Radboud University Nijmegen

Bachelor Thesis

AndroidCard

A framework for writing NFC applets on Android

Supervisor:
Erik Poll

Co-supervisor:
Roel Verdult

Kevin Valk - 3052273
June 23, 2013

Abstract

Nowadays you already have different Radio Frequency Identification (RFID) tags in your wallet. An example for such a RFID tag would be a card for paying wireless for public transport or a card that grants you access to buildings or assets and there are much more of such wireless cards. All those card are called RFID tags or in short, tags.

The wireless technique to communicate from a reader to a tag also exists in in nowadays smart phones, but then its called Near Field Communication (NFC). As NFC and RFID works over the same basic technology (radio signals) they are compatible with each other. With a smart phone you can read a tag. But it turns out its very hard to switch theses rolls around. In other words letting the smart phone be a tag and use the smart phone to pay for your public transport.

As research turns out, hardware responsible for NFC inside the current generation smart phones can handle this roll switch. So in theory it should be possible to turn your smart phone into a tag or multiple tags. However is this possible in practice? And if so to what extent?

In this paper I will show that it is indeed possible to emulate tags on your smart phone. To provide developers an easy way to write new applications or port existing applications to Android, I have written a framework called AndroidCard. The framework emulates the JavaCard Operation System (OS). JavaCard is the OS in which one can write applications for tags (also called applets). Porting existing JavaCard applets to Android should be an easy task, as there are currently countless existing applets written in JavaCard. To that end I have crafted AndroidCard to have a lot of equalities with JavaCard. Because of these equalities porting is now possible without needing to do a complete rewrite of the existing applets.

Contents

1 Introduction

Near Field Communication (NFC) is a set of standards for devices to communicate wireless through radio signals, this is better known as Radio-Frequency Identification (RFID) (Shepard, 2005). As NFC and RFID uses the same basic technology (radio signals), NFC and RFID systems are compatible with each other.

The next generation NFC chips have the possibility to do virtually anything regarding NFC, be it peer to peer, card emulation or card reading. These chips are also being used in nowadays smart phones, however most manufacturers that make the Operating Systems (OS) for these devices will not allow the use of card emulation. Why would you ask? For example NFC payment systems have to store privacy sensitive information like credit card numbers in the system. The sensitive information should never be accessible by a third party. This protection is achieved by a so called Secure Element (SE). A SE is a mini computer on a single chip and almost always very good protected against unauthorized software and hardware access. The NFC system can publish such a SE to the outside world. So all communication is handled directly by the SE. This makes it a very secure system.

However it is possible to not publish such a SE and redirect all communication to the OS instead. Where a software application handles the incoming and outgoing communication. This is called host-based card emulation. However the security against unauthorized access through software or hardware in a big OS like Android, pales in comparison with a SE. Thus opening possible security and or privacy exploits (Francis, Hancke, Mayes, & Markantonakis, 2010).

Google, the creator of Android OS, has not given us anything to achieve host-based card emulation. With this feature one could implement there own payment system or merge all his RFID tags (in short tags) into his phone. Currently on Android it is possible to get these features by using CyanogenMod (a modified Android OS). BlackBerry officially supports NFC host-based card emulation through there API.

1.1 Research question

To what extent is it possible to emulate an e-passport with the next generation Near Field Communication enabled smart phones?

For answering this question we introduce some sub-questions that would clarify aspects of the research question:

1. To what extent can we achieve host-based card emulation (See page 5) on Android mobile device with NFC capabilities?
2. How can we use CyanogenMod host-based card emulation?
3. To what extent is it possible to use the host-based card emulation to emulate an e-passport?

Initially when starting with this thesis I had access to a development board with a NFC chip that could be used for host-based card emulation. The board was called 'NPX Evaluation Board v4.1' with the PN544 as its NFC chip. However there was no documentation to achieve host-based card emulation and there was no Software Development Kit (SDK). So back then my research goal was to figure out how we could communicate with the chip to achieve this host-based card emulation. If it was possible, the goal was to write a framework for Android to communicate with the chip. To demonstrate the capabilities of the NFC chip the idea was to build an e-passport card emulation using this new framework.

However on some point in my research I discovered this was already done by CyanogenMod (See page 11). This shifted my research quite a lot. After researching CyanogenMod, it turned out there was little to no documentation and code for writing NFC applets on Android. So I decided to make a framework that would behave like JavaCard to make it easy to port JavaCard applets to Android. I call this new framework 'AndroidCard' because it has so much equalities with the JavaCard. The original idea to demonstrate the power of AndroidCard with an e-passport(Mostowski & Poll, 2010) application still holds.

2 Background

This section explains background information that is useful to understand concepts used further on in this paper. Some subsections can be skipped because of the nature of the end product described in this paper.

Topics discussed are:

- Near Field Communication: General overview about the workings of NFC.
- Android: Overview about the Android mobile operating system.
- ISO standards used in NFC: Description about different important specifications from the ISO standards.
- E-passport: General overview about the electronic passport.

