NFC – Synchronizing Mobile Devices with Embedded Systems

COMP 8045 – 9 Credit Half Practicum

David Tran
A00801942 British Columbia Institute of Technology

NFC - Synchronizing Mobile Devices with Embedded Systems

David Tran - A00801942

9 Credit Half-Practicum

COMP 8045 Set 8D

British Columbia Institute of Technology

Tuesday March 17, 2015

Sponsor & Supervisor: Aman Abdulla

Table of Contents

List of Figures

Abstract

The problem we attempt to solve in this project is to be able to use Near Field Communication as a means of security and authentication. To do this, the project seeks to enlist the use of the modern day smartphone as the NFC medium and the use of embedded systems, specifically the Raspberry Pi and Arduino shields. To increase the security of NFC authentication, the project seeks to experiment with well-known security technologies, such as implementing a Public Key Infrastructure. The project seeks to be innovative by marrying smartphones with security technologies, all the while, keeping in mind the emergence and popularity of NFC. The project deliverables shall include the functioning communication between smartphone and NFC reader without the security implementation. In addition, the project shall also deliver successful or unsuccessful attempts with previously aforementioned security implementation.

Forward

The Bachelor of Technology practicum from the British Columbia Institute of Technology is a self-assigned project in which the student does not have a company sponsorship. In turn, because the requirements for the 9-Credit COMP 8045 project practicum requires some form of sponsorship, the project's sponsor shall also be the project's and student's assigned subject expert. Enclosed in the appendices of this document is the sign-off that the sponsor and subject expert assigned has seen the live demonstration of the project.

Project Background

General Background of the Project

In business organizations, the use of card scanning as a means for authentication is common. The requirements of such a system involves a central server that can monitor and maintain access privileges, an integrated method of printing these identification cards, a protocol for authentication, card readers and of course, the cards themselves. Unfortunately, these cards are often made of plastic, and while businesses are not required to have to mass-produce these cards often, it is still not environmentally friendly.

A replacement would require some form of contact or wireless communication that is secure enough for means of authentication. Furthermore, the medium needs to be compliant with current technologies, or be capable to transmit a unique identifier that can quickly and reliably authenticate the user. And of course, the proposed solution needs to be cost-effective for both the user and the organization.

As technology advances in today's society, so too does the technology become readily available to consumers. Prime examples are today's smartphones. Many of today's smartphones come packed with Near-Field Communication (NFC) chips, a means of authentication. Many major banking institutions have adopted this technology to further their mobile banking experience to their users, as well as a means to attract customers to use mobile payment, simply by tapping their phone on tap-enabled electronic fund transactions (EFTs).

If banking institutions trust the technology enough to invest into it, then NFC technology should be secure enough for use by other businesses and by homeowners. At the moment, NFC has not been adopted as a means of security but the medium – the smartphone – has been used in conjunction with many other technologies such as WiFi, and Bluetooth. We want to discover if NFC is capable to provide the same – if not, better – means of security as the previously mentioned technologies.

Project Caveats

Upon further research of NFC capabilities and RFID, while NFC is a subset of the latter it is incompatible with existing RFID technologies. This is because NFC operates at 13.56MHz: a high-frequency subset of RFID while traditional RFID technologies operate at low-frequency 125 - 134kHz. In turn, RFID can be used on an NFC reader and that suggests that future implementations of any form of frequency-based authentication and security should resort to implementing high-frequency technologies over low-frequencies.

Other interesting caveats for this project occurred from experimentation with NFC modules with the Raspberry Pi and Arduino. The Raspberry Pi can efficiently communicate with many different bus protocols, most commonly found are Inter-Integrated Circuits (I2C) and Serial Peripheral Interface (SPI). And while the Pi is flexible with whichever protocol, the Arduino components are not. There are specific libraries engineered by the embedded systems community that are either I2C, or SPI; there is minimal common ground between the two. Why is there such a restriction? This is due to the fact that some of the Arduino hardware is made specifically to a certain implementation and by being compatible with multiple protocols reduces the hardware's efficiency. For further details, refer to the **Project Development Details** section of this document.

Finally, the project has opted to use more modern smartphones as an authentication medium. The intention and rationale behind it is to use the most up-to-date hardware so that the communication would be "top-notch." This has proven to be false because of Samsung's and Google's decision to use Broadcom's NFC chip over NXP's. Due to this change, many MiFare Classic tags that follow the ISO/IEC 14443-A/B standards are not supported by Android smartphones, but conversely are readable by Arduino hardware. The smartphones available for such activities are devices that were released in roughly 2012.

Purpose of the Project & Components for Innovation

The primary purpose of the proposed study is to understand if NFC authentication is an attractive means for security. It is important to understand that this study will not be addressing the security risks between digital and physical authentication. The purpose of this research is to develop and integrate components for innovation that consists of smartphones, embedded systems, and the interrelationship between Radio Frequency Identification (RFID) and NFC.

As stated in the 9-credit COMP 8045 criteria, the project is required to be innovative as well as containing elements that are considered experimental, or exploratory in nature to meet the criteria of the COMP 8045 practicum. The project seeks to be innovative and explorative by attempting to integrate currently used means of encryption in the form of public key infrastructure, digital certificates and digital signatures with NFC technologies. The project also seeks to experiment with the security of the aforementioned implementation by conducting a series of penetration tests using the Kali Linux suite.

Firstly, the innovation criteria of the project can be justified by the fact that NFC is an emerging technology that only recently has been implemented in sensitive transactions. A prime example is the previously highlighted use of NFC and mobile banking. The project seeks to take the next step with NFC to use it as a means of secure authentication. This component also fulfills the criteria for innovation because NFC has yet to be used in place of Bluetooth and WiFi. With that being said, any form of security besides utilizing the mobile carrier's encryption methods shall also be innovative.

Secondly, the explorative criteria of the project is justified because of how NFC is an emerging technology, making any other integrated component to be a part of emerging technology. Therefore, while the general consensus regarding public key infrastructures, digital certificates and digital signatures are considered to be a staple in security and integrity, they become new technology when applied to other newly emerging technologies. The explorative component comes from researching and understanding the above-mentioned means of security, implementing them, and then ensuring that proper communication ensues.

Finally, the experimental component of the project is achieved by conducting a series of penetration tests on NFC technologies. Ultimately this will highlight any drawbacks or downfalls of using NFC technologies as a means of security and authentication. By revealing these flaws, the project can then seek to patch these flaws, making NFC authentication that much more attractive. After achieving the above three criteria, the project seeks to continue innovation by utilizing popular and other emerging technologies. These technologies are highlighted below:

Utilizing Embedded Systems

As a proof-of-concept, the project shall implement its system using Raspberry Pi. The Raspberry Pi shall serve as the infrastructure's central server, in which it will process the communication between the NFC reader and the NFC-capable smartphone. The system is also beneficial due to its form factor and simplicity, making the development of a proof-of-concept much more feasible within the limited time frame of this project. The concepts of embedded systems can be applied to any similar embedded system in the present, such as Arduino chipsets, and in the future.

Integration of Mobile Devices

The majority of NFC technologies reside within modern smartphone architecture. Therefore, the prime candidates to test NFC hardware and software would be to utilize smartphones. Unfortunately, at this time, only Android- and Windows OS-based smartphones are capable of NFC development; Apple iOS NFC technologies have been extremely limited to mobile payments. The smartphone of choice for this project shall be the Google Nexus 5 or the Samsung Galaxy Note 4; we have opted for an Android-based smartphone because of its availability to the researcher and due to its larger smartphone market share. The choice of either smartphone is dependent on the financial budget of the researcher.

Proposed Solution

The project is comprised of three components: the NFC reader, the NFC medium and the software that bridges the other two components together. For the NFC reader, the project shall be using a Raspberry Pi connected to an Arduino Uno via USB Serial. The Arduino Uno shall have Seeed Studio's NFC Shield V2 connected in order to communicate with the NFC medium.

The NFC medium shall be the Samsung Galaxy Note 4. Although in the previous discussion of our **Project Caveats**, the Note 4's NFC hardware and software capabilities have been shown to cooperate with the NFC reader setup. This is also a better representation in case the project proof-of-concept becomes widely adopted; more users within the next two years shall upgrade to a newer phone with newer NFC capabilities. Specifically for Android, the project shall implement the communication with the NFC reader using Host-Card Emulation.

In addition, the Android operating system is written in the Java programming language and one of the few capable smartphone operating systems that permits the user to utilize NFC technologies outside of mobile payment options. Finally, the software component shall be implemented using a combination of C, C++ and Python programming languages. C and C++

are responsible for handling the operations of the Arduino while Python shall be parsing the serial data for the Pi.

Alternative Solutions & Justifying the Implementation

Alternative solutions for the communication between NFC reader and NFC medium could be to utilize Peer-to-Peer capabilities such as Android Beam or S-Beam, specifically for Samsung instead of Host-Card Emulation. In retrospect, the NFC reader could implement NXP's EXPLORE-NFC module for the Raspberry Pi, eliminating the middleman design currently proposed with the Arduino Uno and its NFC Shield. Consider the following high-level diagram for the following discussion topics:

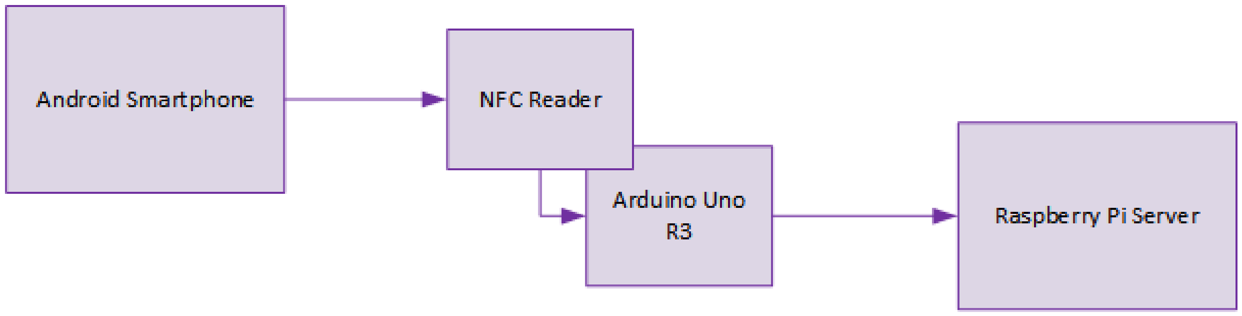


Figure 1 - System High-Level Diagram

It is important to note that the NFC Reader is attached to the Arduino Uno, which then directly communicates to the Raspberry Pi server using a connected USB Serial cable. The smartphone, however, communicates with the system by conducting physical contact with the NFC reader.

Host-Card Emulation over Peer-to-Peer

Peer-to-Peer was the project's first experimentation because the NFC chip requirements were not as strict on the smartphone, so the communication between the reader and the device was seamless. However, there was a few extra steps that had to be taken in order to communicate with the NFC reader when using Peer-to-Peer:

Step	Action
1	The smartphone needs to run the app that utilizes Android Beam
2	The User had to place their smartphone in the vicinity of the NFC Reader
3	The smartphone’s display shall ask the user to touch the device again so that the device can beam the information by prompting the user to “Touch to Beam”
4	The User touches the display
5	if the smartphone was moved away from the vicinity of the reader, shall ask the User to place the smartphone back into the reader’s range
6	Android / Arduino shall output the appropriate messages regarding the successful or unsuccessful transfer of data

When utilizing Host-Card Emulation, we can see that the use case is reduced in comparison with Peer-to-Peer:

Step	Action
1	The smartphone needs to run the app that utilizes Host-Card Emulation
2	Enter the passphrase to be sent to the Arduino
3	Scan the phone; the Android app and/or the Arduino shall output the appropriate messages regarding the successful or unsuccessful transfer of data

Essentially, the Host-Card Emulation approach reduces the use case by about three steps, which makes it much more ergonomic for the user. This was a deciding factor between Peer-to-Peer and Host-Card Emulation.

