currently, many secure element issuers do not even provide development access to their secure elements for average developers.

Therefore, the barriers towards getting an application into a secure element (even for development purposes) are very high. Nevertheless, developers seek simple ways to develop and test their applications (Java Card applets) for secure elements. An ideal debugging and prototyping environment would permit the applets to be tested in-place with other application components while allowing source-level debugging of the applets' code.

In the context of the secure element in an NFC-enabled smart phone, in-place debugging would mean that the secure element (Java Card) would be emulated in software. This emulator would then be accessible to apps through regular secure element APIs, as well as to external smartcard readers, through the contactless NFC interface. Besides source-level debugging, such a scenario would also provide an environment for rapid prototyping of secure element applications without access to an actual secure element. I.e. the emulator could be used as a drop-in replacement for a secure element that provides the same functionality while trading security for openness. Thus, on the one hand, this environment would not provide the high level of security of a smartcard chip, but, on the other hand, would be open to all developers.

In this paper, we describe the conceptual design of such an open secure element development environment for Android devices. We show how such a secure element emulator could be integrated in both, an actual Android smart phone and an Android device emulator.

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 [12]. The run-time environment consists of a Java Card virtual machine (as defined in [13]), a Java Card specific API and Java Card specific security features. In this paper, we refer to Java Card version 2.2.2. While this version is used in many current secure elements, it is not the most recent Java Card version. Our decision is based on the fact that – while Java Card version 3.0 is fully specified since 2009 – it is still unclear when commercial products (particularly secure elements) based on that new and enhanced version of Java Card will hit the mass market.

2.1 Language and API

Java Card has been optimized for tiny embedded devices with limited memory and processing power. Therefore, many features of Java are unavailable in Java Card to assure that applications have a small footprint that matches the constrained resources of a smartcard. For instance, Java Card only supports the primitive data types *boolean*, *byte*, *short* and optionally *int*. The types *char*, *float*, *double*, *long* and the wrapper classes for primitive data types are not supported. Moreover, some language constructs like enumerated types, enhanced *for*-loops for array iteration or variable-length argument lists do not exist in the Java Card language. As a result of the constrained smartcard environment and the command-response interaction with smartcards, Java Card does not offer multi-threading capabilities. Furthermore, most of the core API classes of the Java language are unsupported. Specifically, only classes related to exceptions and the class *Object*, as a common root for the class hierarchy, are available.

Besides these limitations, the Java Card run-time environment provides an API with smartcard-specific classes. This API comprises of classes related to the interaction with the Java Card run-time environment, the structure and operation of Java Card applications and the processing of ISO/IEC 7816-4 smartcard commands (application protocol data units, APDUs). Moreover, the API contains classes related to smartcard-specific security functionality (e.g. management of PIN codes and cryptographic keys, encryption, decryption, and computation of checksums, cryptographic hashes and digital signatures based on various algorithms). Additionally, the Java Card API contains further (partly optional) helper classes for smartcard-related tasks.

2.2 Virtual Machine

The Java Card virtual machine is the execution environment for the Java Card runtime environment and for all Java Card applications. In comparison to other Java virtual machines, the Java Card virtual machine runs throughout a smartcard's lifetime. The virtual machine and all applications persist across power-cycles. This is possible because the code and data memory is backed by persistent storage

technologies. Protection against data corruption due to unexpected power-cycles is achieved with an atomic transaction mechanism.

2.3 Security

Besides the different life-cycle, another major difference between Java and Java Card is the security architecture. Java Card has been designed for security critical applications. Particularly, Java Card introduces a strict separation between the contexts of installed applications. Thus, applications cannot access each other's objects (data) without being granted explicit permissions. It has to be noted, though, that the strict firewall between application contexts seems to be weakened by the lack of an on-card byte code verifier on many platforms (cf. [3, 6]) and, therefore, often relies on a trusted installation process including security evaluation of applications prior to their deployment. This issue also complicates developers' and service providers' access to the secure element.

2.4 Applications and Life-Cycle

The main entry point of a Java Card application is an applet (i.e. an instance of a class that extends the *Applet* class). An applet implements the central interface to the Java Card run-time environment that is used to control the application's life-cycle. 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 virtual machine, Java Card applets execute forever (or until they are explicitly uninstalled). However, applet instances return to the suspended state upon power loss or reset.