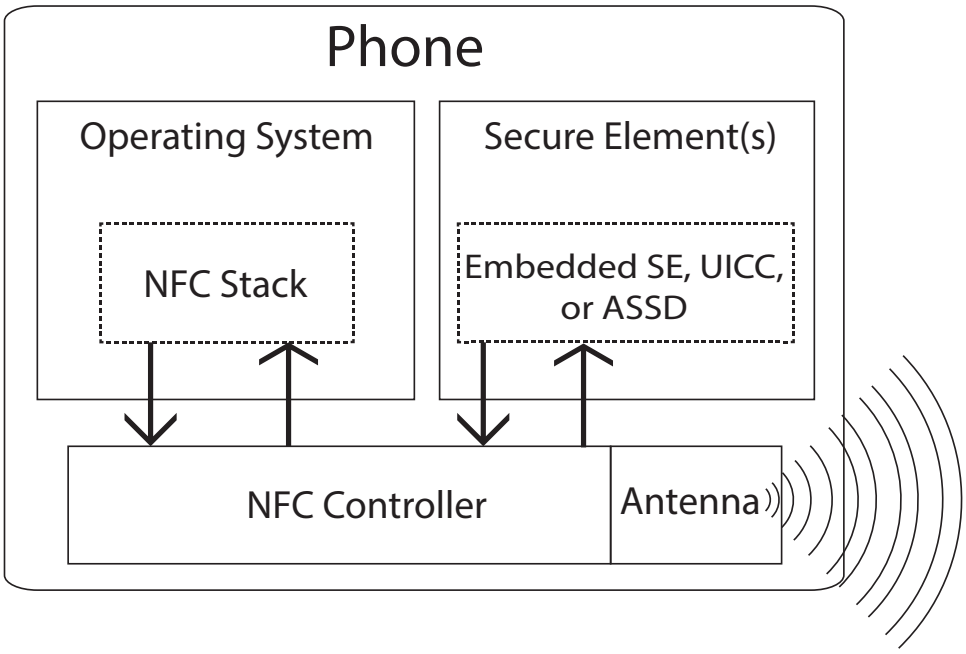
2.1 Near Field Communication

NFC is a contactless technology for short range communication. The distance between the sender and receiver depending on the circumstances can be up to four centimeters.

The NFC controller is the real NFC chip which is responsible for receiving command and to generate the Radio-Frequency (RF) field for the wireless communication. To control the chip one often introduce a NFC stack. The stack is nothing more then stacked libraries to give the Operation System (OS) an Application Programming Interface (API) to quickly control the NFC controller.

Because NFC is a set of standards build upon others standards its compatible with those standards. NFC uses the ISO 14443 standard, it defines how contactless cards should communicate with each other through RFID. Because of this NFC is compatible with RFID tags, for example the Dutch OV-chipcard and the e-passport.

Figure 1: The NFC stack's position in the phone



It is good to note that SE are small micro controllers. They have there own processor and there own OS.

2.1.1 Secure Element

A NFC controller can have access to one or more Secure Elements (SE). When a SE is enabled all communication is handled directly by the SE. A SE is a fully fledged mini

computer on a single chip capable of running applications as any computer can. Most NFC controllers have at least one SE but often have different expansion bays to enable even more SEs.

The name Secure Element is because SE is tamper resistant and should been protected against disallowed third party access. Most SEs also protect against physical access, for example; putting epoxy on the storage chip where all private data is stored and all its contact points. However that example is easy to bypass, so real protection goes much further than putting a little epoxy on the chip.

There are three main types of secure elements often found in devices with a NFC controller.

1. **Embedded** is just a small extra chip that is directly attached to the NFC controller. This chip is often tightly controlled by the manufacturer and has a lot of protection on it to protect against accessing data on the chip. To get an application on this chip one has to sign his application with the private key of the manufacturer and this is often done for a big fee or not done at all.
2. **ASSD** stands for Advanced Security SD card. It is just an SD card with an embedded secure element.
3. **UICC** commonly known for the SIM cards. Accessing this interface is done by using the Single Wire Protocol (SWP). On the other end of interface there should be a smart card enabled device.

2.1.2 Security

There is a lot of work on NFC security (Francis et al., 2010), looking for instance at men-in-the-middle attacks and relay attacks. But when using host-based card emulation, all protection given by a SE is thrown out the window. When an NFC application uses privacy sensitive data this all becomes unencrypted at some point in the OS. Thus making it possible for attackers to manipulate or retrieve this data and opening possible privacy or and security issues like; manipulating of payment records, retrieving sensitive data, exploiting buffer overflows gaining possible arbitrary access to the rest of the system and there will be more exploits and issues.

So when an application uses very privacy sensitive data like credit card information, host-based card emulation should never be used for those applications but instead a SE has to be used.

2.1.3 NFC Modes

NFC communication knows different modes. That is because for example a mobile phone should sometimes have the role of a tag, but sometimes the phone should read a tag. Because of these requirements NFC knows three main modes.

1. **Reading emulation** will put the device in reading mode. This means that it will behave as a reader instead of an tag. In this mode the phone can read tags.
2. **Card emulation** causes the device to behave as an tag. This means that readers can now read this device. BlackBerry was the first phone that would publicly support host-based card emulation (Clark, 2011). Google on the other hand does not support (public or private) host-based card emulation. Google released a system called Google Wallet (Balaban, 2011). This system uses a secure element that only Google can access by signing software, only then it is accepted by the SE.