Seeed Studio NFC Shield V2.0 over Other NFC Readers

The Seeed Studio NFC Shield V2.0 was selected as the NFC reader of choice over other readers that were experimented with in this project. The deciding factor was that the module fulfilled the requirements needed in order to communicate with the Samsung Galaxy Note 4. Other factors were that the NFC Shield uses the SPI bus protocol versus I2C, and had many more available libraries that were robust enough to provide Host-Card Emulation communication.

A case can be made that the EXPLORE-NFC module fully supports libnfc libraries, but the provided examples and sample code for the NFC Shield and Android applications worked in tandem without too much modification to the code. Finally, the NFC Shield came and worked right out of the box with Arduino, which can also be said of the EXPLORE-NFC module. Other readers required finicky installation and setup procedures that makes the proof-of-concept too complicated and cumbersome. However, in the future, a suggestion can be made to try utilizing the EXPLORE-NFC module with libnfc libraries as a potential project.

Project Design Work

Feasibility Assessment Report

The Feasibility Assessment Report is composed of three components in which the project is required to evaluate in order to reach its potential success. **Operational Feasibility** shall highlight the objectives of the report and then determine if the project is capable of attaining such objectives; the higher chances of fulfilling the objectives, the better. **Technical Feasibility** analyzes the hardware and software components of the project and then indicates whether the project can proceed based on the potential limitations or roadblocks found in its analysis; the more doable the project is, the better. Finally, **Legal and Contractual Feasibility** is concerned with any copyright infringement or plagiarism of the project and analyzes the likelihood of breaching them; the lower the likelihood, the better.

Operational Feasibility

The project has a set of objectives that roughly describe its ideal successfulness. These objectives are not so broad that they only conceptual, but they are not so specific on how to implement or obtain said objectives. The project objectives are listed in the following table:

	Objectives	Obtainability
1	Reader and smartphone must be able to communicate in some way.	High
2	The system must implement Public Key Infrastructure.	High
3	Data is encrypted on the smartphone, then decrypted on the reader.	High
4	The data shall have a digital signature along with its payload.	High
5	The Android phone distributes its own digital certificate for authenticity.	Impossible

All objectives are obtainable except for (5) because of project design and implementation. While certificates grant an extra level of authenticity, it is more likely to be implemented in a web application, not in a system such as this project. This objective has been discarded.

Technical Feasibility

The technical feasibility of the project is very high due to the fact that all software components (see **Tools and Equipment Used** section) are readily available. All components listed in the below table were acquired within the first month of the project. While financial issues occurred during the research component of the project, all components were eventually obtained.

	Component	Transportation	Cost	Criticality
1	Adafruit NFC Breakout Board	3 days, $30.00 shipping	$80.00 USD	Medium
2	Adafruit NFC Shield	3 days, $30.00 shipping	$80.00 USD	Medium
3	NXP EXPLORE-NFC Module	5 days, $30.00 shipping	$70.00 USD	Medium
4	Seeed Studio NFC Shield V2	3 days, $30.00 shipping	$70.00 CAN	High
5	Samsung Galaxy Note 4	N / A	Already Owned	High
6	Samsung Galaxy S4	N / A	Already Owned	Medium
7	Raspberry Pi Model B+	14 days, $10.00 shipping	$40.00 USD	Low
8	Parallax RFID Reader & Tags	14 days, $10.00 shipping	$30.00 USD	Low
9	NFC Tags (NTAG213)	7 days, $5.00 shipping	$10.00 CAN	Low
10	Dexter Industries Arduberry	3 days, $30.00 shipping	$70.00 CAN	High
11	Arduino Uno R3	3 days, $30.00 shipping	$50.00 USD	High
12	Breadboard, Wires, Miscellaneous	N / A	$50.00 CAN	Medium

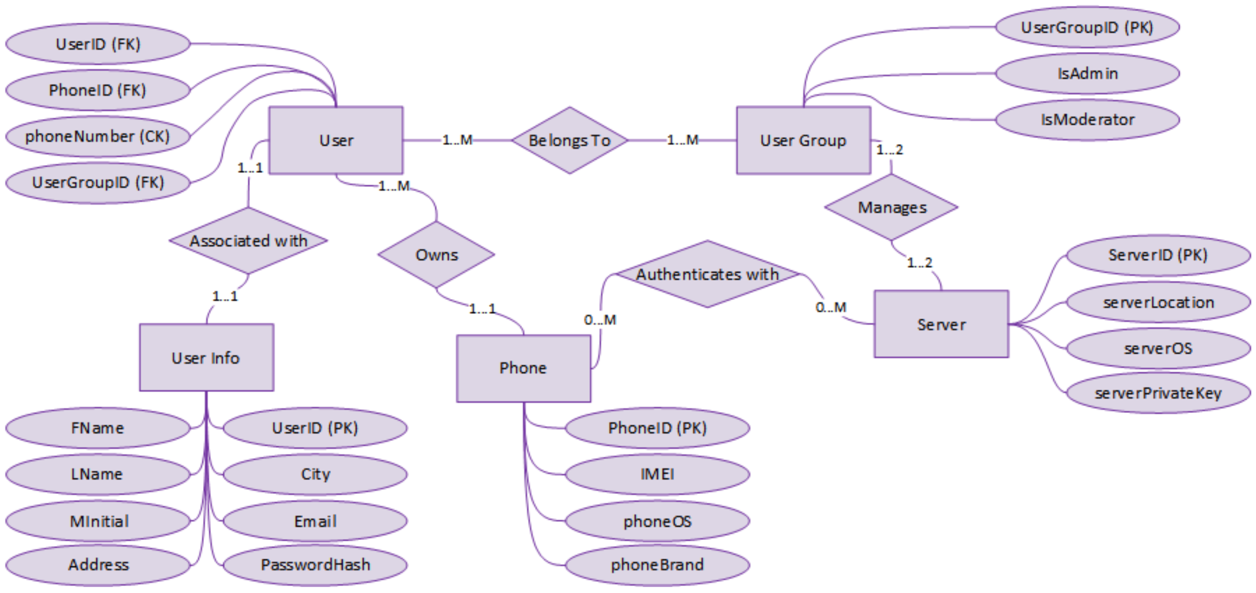
The bulk of technical difficulties that hinder the project's technical feasibility therefore can be limited to the understanding and availability of current software technologies. Hardware difficulties can also arise, but have been mitigated by experimenting with multiple hardware solutions.

Legal & Contractual Feasibility

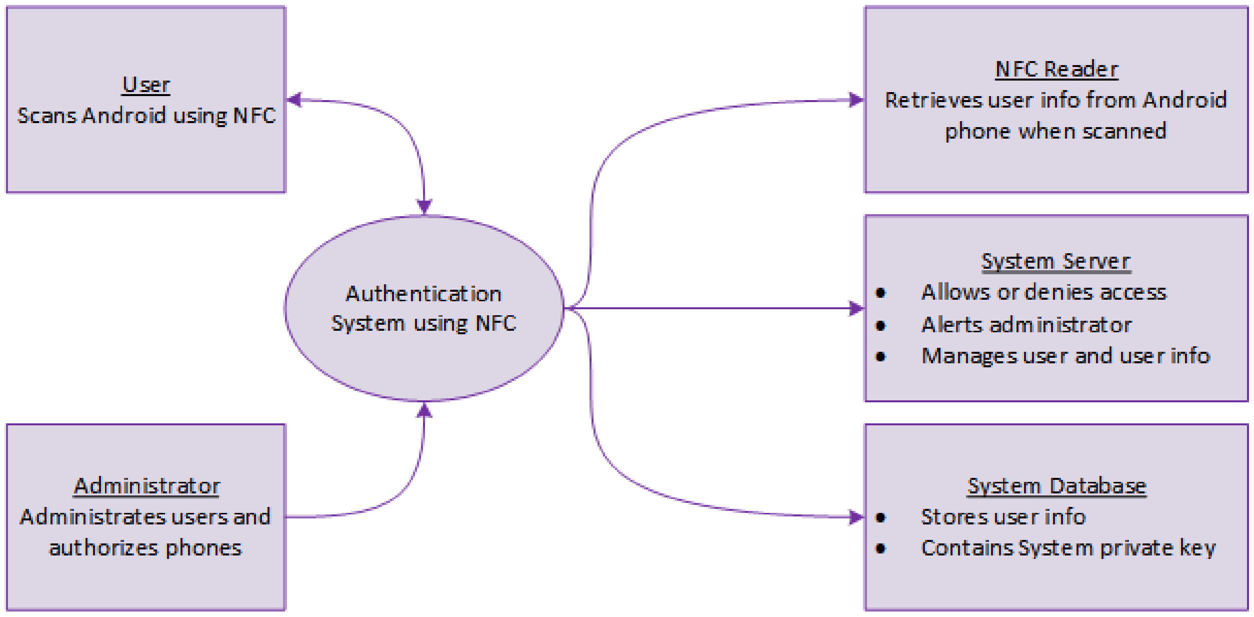
All libraries for Android development are considered open source and have not been obtained or used illegally. All libraries for the Arduino were obtained by the distributor, its community and any authors that have made their code public; these individuals have been credited in the **References** section of this document. All libraries for the Python program of the project are also considered open source and have not been obtained or used illegally.

This project, to its author's and researcher's knowledge, have no recollection or belief that there is an identical project like this one that exists. Due to these low-risk legal and contractual analyses, the project has a low rate of attaining any legal or contractual issues throughout its development and project lifecycle.

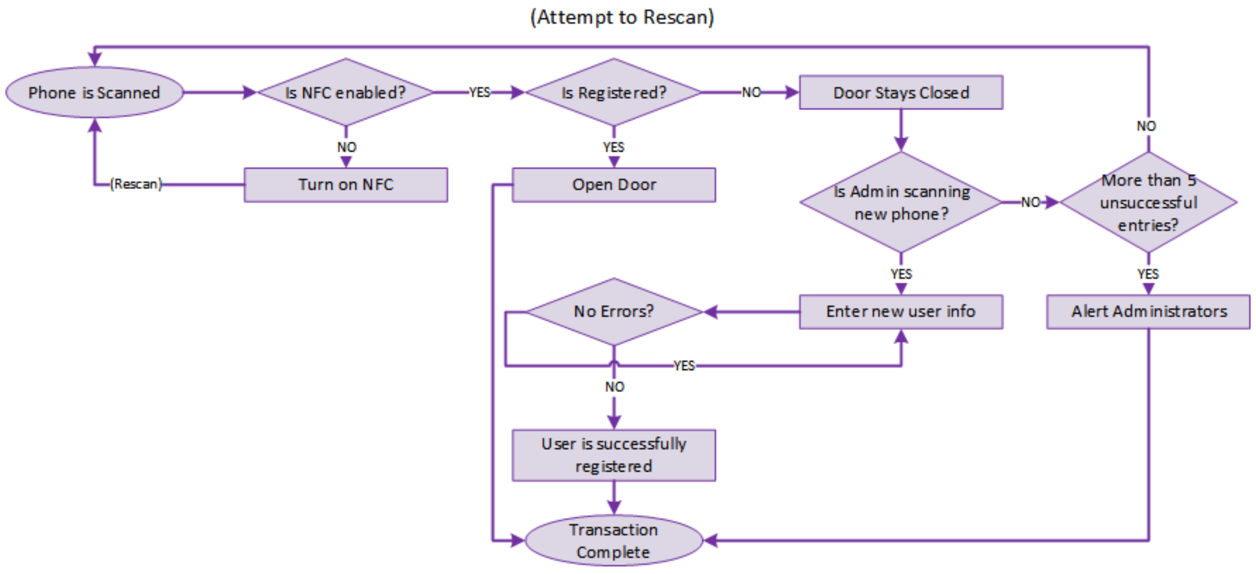
Entity Relationship Diagram (Full Implementation)