3 Java Card Simulators

Several Java Card simulators exist which allow simulation and testing of Java Card applets without real smartcard hardware. These simulators and simulation environments can be divided into three different categories:

1. reference implementations of the Java Card virtual machine and the Java Card run-time environment,
2. smartcard simulators provided by manufacturers for their smartcard products, and
3. general-purpose Java Card simulators that operate on top of Java virtual machines.

Sun's (now Oracle's) Java Card reference implementation is an example for the first category. Regarding debugging and simulation capabilities, this type of simulation environment is comparable to a regular smartcard. It processes compiled Java Card applications and interacts through smartcard commands. For instance, the Java Card reference implementation integrates with Java ME's secure element API. However, it does not offer source-level debugging capabilities of Java Card applications.

An example for the second category are G&D's Java Card Simulation Suite and Gemalto's Simulation Suite. These smartcard manufacturers provide their custom Java Card simulation environments that simulate their specific smartcard architectures and can be used to debug Java Card application prior to deployment to real cards.

The Java Card Workstation Development Environment (JCWDE) and the open-source jCardSim are examples for the third category. These simulators emulate the Java Card run-time environment on top of a standard Java virtual machine. Thus, instead of compiling Java Card applications to Java Card byte code, they are compiled to Java byte code. Both simulators offer source-level debugging for applets based on standard Java debugging tools. However, both implementations lack some features of a full Java Card run-time environment. While the JCWDE provides integration with Java ME's secure element API and direct interaction through APDU scripts, jCardSim offers support for APDU scripts and an interface to the Java Smart Card IO API.

4 Towards a Secure Element Emulator

The available simulators provide a good starting point for developing Java Card applications. However, especially with current NFC-enabled smart phones, developers would be interested in more advanced debugging mechanisms. An ideal debugging environment would offer the following capabilities:

1. a complete (or as much complete as possible) Java Card run-time environment,
2. source-level debugging capabilities, and
3. in-place testing and debugging together with other application components.

In the context of the secure element in an NFC-enabled smart phone, for instance, in-place debugging would mean that the secure element emulator (Java Card emulator) is accessible to apps through regular secure element APIs, as well as to external smartcard readers through a contactless NFC interface.

In such a scenario, Java Card applications can be tested and debugged while they are communicating with apps on the smart phone or while they are communicating with external smartcard reader devices over the contactless interface. Source-level debugging capabilities can come handy at this stage to trace the execution path through the Java Card application during actual communication. Emulating the Java Card run-time environment on top of a regular Java virtual

machine permits using standard Java debugging tools. In comparison, creating a custom Java Card virtual machine would also require to create custom debugging tools that interact with that VM.

Besides source-level debugging capabilities, such a scenario would also provide another advantage: A Java Card emulator that can be used as a drop-in replacement for a secure element in a smart phone would provide an environment for rapid prototyping of secure element applications without access to an actual secure element. Thus, developers and service providers would have an open tool to showcase their secure element applications bypassing the need for a real secure element. While the emulator would operate at a much lower security level (no dedicated smartcard hardware, no protections by a Java Card virtual machine, etc.), it would be an open platform that provides comparable functionality to a regular secure element and that is available to all developers without the restrictions of the complicated and closed ecosystem of a regular secure element chip.

5 Implementing a Secure Element Emulator for Android

We chose Android as the target platform for the secure element emulator. The main reasons for this decision are:

1. Android is an open-source system. This provides a detailed insight into system's internal structures and even makes modification on the system level fairly simple.
2. Android uses a Java-based run-time environment. This is a good starting point for Java Card emulation.
3. An Android implementation of the Open Mobile API, a standardized secure element API, exists. Though this API has not been integrated in main-line Android, manufacturers already integrated it in many NFC-enabled Android devices.
4. Android has a large market share and, consequently, many requests for card emulation capabilities and secure element access focus on the Android platform.

Our vision is that the secure element emulator would integrate into an Android device the same way as any other secure element. Therefore, Java Card applets running in the emulator should be accessible by apps on the mobile device through the secure element API as well as by external smartcard readers through card emulation. As a result, the emulator would become an open drop-in replacement for a secure element for testing, debugging, prototyping and showcasing purposes. The openness would, however, come at the price of less (or no) security.