   Emulation can be done in different ways.

   (a) **SE card emulation** All low level communication that is being received from the NFC antenna goes to the SE. After the SE processes the signals it sends a signal back to send it through the NFC antenna to the reader that is communicating with this device. In other words one can say that the NFC antenna and a SE is just a tag.

   (b) **Host-based card emulation** When using host-based card emulation, all the signals are not send to the SE but to the high level operation systems like Android. Writing code for a high level OS is much easier then a low level OS that is implemented in the constrained SE hardware.

Most hardware that runs Android has megabytes of RAM in contrast to the few kilobytes that SEs has, makes this a big limitation for code that runs on a SE.

3. **Peer-2-Peer** enables two NFC devices to communicate with each other over NFC. P2P knows two modes.
   (a) **Active-Passive** in this mode, one of the two devices, the P2P Initiator, is generating a magnetic field, and the other device, the P2P Target, is receiving it. This mode behaves like classic reader (P2P Initiator) reading the tag (P2P Target).
   (b) **Active-Active** in this mode, both devices generate a magnetic field and are sending and receiving data, this has to be done in turn. This mode is complex and time consuming, as both devices are generating a magnetic field. The complexity comes mainly from all the extra communication needed to manage these fields correctly and avoiding problems. Also because both devices are generating a magnetic field, both devices are draining the battery much quicker.

2.2 Android

Android is an operating system developed by Google and Open Handset Alliance (Burnette, 2009). Nowadays Android runs on a broad range of devices but the original idea of Android was to be an open, simple to develop on mobile OS (Developers, 2011).

Because of the openness of the OS there is a lot of research to improve the hardware usage for the OS (See section 4). This is also the reason why I choice a NFC platform with Android as its OS.

Programming on Android is done in a Java like language (Rogers, Lombardo, Mednieks, & Meike, 2009). So we have to keep this in mind when developing possible solutions for the proposed questions.

2.3 ISO standards used in NFC

Before there where RFID tags and readers, there were old contact cards. These cards had a small contact point (also known as a simcard). The standards for those cards are all defined in (ISO 7816, 1999). Later on when RFID went mainstream, International Organization for Standardization (ISO) introduced a new standard called (ISO 14443, 2008). This standard defined how the original contact cards could communicate over a RFID field.

This section will highlight some points from the standards that are often used in practice.

2.3.1 Answer to reset

Answer To Reset (ATR) (ISO 7816, 1999) is a message that is sent by the tag following a power on. This message contains information about the ISO 14443 and ISO 7816 parameters proposed by the tag. ATR defines the low level protocol of the tag. The reader checks if it can support this protocol and then the communication can begin.

ATR messages also contain historical bytes, with these bytes an tag can define what kind of tag it is or specify some settings for the application protocol. The historical bytes for the Dutch e-passport are `04 38 33 B1` but when you scan a e-passport tag that does not has these historical bytes with a e-passport reader it will continue.

Table 1: ATR of Dutch e-passport, ATR=`3B 84 80 01 04 38 33 B1 BB`

Name	Example	Description
TS	`3B`	Direct convention
T0	`84`	High order = `1000` one interface byte, Low order = 4
TD_1	`80`	High order = `1000`, Low order protocol `T=0`
TD_2	`01`	High order = `0000`, Low order protocol `T=1`
Historical	`04 38 33 B1`	Specification for this card
TCK	`BB`	Checksum xor of the bytes T0 to TCK

Table 2: ATR of CyanogenMod host-based card emulation enabled, ATR=`3B 80 80 01 01`

Name	Example	Description
TS	`3B`	Direct convention
T0	`80`	High order = `1000` one interface byte, Low order = 0
TD_1	`80`	High order = `1000`, Low order protocol `T=0`
TD_2	`01`	High order = `0000`, Low order protocol `T=1`
TCK	`01`	Checksum xor of the bytes T0 to TCK

2.3.2 APDU

Application Protocol Data Unit (APDU) (ISO 7816, 1999) are the communication units between readers and tags. There are two types of APDUs the command APDU and the response APDU.

Table 3: Command APDU

Name	Length (bytes)	Description
CLA	1	Instruction class
INS	1	Instruction code
P1	1	Extra argument data
P2	1	Extra argument data
L_c	0, 1 or 3	The length of the command data
CDATA	L_c	Data
L_e	0, 1, 2 or 3	The maximum number of expected response bytes

The command APDUs (as the name implies) command the tag to do something. The CDATA often also has structures, but those structures can be differ a lot across different commands.

The length flag L_c and L_e can be encoded in different number of bytes. An implementation to parse this can be found in the framework under `apdu/CommandApdu.java`. In short: APDUs can have extended length or normal length. Extended length is defined by setting the first byte of L_c to `0x00`. If the APDU is extended, then both L_e and L_c are extended (two bytes long exclusive the `0x00` marker). If the APDU has not extended length then L_e and L_c are one byte.

It is important to note that we can have none, one or both of the L_e and L_c fields.