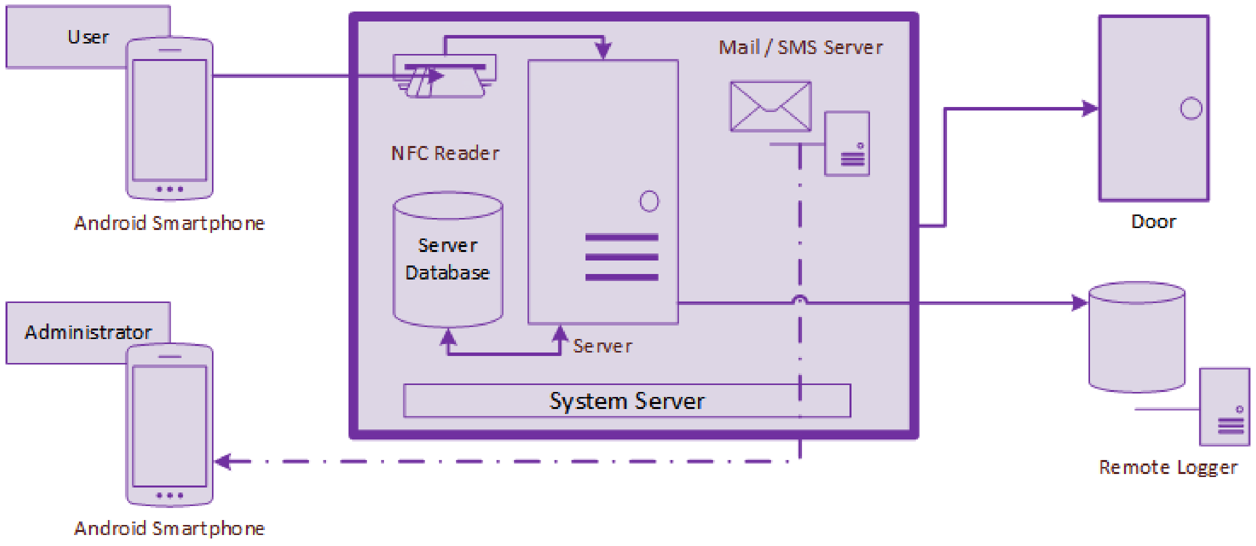
Context Diagram


User
Scans Android using NFC
Authentication System using NFC
Administrator
Administrates users and authorizes phones
NFC Reader
Retrieves user info from Android phone when scanned
System Server
• Allows or denies access
• Alerts administrator
• Manages user and user info
System Database
• Stores user info
• Contains System private key

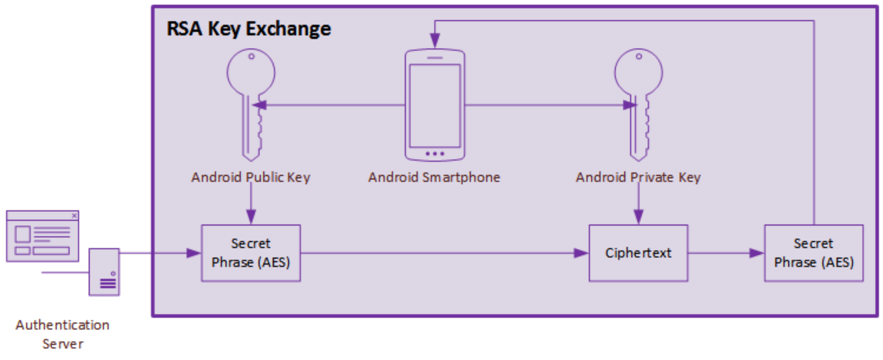
Data Flow Diagram



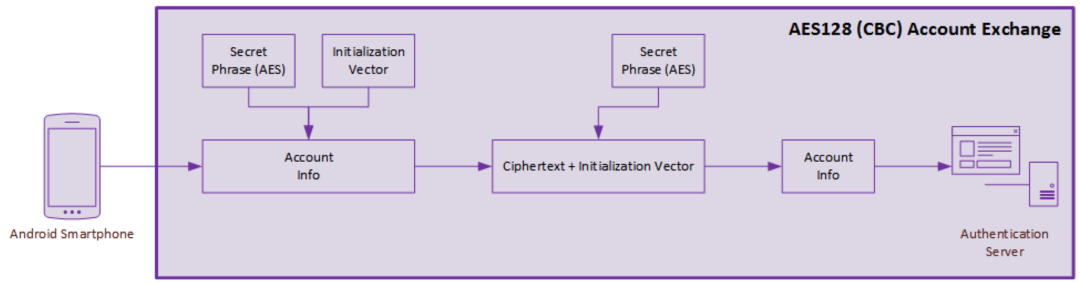
Network Architecture Diagram



RSA Key Exchange Diagram



AES Encryption & Decryption Diagram



Tools & Equipment Used

Hardware Components of the RFID / NFC Reader & NFC Medium

Part #	Hardware Name	Description / Details
1	Raspberry Pi B+ Model*	
2	Parallax RFID USB Reader	(not used after initial experimentation)*
3	RFID tags	(for testing; used after experimentation)*
4	Samsung Galaxy S4	(for P2P and Card Emulation testing)
5	Samsung Galaxy Note 4	
6	Adafruit PN532 NFC Breakout Board*	
7	Adafruit PN532 NFC Shield*	
8	NXP PN523 EXPLORE-NFC	Raspberry Pi NFC Module*
9	Seeed Studio PN532 NFC Shield V2*	
10	Dexter Industries Arduberry	Arduino Add-On (for Raspberry Pi)*
11	Arduino Uno R3	with ATMega328 Microprocessor*
12	Breadboard, Wires, Soldering Kit, Soldering Wire*	
13	MiFare Classic 1K Tags	(ISO-14433-3A compliant)*
14	NTAG213 NFC Tags	(ISO-14433-3A compliant)*

*Components are purchased for the purpose of research and experimentation of this project.

Hardware Components for Project Development

Part #	Hardware Name	Description / Details
15	MacBook Pro 15” Summer 2011 Model	● remote, mobile development
16	BCIT Lab PC (SE12-323)	
17	Windows-based Personal PC	● runs Arduino IDE ● runs Android Studio

Software Components

Part #	Software Description
18	Android Development Environment
19	Python / C++ / C languages (on Raspberry Pi)
20	Arduino IDE 1.0.6
21	Arduino NDEF & Seeed Studio Libraries
22	Arduino PN532 Libraries
23	Arduino I2C Libraries
24	Arduino SPI & Seeed Studio Libraries
25	Arduberry Libraries (drivers for Arduino to detect)
26	Linux Fedora 20 64-bit
27	Windows 8.1 64-bit
28	Apple iOS “Mavericks”
29	Linux Raspbian “Wheezy”
30	baksmali-2.0.5.jar

Project Development Details

This section discusses the potential implementations for the project with different embedded modules used. Each different implementation experiments with the technical feasibility of the project and explains why the implementation would or would not work.

Parallax RFID Reader Experimentation

The initial project proposal intended to utilize an RFID reader over an NFC reader based on the assumption that because NFC is a subset of RFID, the reader should be able to cooperate with NFC devices and tags. However, this has proven to be false because of the different frequency levels each device operates under. Below is how the system would have looked like if the project were to still utilize the RFID reader. However, this system has been scratched for the remainder of the project.

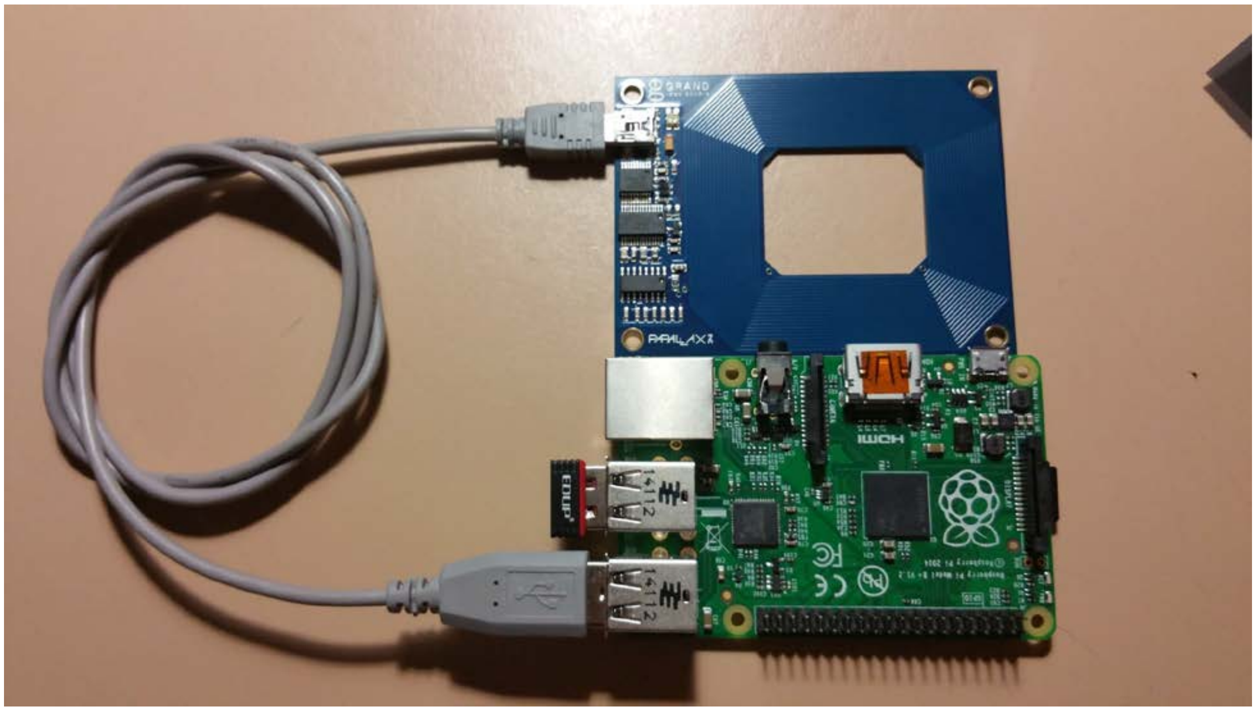


Figure 2 - Parallax RFID System

PN532 NFC Breakout Board Experimentation

Following the tutorial on Adafruit's website, these were the configurations shown in order to utilize the NFC breakout board on the Arduino. However, this configuration failed to operate because of the unnecessary level shifter. In this picture, the breakout board does not have a connection for a 5V option.

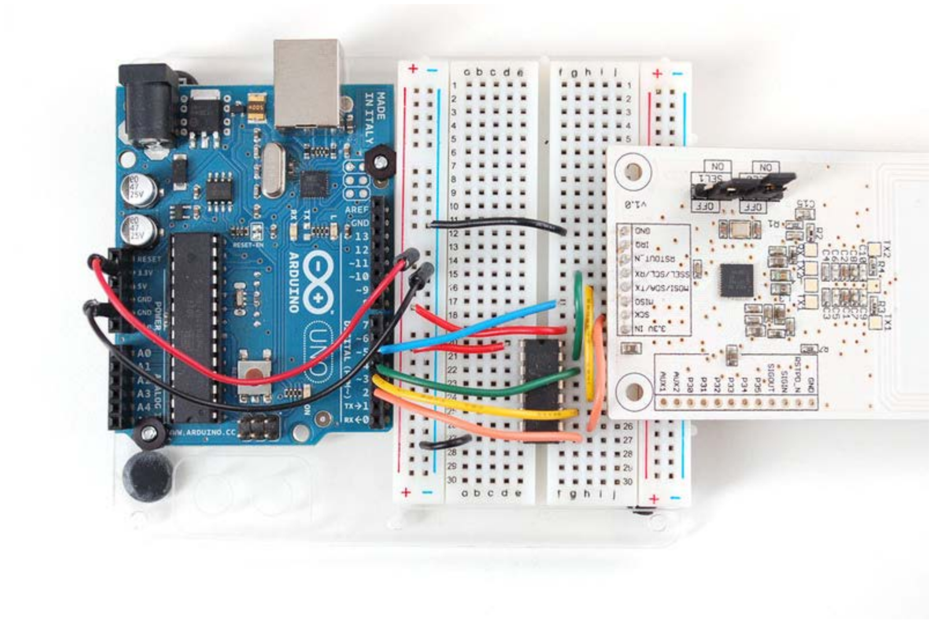


Figure 3 - PN532 Breakout Board & Arduino

Instead, it was more practical to wire each connection directly and using the 5V option on the NFC breakout board as shown below.