We developed two scenarios for integrating such an open environment into Android:

1. integration of the secure element emulator with the Android emulator and
2. integration of the secure element emulator with an actual Android device.

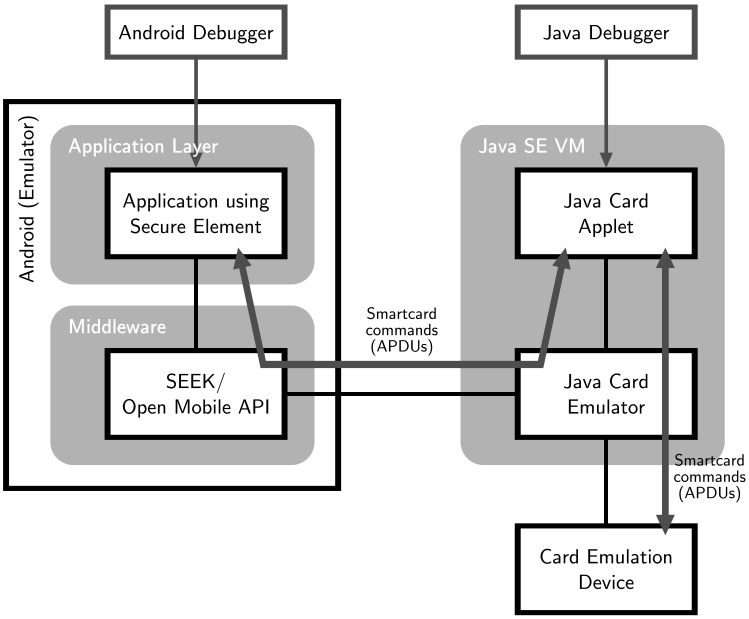

Fig. 1. Java Card emulator attached to the Android platform emulator

5.1 Integration into the Android Emulator

Fig. 1 shows our scenario for integrating the secure element (Java Card) emulator with the Android emulator. We chose a split approach, where the secure element emulator runs separately from the Android emulator. The Java Card emulator and the Java Card run-time environment operate on top of a standard Java SE virtual machine. Therefore, standard Java debugging tools can be used to debug Java Card applets that run inside this environment. Moreover, the separation makes the secure element emulator independent of the Android emulator's life-cycle (e.g. restarts of the emulator, etc.)

Our secure element emulator has two interfaces: One is connected to the Android emulator so that apps can access the emulated secure element and the other is connected to a card emulation device.

In order to connect our open environment to the Android emulator, it listens on a TCP socket for smartcard commands. On the Android side, Android's Open Mobile API-based secure element API (cf. [11]) is extended with a terminal interface that connects to our emulator using a TCP/IP socket. This approach is used by existing secure element simulators (e.g. Java Card reference implementation and JCWDE) too. Thus, if we use the same communication protocol over the TCP socket, we can maintain compatibility to these simulators as well as to the tools that use them.

The second interface connects the emulator to card emulation hardware (e.g. an NFC reader, like the ACR122U, in software card emulation mode). Consequently,

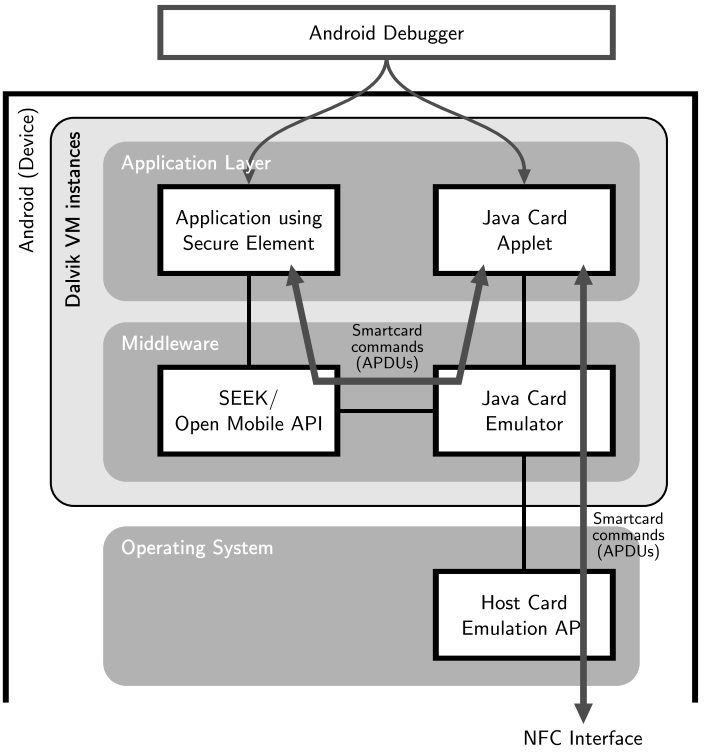

Fig. 2. Java Card emulator integrated into an Android device

external smartcard readers can communicate with Java Card applets running inside the emulator.