After the tag processes the command, it will respond with a response APDU.

Table 4: Response APDU

Name	Length (bytes)	Description
RDATA	Total length - 2	The response data (should never exceed L_e)
SW1-SW2	2	Status code

2.3.3 Applets

The term applet means a very specific small piece of code. Thus when talking about applications for tags and contact smart cards we often speak just about applets.

On a single tag there can be a multiple number of applets and the terminal has to select an applet before continuing with the communication. This is done by first sending an APDU selecting an applet by an application id (AID), after that the tag knows which applet should handle the other APDUs.

2.4 E-passport

E-passports are RFID tags integrated in the physical passports. These tags can be read by RFID readers to easily access your personal identification. One can not suddenly get access to your data stored on the e-passport as it is tightly protected by a number of security layers. To read the most basic information from the e-passport you need the document number, date of birth and date of expiry of the passport.

E-passports suffer from small security issues. A specific attack would be a trace attack (Chothia & Smirnov, 2010). A very few old passports had a static identification number. With that number one could track an e-passport throughout multiple sessions. However nowadays the identification is random for each session.

There are more possible security flaws in the e-passport like scanning, tracking, cloning and eavesdropping (Juels, Molnar, & Wagner, 2005).

There is currently an implementation for the e-passport by the JMRTD project (Martijn Oostdijk, 2006). They also created an e-passport applet. The applet is written in JavaCard and has been ported over to the AndroidCard framework.

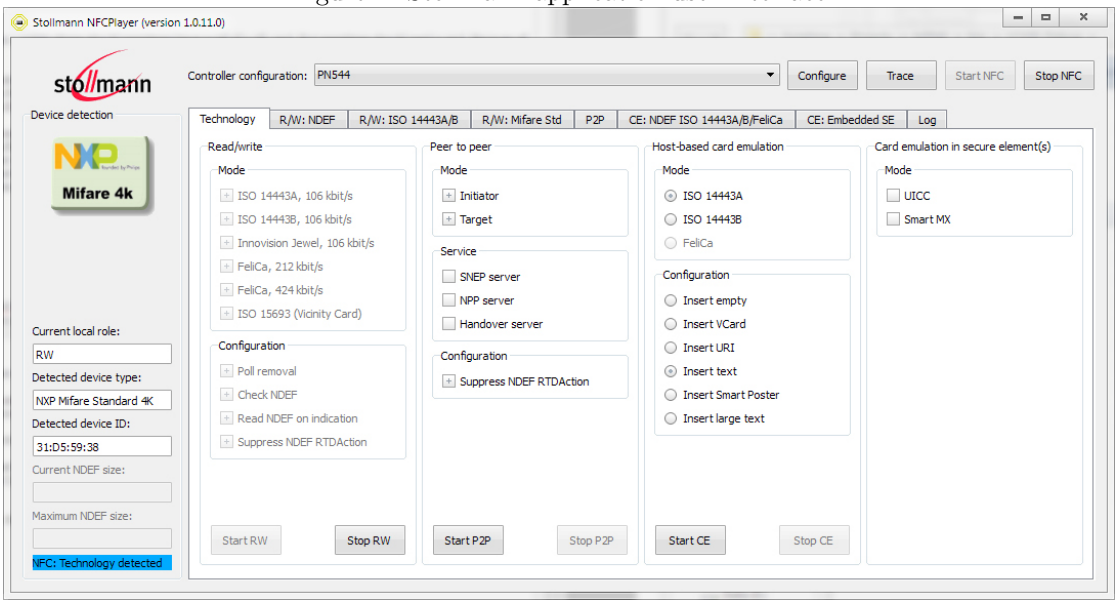
3 Low level NFC

This section and its subsections are not relevant for this paper anymore because CyanogenMod enables host-based card emulation on Android. Before finding CyanogenMod, Stollmanns application claimed to be able to put the NFC chip in host-based card emulation so I used this as a research point.

3.1 Stollmanns application

The Stollmann application is a test suite for testing the NFC stack. The application runs on Windows and can communicate with a few different NFC controllers. It also supports our chip the PN544. Stollmann has a few different version of the application and currently (February 2013) it is on version five. When I started working on this thesis the version was still on only two. The application had a lot of bugs. The current state of the program on the other hand is quite good. The interface is clear and you can do almost everything what the targeted NFC chip could (See figure 2).

Figure 2: Stollmann application user interface



3.1.1 Problems

Stollmanns application always fails to start host-based card emulation. I had contact with Stollmann about this, but they said host-based card emulation should work. Perhaps it has something to do that the first time I try to start card emulation it always fails. But the second time Stollmann application says the chip is in host-based card emulation mode.

After this I tried reading the supposedly working NFC chip with my HTC one X and with the HID Omnikey but both did not show any response at all.

However in version two of Stollmann application I did succeed in getting some response. However I was not able to reproduce this.

3.2 Protocols

In this section we will describe all the protocols used when we talk with a NFC controller from an OS. This section is not needed to understand the rest of the paper as we use an abstraction from this in our final product.

The NFC receiver is a NFC enabled device that is coupled with our NFC stack. Between these two devices communication can now flow.