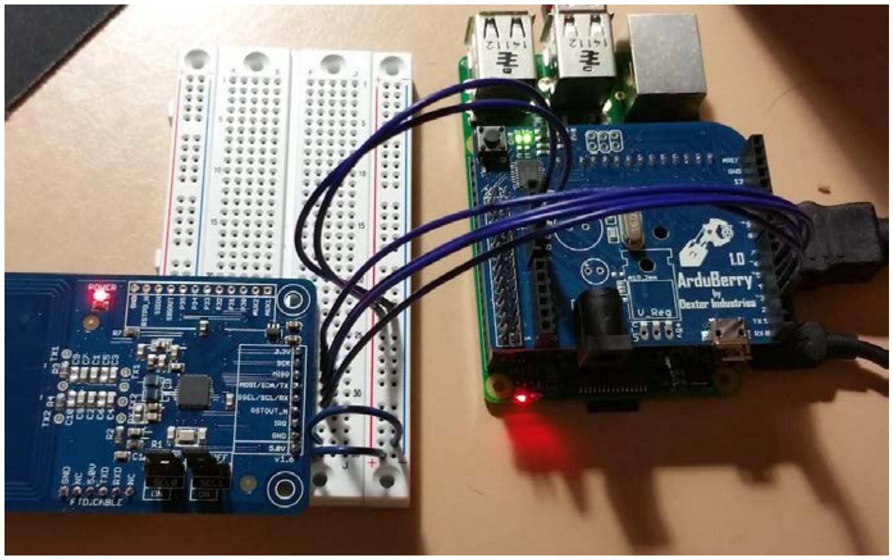


Figure 4 - PN532 Breakout Board & Arduberry

By implementing the breakout board to have each of its connectors wired to a certain pin on the Arduino (or in this case, the Arduberry) the configuration becomes much easier. However, even then, the components were insufficient and was unable to cooperate with specific SPI Arduino libraries.

PN532 NFC Shield Experimentation

Following Adafruit's tutorial on their website prompted an alternative implementation and required a different NFC reader. Adafruit's NFC *Shield* (not to be confused with the NFC *Breakout Board*) is an I2C-programmed board that utilizes all the pins on the Arduino (or in this case, the Arduberry).

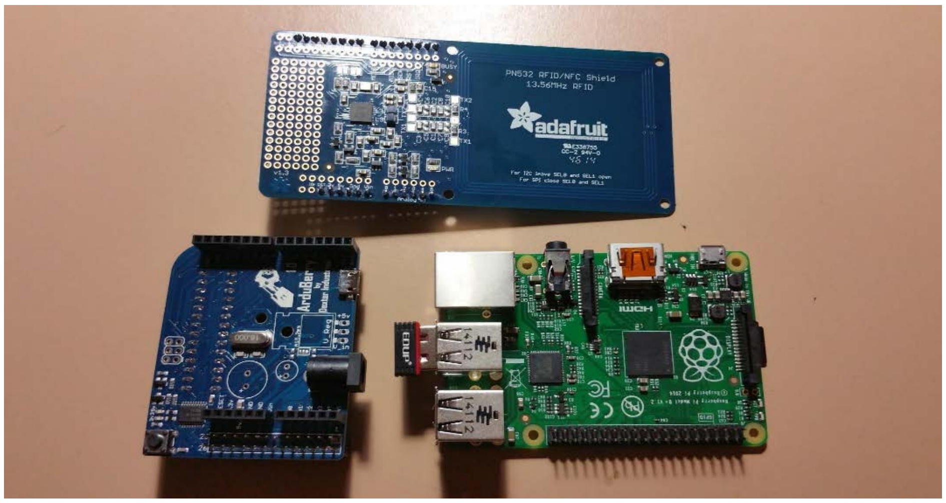


Figure 5 - PN532 Shield Components

In order to set up the system, the Pi sits on the bottom, and using its GPIO pins, the Arduberry is stacked on top. Once it is in place, the NFC Shield is placed into its respective pins on the Arduberry. The results should look like the image below:

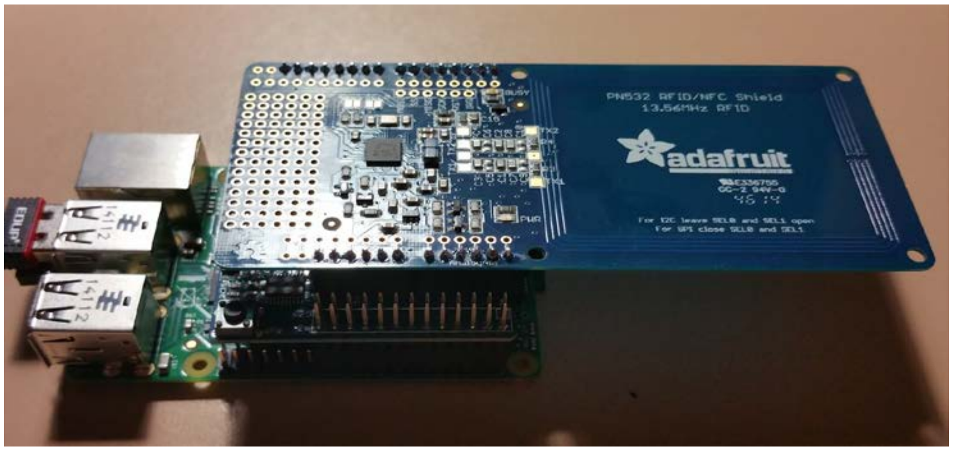


Figure 6 - PN532 Shield Assembled

Recall our **Project Caveats**; the NFC Shield is specifically programmed as an I2C device, thus disallowing it to fully cooperate with the smartphone. And while it has some libraries that have some decent functionality, the robustness of its applications are limited. However, this system architecture functions well if the purpose is to simply use passive pre-programmed NFC tags.

NXP EXPLORE-NFC Module Experimentation

The EXPLORE-NFC module is by far, one of the most powerful and effective NFC reader. It was able to communicate with many types of NFC tags, as well as being able to recognize a Canadian Passport, in addition to dumping some data from a Visa credit card. Below is the module next to the Raspberry Pi.

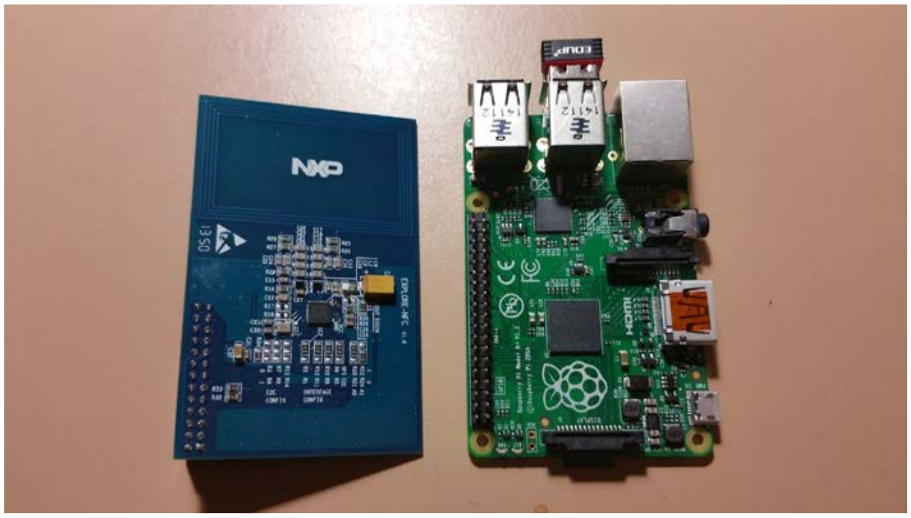


Figure 7 - NXP EXPLORE-NFC Components

The implementation of this reader is more beneficial and requires less hardware than other reader components because it simply sits on the Raspberry Pi. It also has full compliance with libnfc that is available for Linux operating systems.

After experimentation, however, the module seemed to lack examples specifically for Host-Card Emulation with an Android smartphone. Regardless, there is high optimism in the future for this module as Host-Card Emulation takes more of a prominent role and research into developing a solution using this module is highly recommended for future research. Below is what the module looks like assembled:

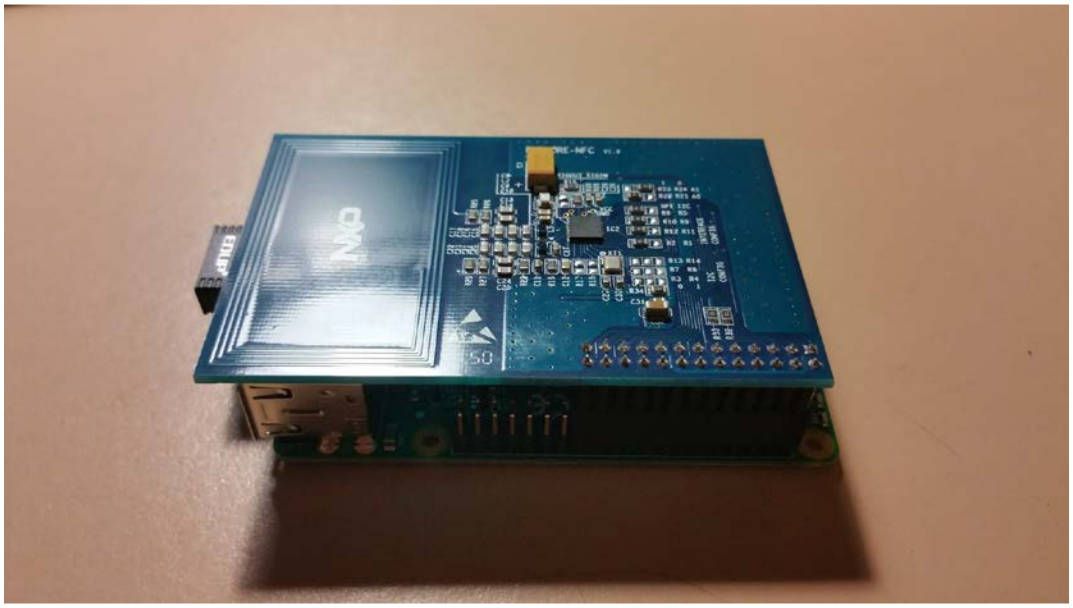


Figure 8 - NXP EXPLORE-NFC Assembled

Seeed Studio PN532 NFC Shield V2.0 Experimentation

The Seeed Studio NFC Shield was an interesting component as it used an antenna compared to the rest of the modules that utilized a board instead. For the purposes of prototyping in this project, the Seeed Studio is recommended for convenience and development purposes. However, in actual deployment, a board-like approach is much more favoured over an antenna scanner. Below are the components, utilizing the Arduberry module:

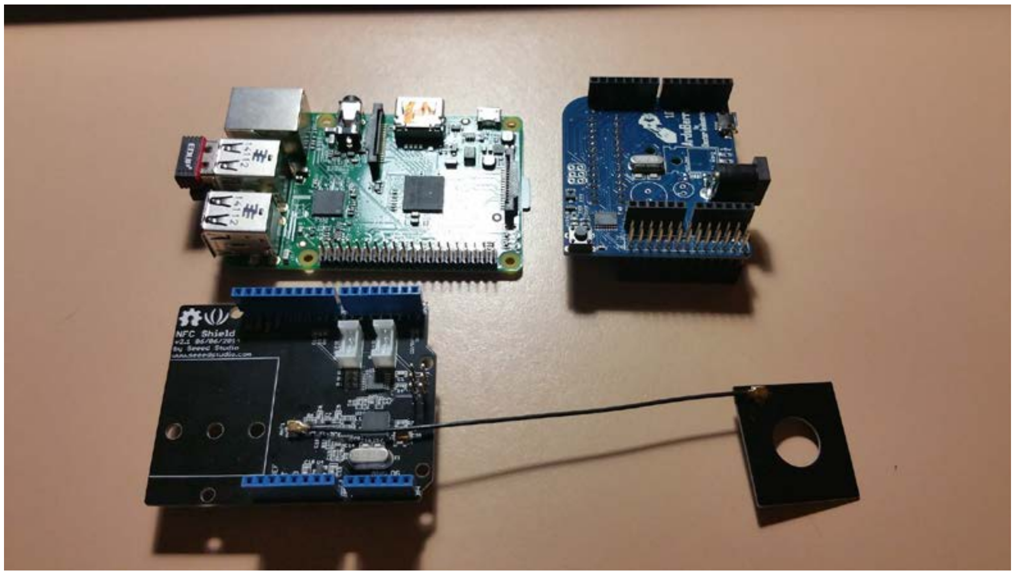


Figure 9 - Seeed Studio NFC & Arduberry

This image shows how the NFC Shield is stacked on the Arduberry, and how the two of them are stacked on the Raspberry Pi:

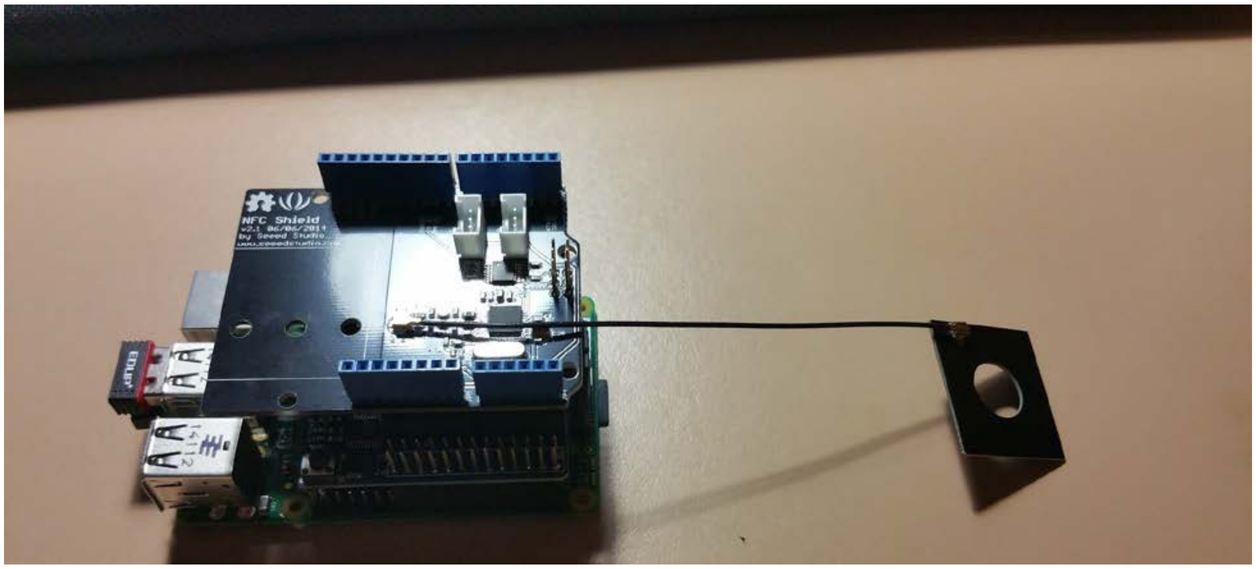


Figure 10 - Seeed Studio & Arduberry Assembled

While the above image depicts an optimistic approach, the Arduberry is manufactured with some missing and misaligned pins that the NFC Shield requires to function. This ultimately makes the Arduberry obsolete in this approach. However, some re-engineering by Dexter Industries can easily make the Arduberry fully compatible with the NFC Shield. As a workaround, we introduce the Arduino Uno instead, as seen below:

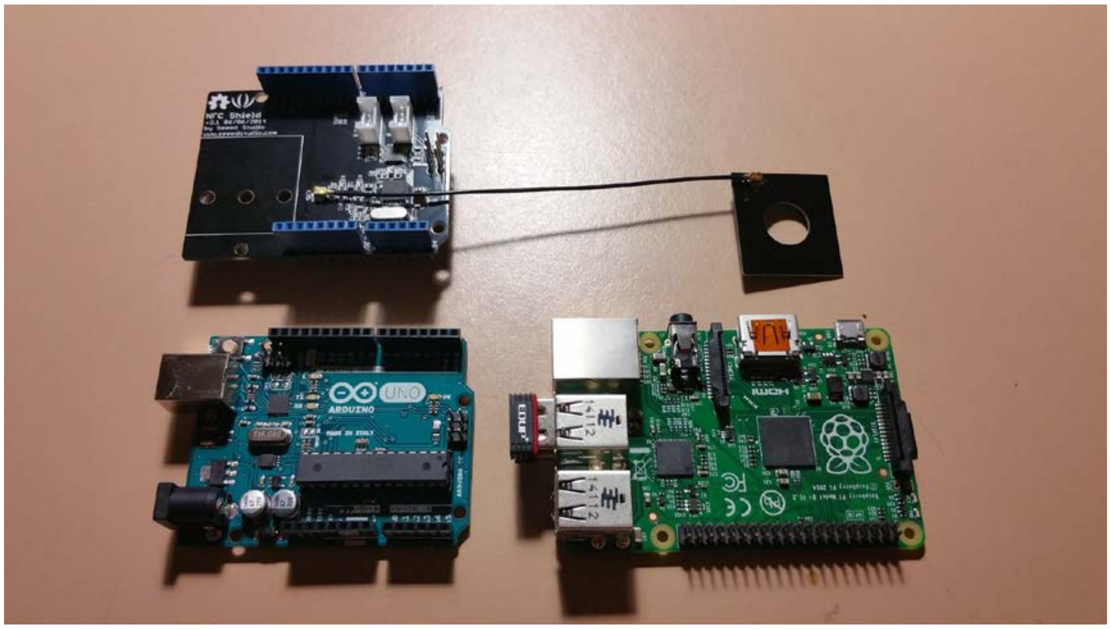


Figure 11 - Seeed Studio & Arduino

The Uno is fully compatible with the NFC Shield and the latter sits flush on top of the Uno. This workaround requires a serial USB connection between the Arduino and the Raspberry Pi, as seen in the image below:

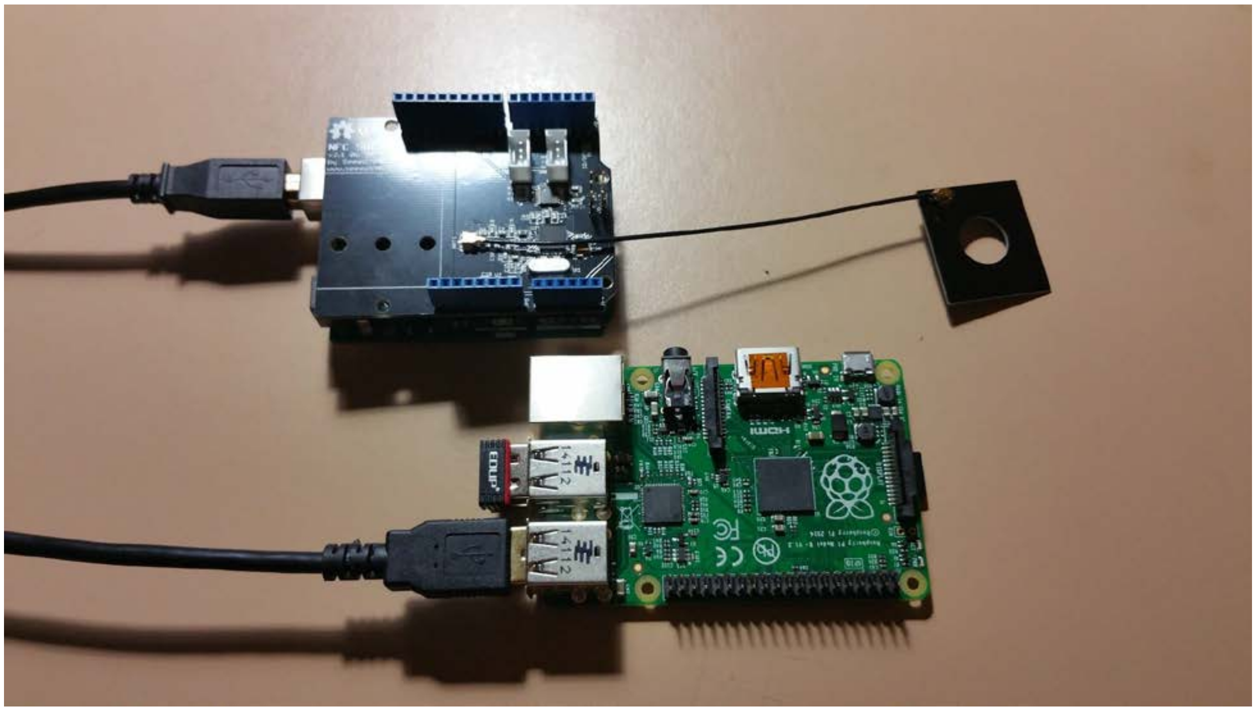


Figure 12 - Seeed Studio & Arduino Assembled

Project Test Cases

Test Cases for NFC Medium

Note: To test the NFC devices for communication capabilities, we shall be using two Android devices instead of issuing communication with the NFC reader again.

Case #	Test Case	Tools Used	Expected Outcome	Test Results
1	Android app notifies any prerequisites in order to run properly.	Android device / App notification	If NFC is turned off, there shall be a toast that asks the user to turn on NFC.	PASSED. See test result details.
2	Android app notifies any prerequisites in order to run properly.	Android device / App notification	If NFC is turned on, there shall be a toast that notifies the user that the App shall run as expected.	PASSED. See test result details.
3	CardReader and CardEmulation can find each other when devices contact each other.	Android devices	If both devices are not running their respective apps, no communication related to the apps shall occur.	PASSED. See test result details.
4	CardReader and CardEmulation can find each other when devices contact each other.	Android devices	If either device is running their respective apps but the other is not, then there shall be limited communication.	PASSED. See test result details.
5	CardReader and CardEmulation can find each other when devices contact each other.	Android devices	If both devices are running the app, then they shall attempt to communicate with each other.	PASSED. See test result details.
6	CardReader app is able to retrieve data using the same AID as the CardEmulation app.	Android device / in-app logs	If the AID is not the same between CardReader and CardEmulation apps, nothing shall be retrieved.	PASSED. See test result details.
7	CardReader app is able to retrieve data using the same AID as the CardEmulation app.	Android device / in-app logs	If the AID is the same between CardReader and CardEmulation apps, the account number shall be retrieved.	PASSED. See test result details.

Test Cases for NFC Reader

Case #	Test Case	Tools Used	Expected Outcome	Test Results
8	Arduino code can be compiled and uploaded to NFC reader	Arduino IDE	If there are any errors in the code, they shall be indicated in the Arduino IDE	PASSED. See test result details.
9	Arduino code can be compiled and uploaded to NFC reader	Arduino IDE	If there are no errors in the code, it should be uploaded successfully and is indicated in the Arduino IDE	PASSED. See test result details.
10	Arduino device can detect a connected PN532 board	Serial Monitor on Arduino / minicom	If the device is not connected, there will be an error message indicating that the device is not found.	PASSED. See test result details.
11	Arduino device can detect a connected PN532 board	Serial Monitor on Arduino / minicom	If the device is connected, there will be a message indicating that the device is found.	PASSED. See test result details.
12	Placing an NFC tag in the vicinity of the NFC reader.	Serial Monitor on Arduino / minicom	There is some form of output that shows that the NFC reader recognizes the NFC tag.	PASSED. See test result details.
13	Placing an NFC medium with NFC enabled in the vicinity of the NFC reader.	Serial Monitor on Arduino / minicom	If there is an output, it means that the NFC reader can already recognize the NFC device, otherwise it cannot recognize the device.	PASSED. See test result details.
14	When NFC device with sample CardEmulation app is near, the reader can retrieve the account number.	Serial Monitor on Arduino / minicom	If the AID specified is incorrect, there will be an error message indicating that the AID cannot be retrieved.	PASSED. See test result details.
15	When NFC device with sample CardEmulation app is near, the reader can retrieve the account number.	Serial Monitor on Arduino / minicom	If the AID specified is correct, there will be a proper dump of the account number on the NFC device.	PASSED. See test result details.