5.2 Integration into an Android Device

Fig. 2 shows our scenario for embedding the Java Card emulator into an Android device. In this scenario, the Java Card emulator and the Java Card run-time environment operate on top of the Dalvik virtual machine (Android's version of a Java virtual machine). Therefore, Android's Java debugging tools can be used to debug Java Card applets that run inside this environment. Besides debugging, this scenario can also be used to showcase applications that would normally require a secure element. These applications could use our Java Card emulator platform as an open prototyping environment.

Similar to the scenario with the Android emulator, our secure element emulator has two interfaces: One is connected to Android's secure element API so that apps can access the emulated secure element. In this scenario, the Java Card emulator is directly connected to the Open Mobile API through a terminal interface without the need for a TCP/IP connection. The second interface 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. [8,14]).

5.3 Building a Java Card Run-Time Environment

When implementing a Java Card run-time environment on top of an existing Java virtual machine, the main implementation tasks are the Java Card API and the Java Card application life-cycle management (i.e. installation of, selection of and communication with applets). In order to minimize the implementation effort, we decided to base our emulator on the existing open-source Java Card run-time environment simulator jCardSim[1]. jCardSim already provides an implementation of the application life-cycle management and of many parts of the Java Card API including cryptography and data sharing between applets [4]. However, jCardSim is based only on Java Card version 2.2.1 and still lacks some core functionality. For instance it does not yet support logical channels and atomic transaction processing. Moreover, jCardSim was designed for short simulation cycles and, therefore, does not consider persistence of the run-time environment and the application state across simulation sessions.

jCardSim is currently available for Java SE so it can easily be integrated into the first scenario. For the second scenario, it is necessary to port jCardSim to Android. However, we consider this a minor issue as most of the functionality of Java SE is also available on Android. Moreover, jCardSim's implementation of the Java Card crypto API is based on the Bouncy Castle cryptography API, an API that is included into the Android system.

In order to connect the Java Card run-time environment to the outside world, jCardSim provides an interface that permits registration ("installation") of Java Card applications and dispatching of commands to the registered Java Card applet instances. This interface could be used to attach our emulator environment to Android's secure element API and to the card emulation hardware.

5.4 Integration with the Open Mobile API

In order to access the secure element emulator from Android apps, it needs to be integrated with a secure element API. Main-line Android currently has no standardized secure element API, though it contains a proprietary, undocumented API to access the embedded secure element on some Android devices. However, many device manufacturers added the SEEK-for-Android[2] framework to their Android distributions in order to expose a secure element API on their devices.

SEEK-for-Android is an implementation of the standardized Open Mobile API [11]. Fig. 3 gives an overview of the architecture of this API. The API consists of a transport API that permits APDU-based access to secure elements as well as a service API that provides a higher abstraction level for access to secure element applications. The Open Mobile API can be used for any type of

[1] `http://jcardsim.org/`
[2] `http://code.google.com/p/seek-for-android/`

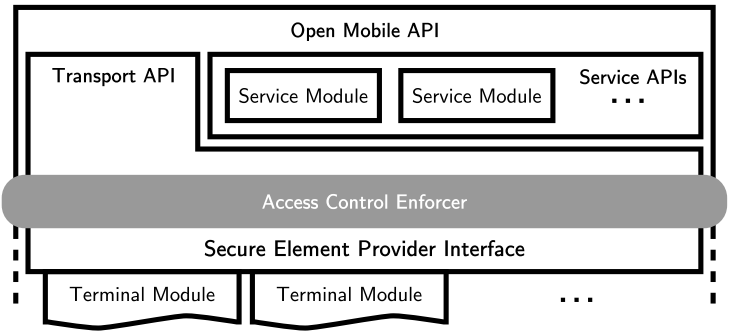

Fig. 3. Architecture of the Open Mobile API [11]

secure element. For each secure element it needs a terminal module that acts as an interface layer between the API and the actual secure element.