To communicate correctly from the NFC stack to the NFC receiver on the other end is not easy. To define this communication path a lot of different protocols intertwine (See figure 3).

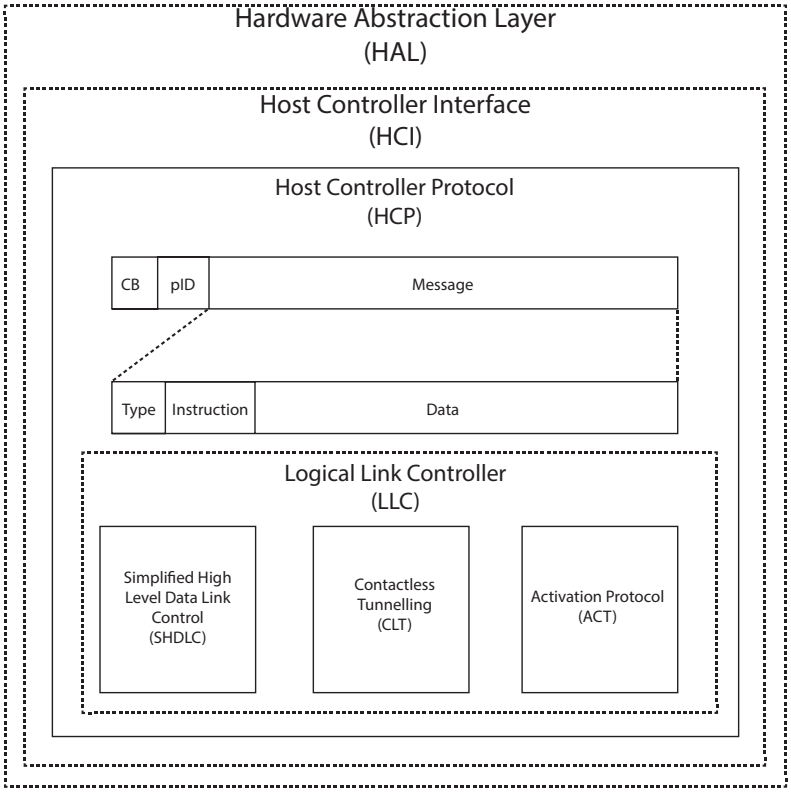
In general when you want to control hardware from software one can often uses a Hardware Abstraction Layer (HAL). This is the whole set of function to manage all communication between high level software and the hardware chip. In this layer are a lot of other protocols to manage each step. Often these steps/protocols are hardware specific.

Communicating with the NFC chip has to be done according to the Host Controller Protocol (HCP). The set of all HCP-related operations is called the Host Controller Interface (HCI).

In our research we found that the HCI packets were wrapped in a unknown protocol when using a special test board from NXP

Inside the HCP data field we find the Logical Link Controller (LLC). This is the lowest layer in the chain. However there are different protocols defined in the LLC. Namely: Simplified High Level Data Link Control (SHDLC), Contactless Tunneling (CLT) and Activation Protocol (ACT). Where CLT is optional and the other two are mandatory.

Figure 3: Overview of the different protocols in the NFC stack and how they interact



In the deepest level we will find the NFC controller (the real chip).

3.2.1 Logical Link Controller

Inside the SHDLC frames one can find the final APDUs (ETSI, 2008). The CLT and ACT frames are management frames.

4 CyanogenMod

CyanogenMod is an open source OS based on Android. It extends the basic Android with patches and numerous extra features. CyanogenMod is brought to life by Steve Kondik (a.k.a Cyanogen). However nowadays there are a lot of freelancers working on CyanogenMod in their free time. One of those features is a patch to enable NFC host-based card emulation on Android.

CyanogenMod started with version six, this is based on Android 2.2. After this CyanogenMod released different versions to keep in sync with the Android releases.

4.1 The patch

The patch to enable NFC is now (24 March 2013) deployed to all versions greater then CyanogenMod 9. This patch extends the Android API to enable host-based card emulation. This is achieved by adding extra intents to the Android system.

4.2 The intricacies of the patch

CyanogenMod has not made any documentation publicly available. However the blog "Android Explorations" has written an article about how it works and how one can use it to write NFC applets (Elenkov, 2012a).

I will summarize how to use the extended NFC intent for CyanogenMod (all code is from (Elenkov, 2012a)). When making an Android application you have to modify the manifest to support permissions for NFC. Then you can register NFC intent in your MainActivity and start handling tags.