Test Cases for Smartphone-Reader-Server Use Case

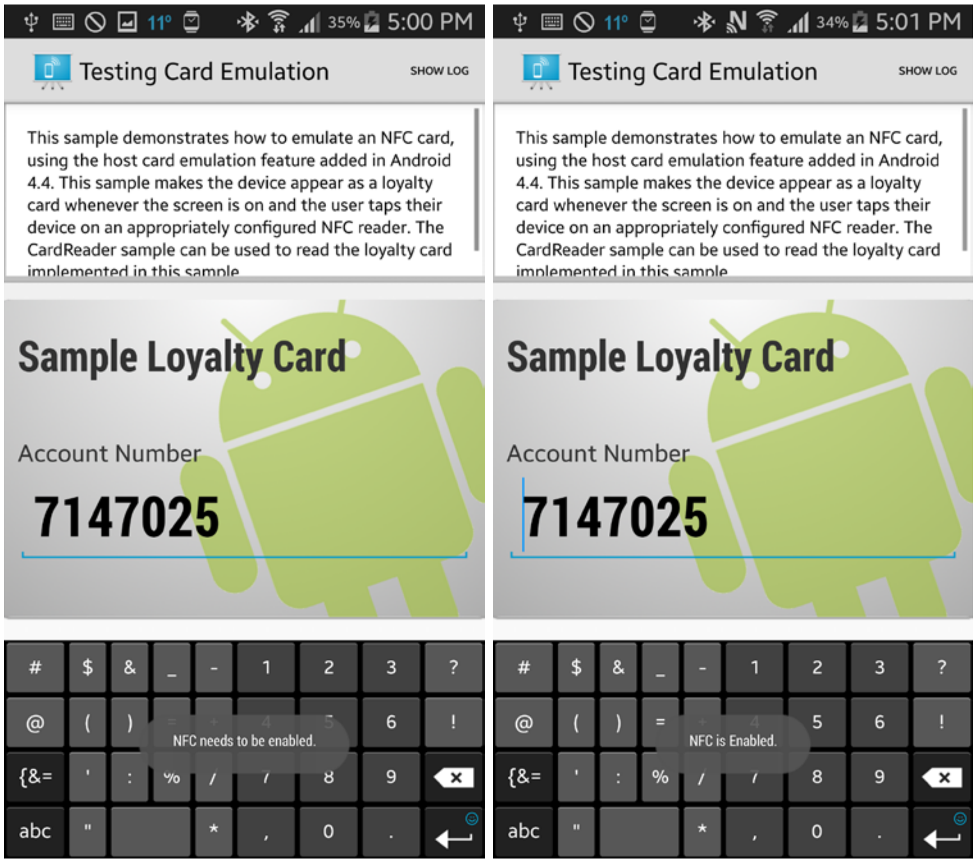
Case #	Test Case	Tools Used	Expected Outcome	Test Results
16	Smartphone is able to generate RSA key pairs	Android device / in-app logs, File Explorer	Two files shall be generated: *-private-key.txt, and *-public-key.txt	PASSED. See test result details.
17	Smartphone is able to load RSA key pairs from external files	Android device / in-app logs	The files shall be loaded and set as KeyPairs	PASSED. See test result details.
18	Smartphone is able to generate a signature based on the private key of the app	Android device / in-app logs	A signature for the data is generated based on the device's private key.	PASSED. See test result details.
19	Smartphone is able to use the generated signature to verify the contents of the data to be sent	Android device / in-app logs	Data is verified.	PASSED. See test result details.
20	Smartphone is able to encrypt the data using AES128 encryption algorithm	Android device / in-app logs	Data is able to be encrypted.	PASSED. See test result details.
21	Smartphone is able to decrypt the data using AES128 encryption algorithm	Android device / in-app logs	Data is able to be decrypted	PASSED. See test result details.
22	Smartphone is able to append the data and signatures together	Android device / in-app logs	Data and its respective signature are appended together.	PASSED. See test result details.
23	Smartphone is able to send the encrypted data over the NFC reader	Arduino serial monitor	The Arduino is able to parse the data contents and display them on the serial monitor	PASSED. See test result details.
24	Python server is able to retrieve the serial contents of the Arduino	Command Line / Terminal	Command Line / Terminal displays the serial data of the Arduino; these should be identical to the Arduino monitor	PASSED. See test result details.
25	Python server is able to split the ciphertext data and its signature	Command Line / Terminal	Command Line / Terminal distinctly displays the ciphertext and signature separately	PASSED. See test result details.
26	Python server is able to decrypt the encrypted data	Command Line / Terminal	Command Line / Terminal displays the decrypted data; there should distinctly be plaintext data and its digital signature appended to its end	PASSED. See test result details.

27	Python server is able to verify the data using its signature	Command Line / Terminal	If the public key of the Android app is not the same as the public key located on the Python server, the Command Line / Terminal shall deny the authenticity of the data	PASSED. See test result details.
28	Python server is able to verify the data using its signature	Command Line / Terminal	Command Line / Terminal confirms the authenticity of the data	PASSED. See test result details.

Test Case Results

Test Case #1

In this test, we added some code into the CardEmulation app that allowed us to be notified whether the proper components were enabled and loaded. In this test case, we have purposely disabled NFC on our smartphone. Upon launching the app, we received a Toast stating that we need to enable NFC. At this point, there are no API calls that enable NFC remotely from the Settings screen. There is a distinct difference, because on the left image we do not have an NFC symbol compared to the right image.



This sample demonstrates how to emulate an NFC card, using the host card emulation feature added in Android 4.4. This sample makes the device appear as a loyalty card whenever the screen is on and the user taps their device on an appropriately configured NFC reader. The CardReader sample can be used to read the loyalty card implemented in this sample

NFC needs to be enabled.

This sample demonstrates how to emulate an NFC card, using the host card emulation feature added in Android 4.4. This sample makes the device appear as a loyalty card whenever the screen is on and the user taps their device on an appropriately configured NFC reader. The CardReader sample can be used to read the loyalty card implemented in this sample

NFC is Enabled.

Figure 13.1, 12.2 - NFC disabled & enabled

Test Case #2

In this test, we added some code into the CardEmulation app that allowed us to be notified whether the proper components were enabled and loaded. Unlike in Test Case #1, we have enabled NFC. Upon launching the app, we received a Toast stating that NFC is enabled. Note that we have NFC enabled by having the NFC symbol in our notification bar. Refer to the images in the previous test case.

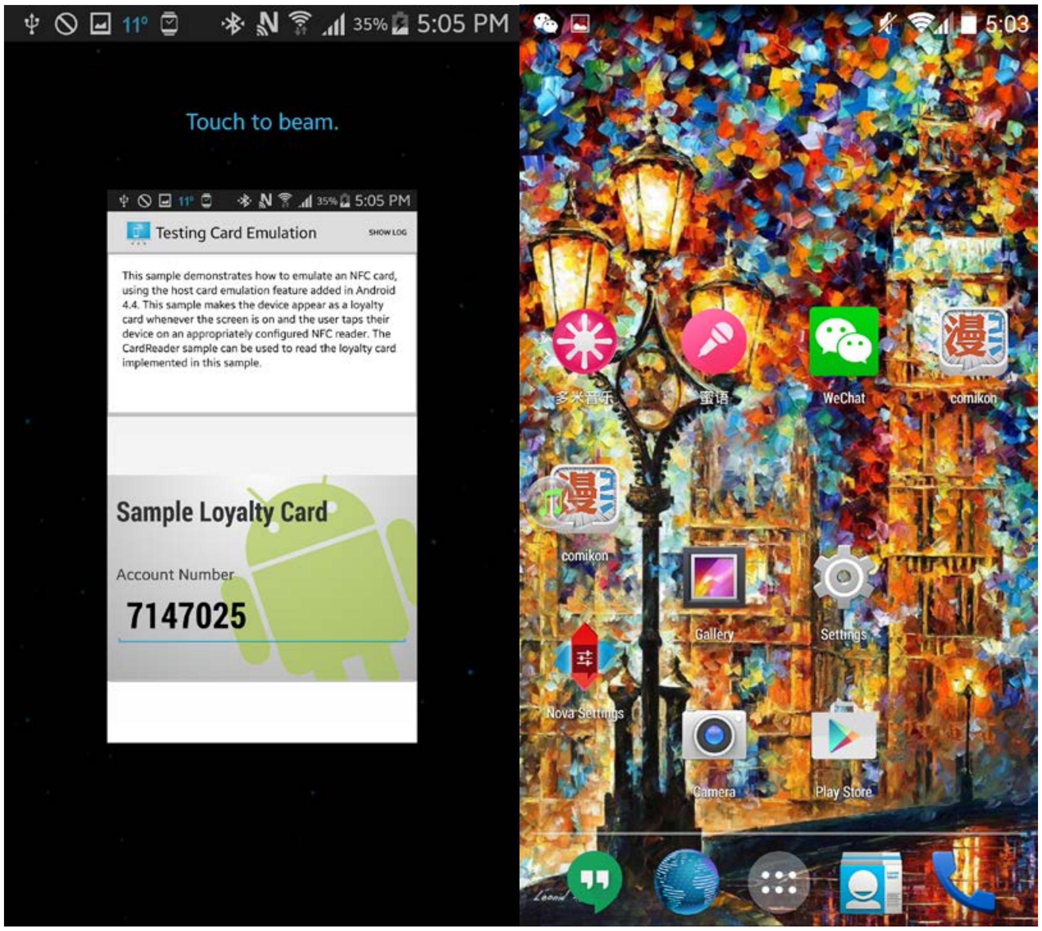
Test Case #3

This test case requires that both sides have their respective apps installed. However, both Android phones shall not be running either of their respective apps in the background or foreground. Below are the two screenshots from each phone and that there are no communications whatsoever when placed back-to-back.

Figure 14.1, 13.2 - No NFC Communication

Test Case #4

For this test case, we alternated app availability on both Android phones. To clarify, the Samsung Galaxy Note 4 shall have the app in the foreground and its counterpart shall have their app disabled or in the background, and then vice-versa in another session. Below are the screenshots of the first session:


Touch to beam.

Testing Card Emulation

SHOW LOG

This sample demonstrates how to emulate an NFC card, using the host card emulation feature added in Android 4.4. This sample makes the device appear as a loyalty card whenever the screen is on and the user taps their device on an appropriately configured NFC reader. The CardReader sample can be used to read the loyalty card implemented in this sample.

Sample Loyalty Card

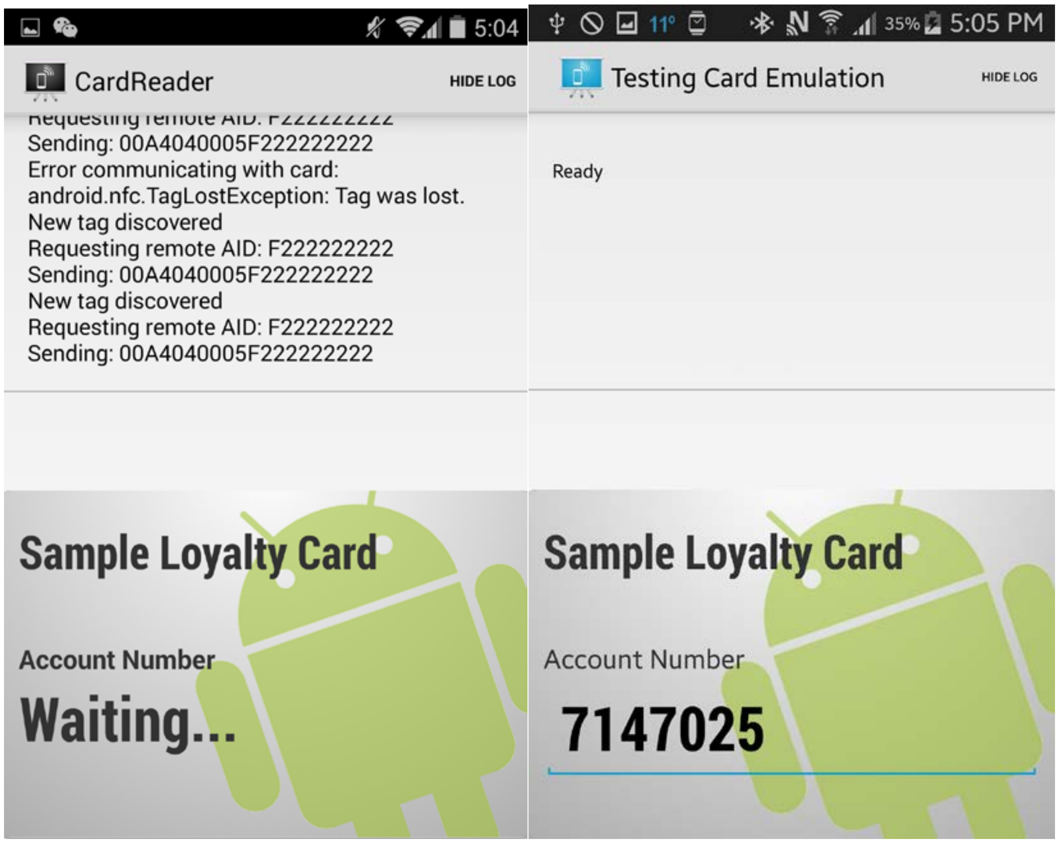
Account Number

7147025

Figure 15.1, 14.2 - One-sided NFC enabled

In this session, the Note 4 tries to communicate with the S4 but instead initiates Android Beam to allow the S4 to access Google Play Store to download this app. We can think of this as an override when certain conditions are met (or in this case, have not been met).