A terminal module is a Java class within the namespace *org.simalliance.openmobileapi.service.terminals* that implements an interface consisting of several methods (cf. [10]):

1. `public byte[] getAtr()`
   This method returns the secure element's answer-to-reset (ATR). Our implementation could return a generic ATR like `'3B 85 01 4A 43 45 4D 55 D0'` that indicates the smartcard protocol T=1 and contains the historical bytes "JCEMU".
2. `public String getName()`
   This method returns the secure element's name (e.g. "JCEMU: JavaCard-Emulator")
3. `public boolean isCardPresent()`
   This method indicates if the secure element is currently available and, therefore, must return `true` for our implementation.
4. `public void internalConnect()`
   This method establishes a connection to our emulator and is used for any initialization that should take place before any communication with the secure element. In the Android emulator scenario, this method can be used to establish the TCP socket to the secure element emulator.
5. `public void internalDisconnect()`
   This method closes an open connection to our emulator and is used for cleanup that should take place after all communication with the secure element. In the Android emulator scenario, this method can be used to close the TCP socket to the secure element emulator.
6. `public byte[] internalTransmit(byte[] command)`
   This method is used to pass smartcard commands (APDUs) to the terminal module and, thus, to our emulator. The resulting responses are returned to the calling application.

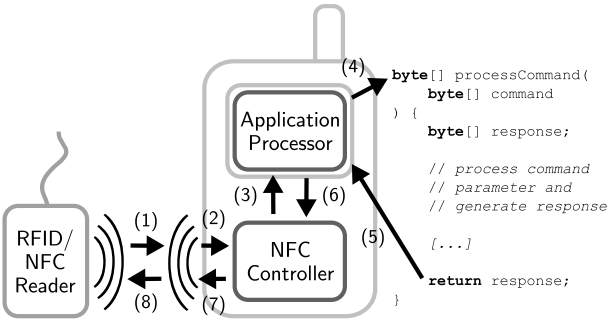

Fig. 4. Communication flow of software card emulation in a mobile phone [8]

7. `public int internalOpenLogicalChannel()`
   This method is used to open a new logical channel to the secure element selecting the logical channel's default applet. The method returns the new logical channel's number.
8. `public int internalOpenLogicalChannel(byte[] aid)`
   This method is used to open a new logical channel to the secure element selecting the applet that matches the given AID. The method returns the new logical channel's number.
9. `public void internalCloseLogicalChannel(int iChannel)`
   This method is used to close a given logical channel.
10. `public byte[] getSelectResponse()`
    This method is used to receive the last response to an applet selection (that was either performed through `internalTransmit` or `internalOpenLogicalChannel`).

5.5 Support for Software Card Emulation

The concept of software card emulation was first brought to mobile phones by BlackBerry (formerly RIM) in their BlackBerry 7 platform [7]. The idea is that the NFC controller acts as a contactless card and forwards all received smartcard commands to an app on the application processor. In turn, responses generated by the app are returned over the contactless interface. Fig. 4 shows the flow of communication in a software card emulation scenario.

Therefore, our secure element emulator would first register for software card emulation with the NFC controller chip. Then, the emulator would receive all smartcard commands received by external smartcard readers and forward them to the Java Card applets for processing. Finally, the response generated within the Java Card environment would be returned through the NFC chip to the smartcard reader.

Using Dedicated Card Emulation Hardware. An example for a device that could be used for card emulation is the ACS ACR 122U NFC reader. It contains NXP's PN532 NFC controller chip. This reader device can be connected to a Java application through the Java Smart Card IO API using PC/SC. Commands for the PN532 that are used to activate card emulation mode and to exchange data during card emulation are wrapped in PC/SC APDU commands.