4.2.1 AndroidManifest.xml

The Android `manifest xml` file should request permission to use the NFC. For the CyanogenMod patch it should also include a `filter_nfc.xml` list with the technology to use. This is simply a small file with the name of the accepted technology.

```
<activity android:label="@string/app_name"
          android:launchmode="singleTop"
          android:name=".MainActivity"
    <intent-filter>
        <action android:name="android.nfc.action.TECH_DISCOVERED" />
    </intent-filter>
    <meta-data android:name="android.nfc.action.TECH_DISCOVERED"
               android:resource="@xml/filter_nfc" />
</activity>
```

4.2.2 filter_nfc.xml

```
<resources>
    <tech-list>
        <tech>android.nfc.tech.IsoPcdA</tech>
    </tech-list>
</resources>
```

4.2.3 TagWrapper.java

As the default API from Android does not have access to these new functions exported by CyanogenMod. We had to make a wrapper class that implements the TagTechnology interface. This is achieved by reflection. Reflection calls the correct methods exported by CyanogenMod. See for full implementation: (Elenkov, 2012b) `se-emulator/src/org/nick/se/emulator/TagWrapper.ja`

```
abstract class BasicTagTechnology implements TagTechnology {
    public boolean isConnected() {...}
    public void connect() throws IOException {...}
    public void reconnect() throws IOException {...}
```

```
5       public void close() throws IOException {...}
        byte[] transceive(byte[] data, boolean raw) throws IOException {...}
7   }
```

4.2.4 MainActivity.java

The `MainActivity.java` is an example to get the real tag from which we can transceive. Its important to note that the `onNewIntent(Intent intent)` function is heavily simplified. As in a real world scenario you want to place a lot of integrity checks on the intent and tag itself.

```
1   public class MainActivity extends Activity{
        public void onCreate(Bundle savedInstanceState){
3           pendingIntent = PendingIntent.getActivity(this, 0, new Intent(this,
                getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP), 0);
5           filters = new IntentFilter[]{new IntentFilter(
                NfcAdapter.ACTION_TECH_DISCOVERED)};
7           techLists = new String[][]{{"android.nfc.tech.IsoPcdA"}};
        }
9
        public void onResume(){
11          super.onResume();
            if (adapter != null)
13              adapter.enableForegroundDispatch(this, pendingIntent,
                    filters, techLists);
15      }

17      public void onPause(){
            super.onPause();
19          if (adapter != null)
                adapter.disableForegroundDispatch(this);
21      }

23      @Override public void onNewIntent(Intent intent){
            tag = (Tag) intent.getExtras().get(NfcAdapter.EXTRA_TAG);
25          TagWrapper tw = new TagWrapper(tag, TECH_ISO_PCDA);
        }
27  }
```

5 AndroidCard

CyanogenMod has no nice support for writing NFC applets, thus making this a tedious task. To provide better support for this and to keep the applets as close as possible to original applets written in JavaCard, I have written a framework called AndroidCard. This provides an interface that is convenient and easy to use. I also tried to make AndroidCard as equal as possible to JavaCard, to make porting applets from JavaCard to AndroidCard easy.

The different parts of AndroidCard including the test and sample application can be found on Github (framework[1], e-passport[2] and test application[3]).

5.1 Framework

The framework consists out of a management system and a lot of classes to handle APDUs. AndroidCard is the main class of the framework and you can use AndroidCard to register a new applet. After you have registered your applets you can call `handleTag` method to let the framework and applet handle the communication correctly.

```
1 public class AndroidCard
  {
3     public boolean register(Applet applet); // Registers an applet to the framework
      public boolean handleTag(TagWrapper tag); // Handles a new incoming reader
5 }
```

5.1.1 Management system

The management system simple holds a list with all registered applets. Whenever a new NFC terminal tries to read the NFC phone we pass the terminal to the management applet selector. Inside the applet selector it will retrieve the applet selection APDU and checks if we have an applet with the corresponding application id. If so we start that applet and pass the communication from the terminal onto that applet. From that point on the applet receives all APDUs in the `ResponseApdu process(CommandApdu apdu)` method.

When registering an applet to the system we have to use `boolean register(Applet applet)` where applet can be an extended applet.

5.1.2 Applets

Applets in the framework are classes that extend the abstract class Applet. They are ran in there own threads and when the terminal sends an APDU to the phone the `process()` method gets called with the APDU. The `process()` method has to return `null` or a `ResponseApdu` that will be send back to the terminal.

```
1 public abstract class Applet
  {
3     public abstract ResponseApdu process(CommandApdu apdu);
      public void select(); // Called when terminal selects this applet
5     public void deselect(); // Called when connection to terminal is lost
      public abstract String getName(); // The name of the applet (friendly)
7     public abstract byte[] getAid(); // The application identifier
  }
```

[1] https://github.com/kevinvalk/android-hce-framework
[2] https://github.com/kevinvalk/android-hce-applet-passport
[3] https://github.com/kevinvalk/android-hce-test

5.1.3 APDUs

In AndroidCard we have support for the JavaStruct layout, this means one can define a structure in Java and cast raw bytes to such a structure visa versa. There are two main APDUs in the system: ResponseApdus and CommandApdus. CommandApdus travel from terminal to phone and ResponseApdus travel from phone to terminal.

```
@StructClass public class ResponseApdu extends Apdu
{
    @StructField(order = 0)
    public byte[] data;
    @StructField(order = 1)
    public short sw;

    @Override public byte[] getBuffer() // Get the full raw APDU
}
```

```
@StructClass public class CommandApdu extends Apdu
{
    @StructField(order = 0)
    public byte cla;
    @StructField(order = 1)
    public byte ins;
    @StructField(order = 2)
    public byte p1;
    @StructField(order = 3)
    public byte p2;

    public int getLc() // Get the length of the data (or zero)
    public int getLe() // Get the length of the expected data (or zero)
    public byte[] getData() // Get the data buffer
    @Override public byte[] getBuffer() // Get the full raw APDU
}
```