However, we do not wish to beam the information across as it is not our true intention. We conclude that there is some form of limited communication. Conversely, refer to the screenshot below when the roles have been reversed:



```
Requesting remote AID: F222222222
Sending: 00A4040005F222222222
Error communicating with card:
android.nfc.TagLostException: Tag was lost.
New tag discovered
Requesting remote AID: F222222222
Sending: 00A4040005F222222222
New tag discovered
Requesting remote AID: F222222222
Sending: 00A4040005F222222222
```

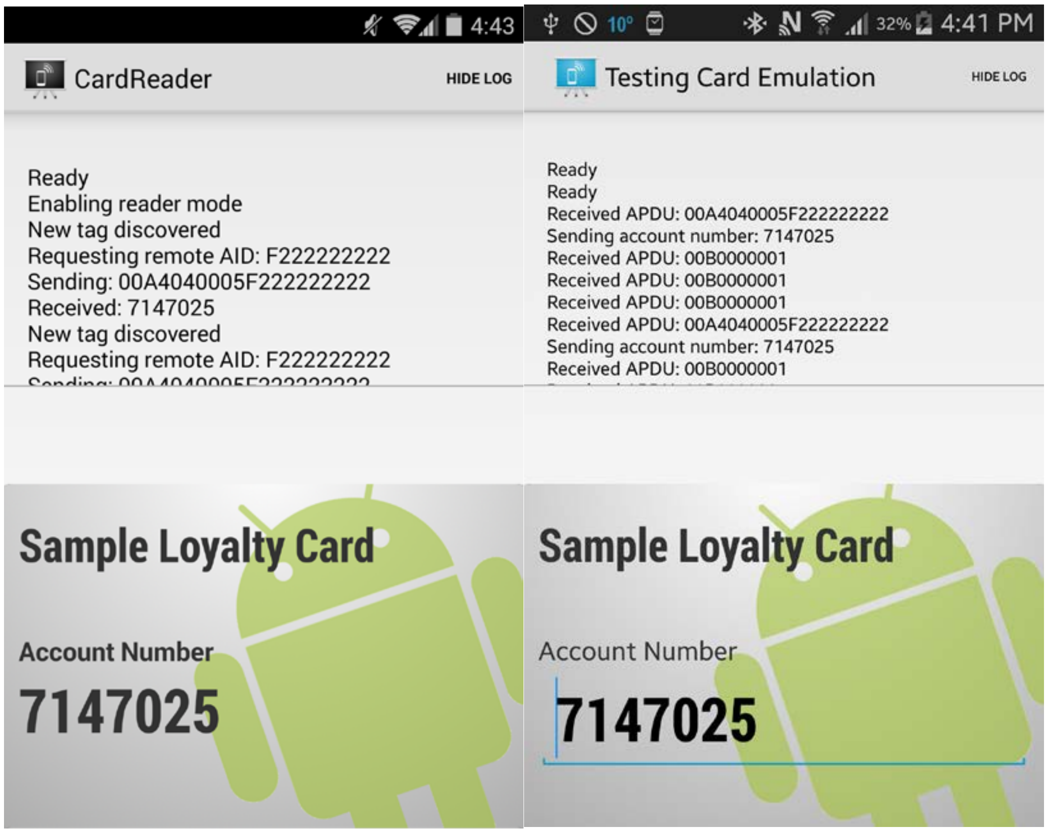
```
Ready
```

Figure 16.1, 15.2 - Role Reversal One-sided NFC enabled

What occurs in this session is the CardReader discovers an NFC reader, and attempts to send a request to pull information from the CardEmulation app. However, because it was in the background, the CardReader receives nothing. A post-experiment screenshot on the Note 4 (right) depicts that it had not received any communication attempts from the CardReader app.

Test Case #5

Conversely, once both smartphones have the app running in the foreground, information is transferred accordingly:



```
Ready
Enabling reader mode
New tag discovered
Requesting remote AID: F222222222
Sending: 00A4040005F222222222
Received: 7147025
New tag discovered
Requesting remote AID: F222222222
```

```
Ready
Ready
Received APDU: 00A4040005F222222222
Sending account number: 7147025
Received APDU: 00B0000001
Received APDU: 00B0000001
Received APDU: 00B0000001
Received APDU: 00A4040005F222222222
Sending account number: 7147025
Received APDU: 00B0000001
```

Figure 17.1, 16.2 - CardReader & CardEmulation

We know that this is successful when the Loyalty Card component shows the correct number on the CardReader app.

Test Case #6

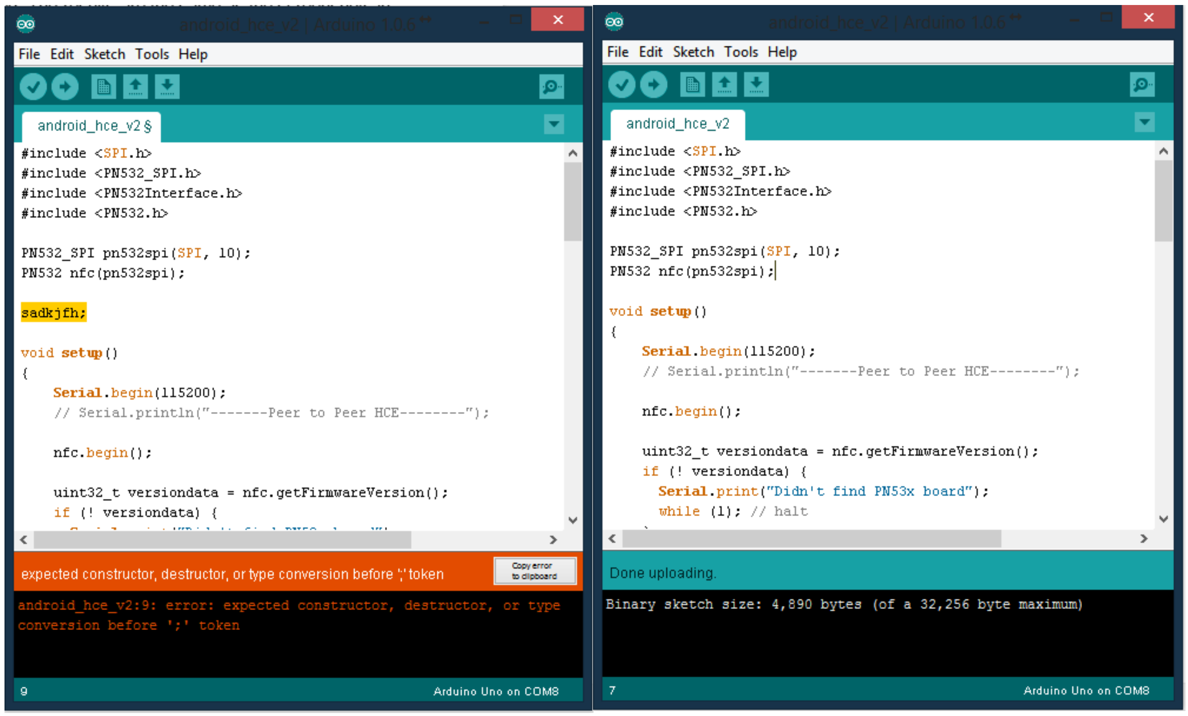
When the Android Identification number (AID) is mismatched on either side of the application, there is limited communication involved, similar to Test Case #4. This number is hard-coded into the source code of the application, and it should not be changed during deployment or runtime. This is a security feature built into Host-Card Emulation in Android 4.4 Kitkat. Refer to the screenshots of Test Case #4 for evidence.

Test Case #7

Once the AID is the same on both clients, the communication is successful assuming all other prerequisites and conditions have been fulfilled. The results are the same as the screenshots in Test Case #5.

Test Case #8

To test the Arduino code and its IDE, we have based our code off of an example provided by Seeed Studio's developers. In this test case, we have purposely added an error into the source code. Below is the screenshot to highlight the error and the evidence:



```
#include <SPI.h>
#include <PN532_SPI.h>
#include <PN532Interface.h>
#include <PN532.h>

PN532_SPI pn532spi(SPI, 10);
PN532 nfc(pn532spi);

sadkjfh;

void setup()
{
    Serial.begin(115200);
    // Serial.println("-------Peer to Peer HCE--------");

    nfc.begin();

    uint32_t versiondata = nfc.getFirmwareVersion();
    if (! versiondata) {
```

```
expected constructor, destructor, or type conversion before ';' token
android_hce_v2:9: error: expected constructor, destructor, or type
conversion before ';' token
9                                                  Arduino Uno on COM8
```

```
#include <SPI.h>
#include <PN532_SPI.h>
#include <PN532Interface.h>
#include <PN532.h>

PN532_SPI pn532spi(SPI, 10);
PN532 nfc(pn532spi);

void setup()
{
    Serial.begin(115200);
    // Serial.println("-------Peer to Peer HCE--------");

    nfc.begin();

    uint32_t versiondata = nfc.getFirmwareVersion();
    if (! versiondata) {
      Serial.print("Didn't find PN53x board");
      while (1); // halt
```

```
Done uploading.
Binary sketch size: 4,890 bytes (of a 32,256 byte maximum)
7                                                  Arduino Uno on COM8
```

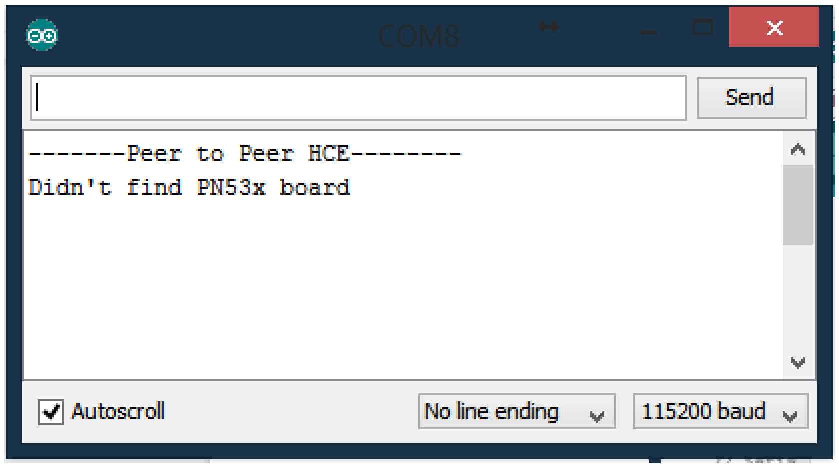
Figure 18.1, 17.2 - Arduino IDE with, without Error

Test Case #9

Continuing from Test Case #8, once we have removed the error in the previous screenshot that was highlighted by the IDE, the compiling of the source code is complete. The IDE then uploads the compiled program to the Arduino. See the following screenshot for evidence:

Test Case #10

Once the program is uploaded to the Arduino, we want to see some serial output from the system before we can continue with further testing. In this test case, we have purposely removed the Seeed Studio NFC shield to see if the Arduino is smart enough to determine that it is missing. Below is the evidence that supports this claim:



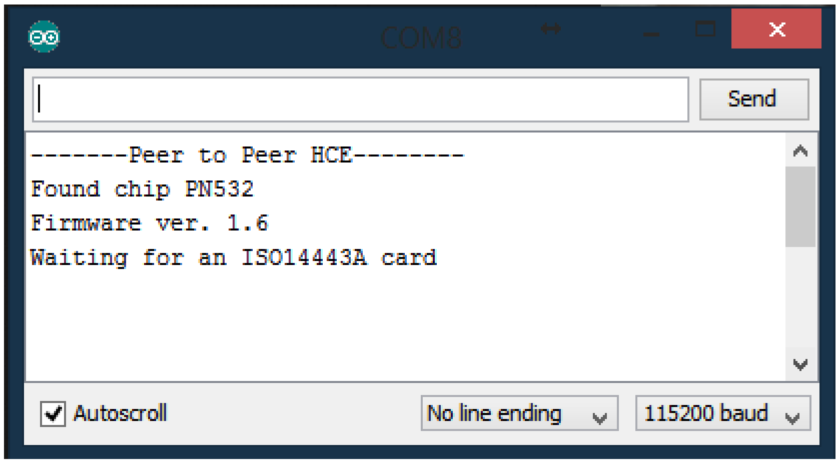
```
-------Peer to Peer HCE--------
Didn't find PN53x board
```

Figure 19 - Arduino Cannot find NFC Reader

As expected, because the shield was not connected to the Arduino, the program is incapable of finding the right hardware components.