An explanation on how to put the PN532 NFC chip into card emulation mode is given in [9]. First, the PN532 has to be set up for ISO/IEC 14443 Type A at 106 kbps:

```
PN532.WriteRegister(
    CIU_TxMode,
    TxCRCEn | TxSpeed = 106 kbps | TxFraming = ISO/IEC 14443A);
PN532.WriteRegister(
    CIU_RxMode,
    RxCRCEn | RxSpeed = 106 kbps | RxFraming = ISO/IEC 14443A);
PN532.WriteRegister(CIU_TxAuto, InitialRFOn);
```

Then, the PN532 has to be initialized to PICC (proximity integrated circuit card) mode:

```
PN532.SetParameters(fISO14443-4_PICC | fAutomaticRATS);
PN532.TgInitAsTarget(
    PICCOnly | PassiveOnly,
    MifareParams = { SENS_RES = { 0x00, 0x04 } |
                     NFCID1t = { 0x76, 0x82, 0x4F } |
                     SEL_RES = 0x20 },
    FelicaParams = { 0x00, ..., 0x00 },
    NFCID3t = { 0x00, ..., 0x00 },
    GeneralBytes = { },
    HistoricalBytes = { 0x4A, 0x43, 0x45, 0x4D, 0x55 });
```

Finally, the card emulator listens for commands received on the contactless interface, passes them to the Java Card emulator and sends the responses back over the contactless interface:

```
while (emulating) {
    byte[] command = PN532.TgGetData();
    byte[] response = JCEmulator.process(command);
    PN532.TgSetData(response);
}
```

Using Android. On Android, software card emulation is currently only available in version 9.1 and later of the CyanogenMod aftermarket firmware for Android devices. Elenkov [5] explains how to use software card emulation in apps on the CyanogenMod platform. The basic idea is to register for detection of either an ISO 14443 Type A (IsoPcdA) or Type B (IsoPcdB) smartcard reader:

```
PendingIntent pi = activity.createPendingResult(
    1, new Intent(), 0);
nfcAdapter.enableForegroundDispatch(
    activity, pi,
    new IntentFilter[]{
        new IntentFilter(NfcAdapter.ACTION_TECH_DISCOVERED)
    },
    new String[][]{
        new String[]{ "android.nfc.tech.IsoPcdA" }
    });
```

As soon as a smartcard reader connects to the emulated card, the app gets triggered and can communicate with the smartcard reader. Thus, commands can be received from the smartcard reader and can be passed to the Java Card emulator. In turn, responses from the Java Card emulator can be sent back to the smartcard reader:

```
IsoPcdA isoPcd = IsoPcdA.get(tag);

isoPcd.connect();
byte[] response = new byte[]{ (byte)0x90, (byte)0x00 }

while (emulating) {
    byte[] command = isoPcd.transceive(response);
    response = JCEmulator.process(command);
}

isoPcd.close();
```

6 Developing a First Prototype

Fig. 5 shows the architecture of our first prototype of the secure element emulator platform. Instead of separating the emulator from the apps that access it, both, the app and the emulator environment are combined into one app. The emulator platform consists of a minimal implementation of the Java Card API and the run-time environment. We used an existing Java Card applet of a payment application that we previously used on a Java Card smartcard. We designed our Java Card API and run-time environment implementation so that the applet could be run without any modifications. We added a user interface that communicates with the applet by invoking the emulator environment. Moreover, the user interface allows to switch to external card emulation, where CyanogenMod's software card emulation mode API is used to communicate with the emulated applet.

Our applet worked as expected, both when communicating with the app and when communicating through software card emulation. We were able to do source-level debugging and to single-step through the applet's source code.

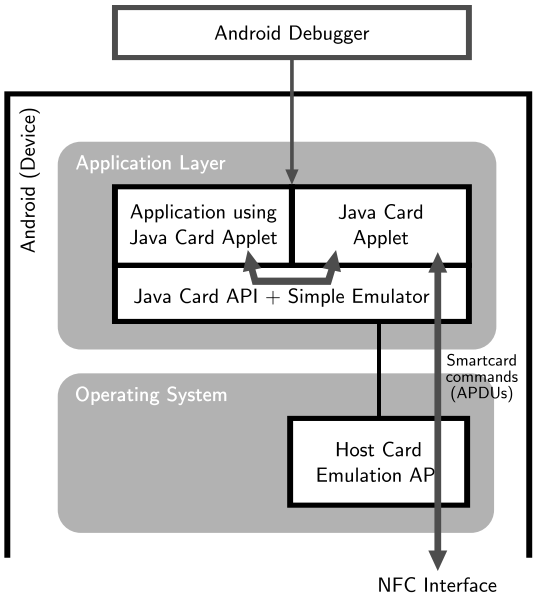

Fig. 5. First prototype for Java Card emulation on Android

However, during implementing and testing of our prototype we found several issues that need to be addressed in future research:

1. We were not able to implement Java Card's atomic transaction mechanism using the available set of APIs on the Android platform. This was not an issue for most transactions, as these transactions would be completed even after the emulated card was torn from the card reader's RF field. The reason is that while an actual smartcard stops all processing and clears its RAM upon power-loss, the emulator platform continues and finishes applet execution. However, we could not implement the case when a Java Card applet tries to intentionally roll-back a transaction. Therefore, a method to roll-back an application's state to a defined boundary is necessary.
2. When our application process (including the emulation environment) is terminated, the state of the Java Card applets is lost. Therefore, as soon as the app starts again, the Java Card applets start from the beginning of their life-cycle. However, for long-term testing and prototyping, the secure element emulator needs to maintain its state beyond application process lifetime and particularly across reboots of the Android device. Therefore, a method to extract and restore application state has to be implemented.

7 Conclusion

In this paper, we showed two scenarios for integrating a secure element emulator into the Android platform. Such a secure element emulator can be used to

test and debug secure element applets written in the Java Card language. Besides source-level debugging, a secure element emulator that is integrated into an Android device and that is accessible through the same interfaces as a regular secure element brings a significant advantage for rapid prototyping and for showcasing of applications. Application developers can design, test and even use their applications with the secure element emulator instead of using a real secure element. While the emulator provides significantly less security than a real secure element, it also avoids the complicated ecosystem of a real secure element. Thus, it could significantly simplify and reduce the cost of development of secure element applications.

Besides our conceptual scenarios, we implemented a working prototype of the secure element emulation system. While our prototype successfully emulated a Java Card applet, we found that there are several issues that need to be resolved in order to build a fully working secure element emulator. These issues particularly focus on the different life-cycle of the Java Card virtual machine in comparison to other Java virtual machines that is introduced by the use of persistent memory for storing application state on smartcards.

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. Balaban, D.: Telcos Close Ranks as Google Threat Looms. NFC Times Blog (July 2011), http://www.nfctimes.com/blog/dan-balaban/telcos-close-ranks-google-threat-looms
2. Balaban, D.: With Launch of Google Wallet, the Wallet War Begins. NFC Times Blog (June 2011), http://www.nfctimes.com/blog/dan-balaban/launch-google-wallet-wallet-war-begins
3. Barbu, G., Giraud, C., Guerin, V.: Embedded Eavesdropping on Java Card. In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) SEC 2012. IFIP AICT, vol. 376, pp. 37–48. Springer, Heidelberg (2012)
4. Dudarev, M.: jCardSim – Java Card is simple! Presentation at JavaOne Russia (April 2013), http://jcardsim.org/sites/default/files/CON1160.pdf
5. Elenkov, N.: Emulating a PKI smart card with CyanogenMod 9.1. Android Explorations (October 2012), http://nelenkov.blogspot.com/2012/10/emulating-pki-smart-card-with-cm91.html

6. Mostowski, W., Poll, E.: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008)
7. RIM: Blackberry API 7.0.0: Package net.rim.device.api.io.nfc.emulation (2011), http://www.blackberry.com/developers/docs/7.0.0api/net/rim/device/api/io/nfc/emulation/package-summary.html
8. Roland, M.: 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), http://www.medien.ifi.lmu.de/iwssi2012/papers/iwssi-spmu2012-roland.pdf
9. Roland, M.: Security Issues in Mobile NFC Devices. Ph.D. thesis, Johannes Kepler University Linz, Department of Computational Perception (January 2013)
10. SEEK for Android: AddonTerminal: How to create an Addon Terminal (May 2012), http://code.google.com/p/seek-for-android/wiki/AddonTerminal
11. SIMalliance: Open Mobile API specification (June 2012)
12. Sun Microsystems, Inc.: Java Card Platform: Runtime Environment Specification, Version 2.2.2 (March 2006)
13. Sun Microsystems, Inc.: Java Card Platform: Virtual Machine Specification, Version 2.2.2 (March 2006)
14. Yeager, D.: 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 (January 2012), https://github.com/CyanogenMod/android_packages_apps_Nfc/commit/d41edfd794d4d0fedd91d561114308f0d5f83878