5.2 E-passport applet

As stated in the introduction of this paper the goal was to show that with AndroidCard we could emulate an e-passport on a smart phone. For that reason I started working on a well defined Applet using the complete power of Android and AndroidCard. However the e-passport is quite hard to fully understand and it was taking a lot of time to do a complete rewrite. About halfway in this rewrite I decided to halt my work, but instead do a port on the JMRTD JavaCard applet (See chapter 6)

5.2.1 Structures

Because the `CDATA` inside the `CommandApdus` are also structured and because we where already using JavaStruct, I defined structure classes to easily access the different data fields inside the APDU `CDATA`.

For example Basic Access Control (BAC) uses the following structure for the instruction `INS_MUTUAL_AUTHENTICATE`.

```
@StructClass public class MutualAuthenticate extends Structure
{
    @StructField(order = 0)
    public byte[] randomFrom = new byte[Constant.RND_LENGTH];
    @StructField(order = 1)
    public byte[] randomTo = new byte[Constant.RND_LENGTH];
    @StructField(order = 2)
    public byte[] key = new byte[Constant.KEY_LENGTH];
```

```
10      MutualAuthenticate(byte[] data) // New from already decrypted data
        MutualAuthenticate(byte[], SecretKey) // New from cipher and decrypt it
12      byte[] getEncoded(SecretKey, SecretKey) // Encoded buffer cipher||mac
}
```

To use this in practice I first extract the cipher from the APDU and then initialize the structure. From that point onwards I can freely work with the structure.

```
1  byte[] cipher = Arrays.copyOfRange(apdu.getData(), 0,
       Constant.LC_MUTUAL_AUTHENTICATE_DATA);
3  data = new MutualAuthenticate(cipher, passport.mutualEncKey);
```

The reason that I used this approach, is because coping data from buffers with offsets and lengths is very error prone. With this method, when you defined your structure correctly you can hardly make any errors regarding to data access.

5.3 Test application

The test application is a relatively simple Android application that uses the techniques described in section 4 to registers the host-based card emulation routines and passes the incoming terminals to the framework. The test application also registers the e-passport applet to the framework in turn enabling terminals to communicate with the e-passport applet.

6 E-passport JavaCard to AndroidCard port

As explained throughout this paper the idea was to demonstrate the framework by emulating the e-passport on a smart phone that could run AndroidCard. At first I tried to create the e-passport application from scratch. This was a very tedious and time consuming process and on some point I decided to see if porting the existing e-passport applet written in JavaCard would be quicker.

This section will walk you through the steps I took to port the existing applet over to AndroidCard. It took me just two hours to get the source compilable but then it took me an extra 15 hours to fix all bugs introduced in the porting process.

6.1 The process

As the existing applet was quite big and using an awful lot of JavaCard cryptographic APIs, I decided to rip jCardSim (LLC, 2011). jCardSim is a JavaCard runtime simulator but it had all implementations for the used cryptography ready. The following steps I took to get jCardSim working as a library that could be used in the ported applet:

- Removed unneeded parts that had to do with simulating
- Replaced bouncycastle with spongycastle (provided by AndroidCard)
- Rewrote JCSystem to use `new` constructor
- Fixed a small bug in the `SymmetricSignatureImpl.java` sign function when calling with null input

With this, now working library, almost all compile problems where solved, but not all. To completely get the applet port compiling I had to do following steps for all files.

- Solved wrong imports and add correct imports
- Replaced ISOException with IsoException
- Replaced ISO7816 with Iso7816
- Use Iso7816 class to access constants

It was compiling now but was far from working, from this point on the tedious task of finding bugs and fixing them started. I also had to extend some parts of the system to correct the applet to the new environment. I will now shortly list the steps I took to achieve this.

- Some general cleanup (removing some unneeded parts)
- Implemented `getName` and `getAid`
- Rewrote process to accept `CommandApdu` and return `ResponseApdu`
- Replaced all APDU handlers with `CommandApdu` handlers
- Let the APDU handlers response with `ResponseApdu`
- Get the expected return length from `CommandApdu.getLe()`
- Because the JavaCard `APDU` class has a global buffer you can use the buffer for your own calculations. However in AndroidCard the buffer is local and as big as the original APDU. So to let the in buffer calculation work I made a new buffer in every APDU handler function that was five times the size of the original APDU. This was only a quick and dirty workaround to achieve this port.
- Added `select()` to correctly reset the passport state
- Rewrote big parts of `unwrapCommandAPDU` and `wrapResponseAPDU` to correct bugs and support `CommandApdu` and `ResponseApdu`
- Added `setFile` method to write data group files to the passport file system
- Added `setPassport` method to setup the parameters for Basic Access Control (BAC)

The JMRTD applet port was now complete. I had to dump my real e-passport file system and add those files to the test application. In the test application I added some extra code to write my passport files to the applet and set the correct BAC. It now was a working clone of my original e-passport. Sadly there is a strange bug where given names do not show up when reading the applet.

The process was tedious and quite hard, mainly because using in buffer calculations for cryptography. But all in all you can quite easily port existing applets to AndroidCard.

7 Conclusion

With CyanogenMod it is already possible to put an Android NFC enabled phone into host-based card emulation, so with that my original research question has already been answered. However what CyanogenMod publishes is not user friendly and makes it very hard to port or write applets for Android.

For this reason I have written AndroidCard that uses the concepts of JavaCard but with the extra resources of the fully fledged Java of Android to support the writing of Applets. This framework only supports Android with the CyanogenMod because CyanogenMod is the only currently known method to achieve host-based card emulation on Android.

Before discovering CyanogenMod the answer to my question was less clear. So for that reason I received a NXP development board with a NFC chip. With this board I tried to document the chip, to further down the research write code to use the chip. This was very hard to do so, as there is no public information available whatsoever. Stollmann is a company that produces SDKs and testing application for NFC stacks and with this application I was able to use the chip. However it turned out to be next to impossible to put that chip into host-based card emulation, the thesis was at a loss because of this. But luckily I found the solution through CyanogenMod.

To show that AndroidCard really works I ported an applet written in JavaCard for the e-passport to AndroidCard. This process took somewhere between 15 and 20 hours to make a fully working port. There was not an awfully lot of code to adjust. However fixing the little bugs that where introduced in the process of porting, where very hard to squash.

With the framework one can now easily write NFC applets on Android. AndroidCard is open source and can be found on Github (See section 5).

7.1 Further work

When writing this thesis there was some questions left unanswered.

1. Figure out the unknown protocol that is used over the HCI diagram in our tests (See section 3.2).
2. Is it possible to use a relay attack with two NFC enabled phones in which one functions as RFID tag and the other as RFID terminal? It turns out this is possible as shown in (Francis, Hancke, Mayes, & Markantonakis, 2012)

References

Balaban, D. (2011, June). *With launch of google wallet, the wallet war begins.* Retrieved from `http://nfctimes.com/blog/dan-balaban/launch-google-wallet-wallet-war-begins` (Blog post)

Burnette, E. (2009). *Hello, android: introducing google's mobile development platform.* Pragmatic Bookshelf.

Chothia, T., & Smirnov, V. (2010). A traceability attack against e-passports.

Clark, S. (2011, May). *RIM releases BlackBerry NFC APIs.* Retrieved from `http://www.nfcworld.com/2011/05/31/37778/rim-releases-blackberry-nfc-apis/` (Blog post)

Developers, A. (2011). What is android? *http://developer.android.com/guide/basics/what-is-android.html*, *2*.

Elenkov, N. (2012a, October). *Emulating a PKI smart card with CyanogenMod 9.1.* Android Explorations. Retrieved from `http://nelenkov.blogspot.nl/2012/10/emulating-pki-smart-card-with-cm91.html` (Blog post)

Elenkov, N. (2012b, October). *Virtual PKI Smart Card.* Github. Retrieved from `https://github.com/nelenkov/virtual-pki-card` (Github)

ETSI. (2008, Septembre). Smart Cards; UICC - Contactless Front-end (CLF) Interface; Part 1: Physical and data link layer characteristics (Release 7). *ETSI TS*(102 613), 57. (Specifications)

Francis, L., Hancke, G., Mayes, K., & Markantonakis, K. (2010). Practical Relay Attack on Contactless Transactions by Using NFC Mobile Phones. Cryptology ePrint Archive, Report 2011/618. Retrieved from `http://eprint.iacr.org/2010/228` (lishoy@gmail.com 14722 received 22 Apr 2010)

Francis, L., Hancke, G., Mayes, K., & Markantonakis, K. (2012). Practical relay attack on contactless transactions by using nfc mobile phones. In N.-W. Lo & Y. Li (Eds.), *The 2012 workshop on rfid and iot security (rfidsec 2012 asia)* (Vol. 8, p. 21 - 32). IOS Press.

ISO 14443. (2008). ISO/IEC 14443 - Identification cards - Contactless integrated circuit cards - Proximity cards - Part 1 to 4. *International Standard*. (Specifications)

ISO 7816. (1999). ISO/IEC 7816 - Identification cards - Integrated circuit(s) cards with contacts - Part 1 to 15. *International Standard*. (Specifications)

Juels, A., Molnar, D., & Wagner, D. (2005). Security and privacy issues in e-passports. In *Security and privacy for emerging areas in communications networks, 2005. securecomm 2005. first international conference on* (pp. 74–88).

LLC, L. (2011). *jCardSim: Java Card Runtime Environment Simulator.* (Google code)

Martijn Oostdijk, e. a. (2006). *JMRTD: An Open Source Java Implementation of Machine Readable Travel Documents.* (Manufacturer website)

Mostowski, W., & Poll, E. (2010). Electronic passports in a nutshell.

Rogers, R., Lombardo, J., Mednieks, Z., & Meike, B. (2009). *Android Application Development: Programming with the Google SDK* (1st ed.). O'Reilly Media, Inc. (Book)

Shepard, S. (2005). *RFID: radio frequency identification.* McGraw-Hill New York.