Test Case #11

In contrast to Test Case #10, we have reconnected the shield to the Arduino. As expected the Arduino is able to recognize and register the shield into its system and the program reflects this in its output. The screenshot provided below supports this expectation:



```
-------Peer to Peer HCE--------
Found chip PN532
Firmware ver. 1.6
Waiting for an ISO14443A card
```

Figure 20 - Arduino Successfully Found NFC Reader

In the previous test case, after it was unable to find the shield, the program immediately goes into a state of “halt”. In this case, the program detects the shield, and then transitions into a loop where it forever looks for nearby NFC tags until something stops it.

Test Case #12

In this test case, instead of using the Android smartphone with the CardEmulation app we shall test the Arduino program by using passive NFC tags. In the below screenshot, we have some sample tags; some are empty tags, others are not. In this test case, we are indifferent with the content of the tags.

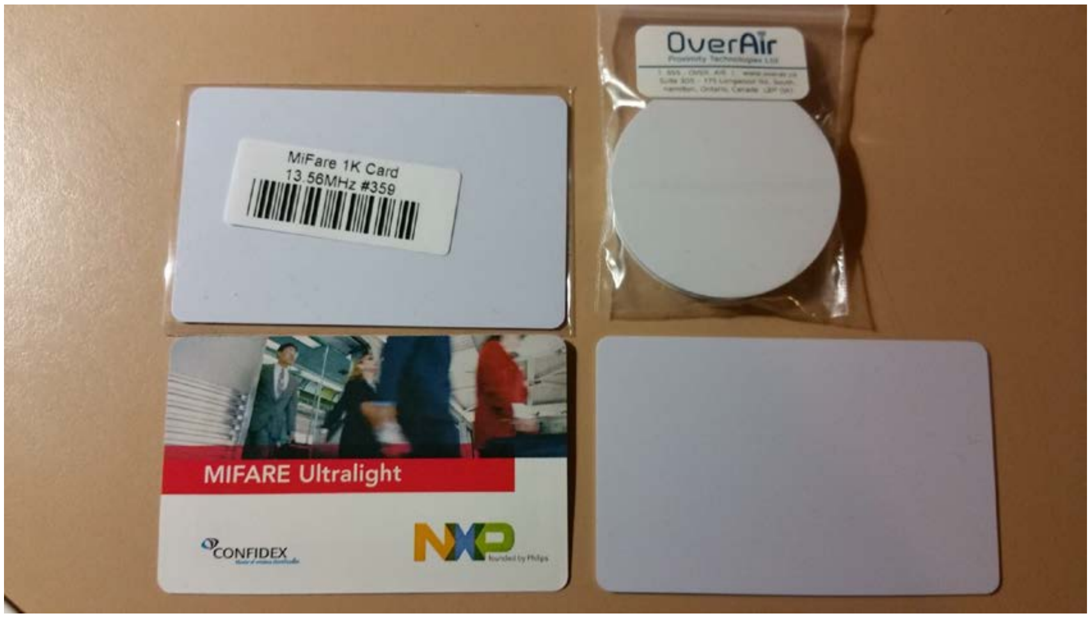
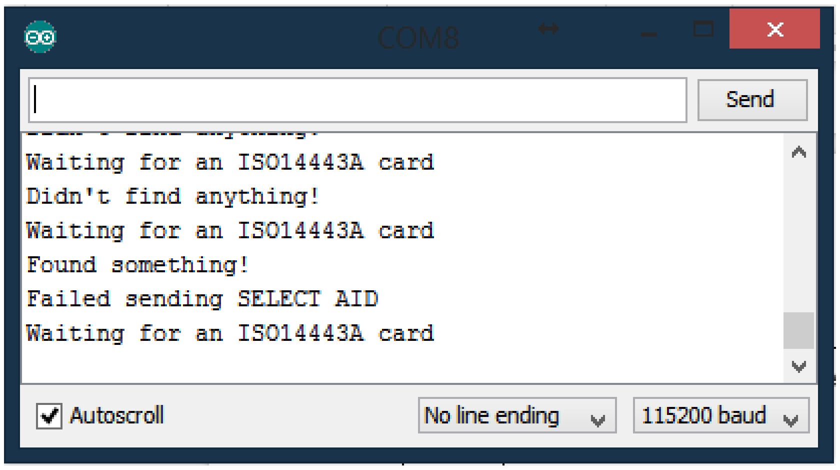


Figure 21 - Various NFC Tags

From left to right, top to bottom each of the tags are: a MiFare 1(kilobyte) Classic Card, a circular NTAG213-formatted NFC tag, a MiFare Ultralight card by NXP, and finally a rectangular NTAG213-formatted NFC card. Of these four tags, only the MiFare Ultralight card has any data. Regardless, when placed in the vicinity of the NFC reader's antenna, the resulting screenshot is shown:



```
Waiting for an ISO14443A card
Didn't find anything!
Waiting for an ISO14443A card
Found something!
Failed sending SELECT AID
Waiting for an ISO14443A card
```

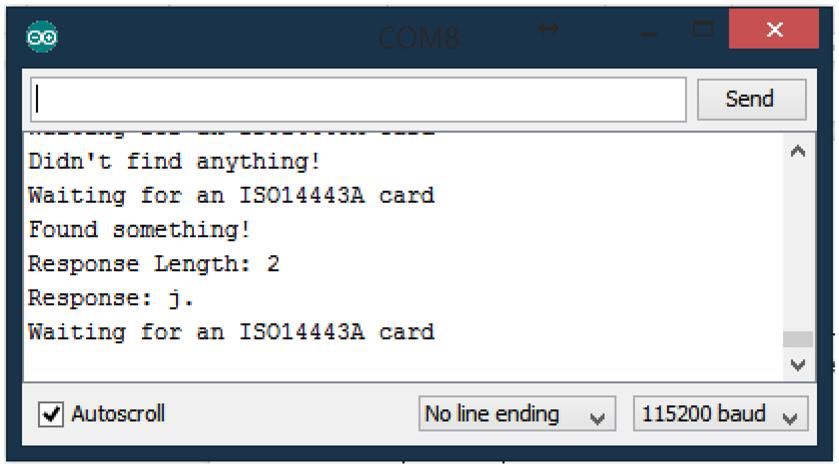
Autoscroll: ✓ · No line ending · 115200 baud

Figure 22 - Detecting NFC Tags

To elaborate, the Arduino and its shield were able to find an NFC-compliant card. This is found through the serial output “Found something!”. However, it fails to continue its programmed duties since each of these tags are passive, so it could not send a request for an Android ID. Once it fails, it delays for one second before it reestablishes its loop.

Test Case #13

Before we actually use our Samsung Galaxy Note 4 (that has the CardEmulation app), we shall attempt to use a smartphone that does not have card emulation capabilities, but does have NFC capabilities. In this test case, we took our Samsung Galaxy S4 and scanned it to our NFC reader. Below is the serial screenshot of what was outputted by the Arduino:



```
Didn't find anything!
Waiting for an ISO14443A card
Found something!
Response Length: 2
Response: j.
Waiting for an ISO14443A card
```

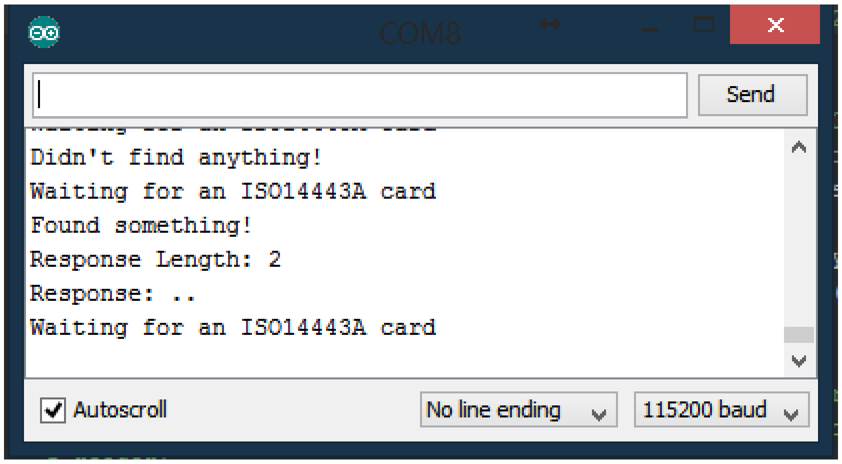
☑ Autoscroll No line ending 115200 baud

Figure 23 - Smartphone without CardEmulation App

Here, we were able to get past the “Failed sending SELECT AID” error and were able to retrieve some form of a response. On our Android phone, what probably happened is that, because no service had that specific AID, it responded with a “reject” message. This confirms that the NFC reader could potentially read from NFC-enabled smartphones.

Test Case #14

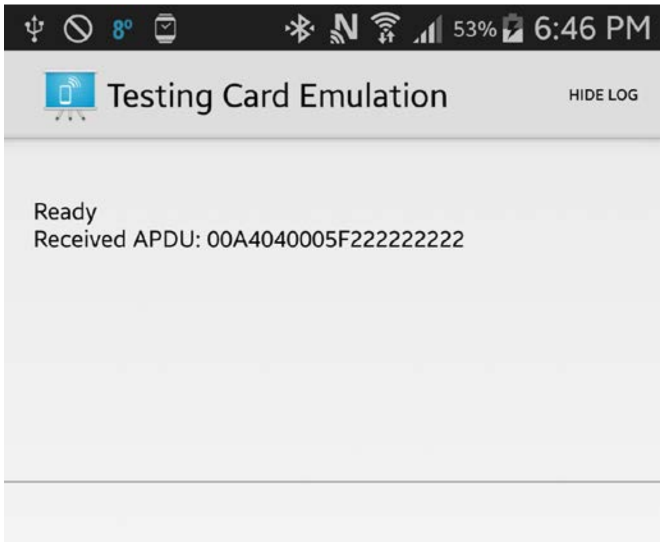
In this test case, we have purposely changed the AID on our Android app and re-uploaded it to our Note 4 before scanning it to the reader. Below is the screenshot from the Arduino serial monitor:



```
Didn't find anything!
Waiting for an ISO14443A card
Found something!
Response Length: 2
Response: ..
Waiting for an ISO14443A card
```

Figure 24 - Inconsistent AID

Interestingly, it shows a different output than from Test Case #13. We assume that the error is similar to this test case or to Test Case #13 when the SELECT AID is changed on the Arduino side. For further evidence, below is a screenshot of the CardEmulation app after changing the AID and scanning it:



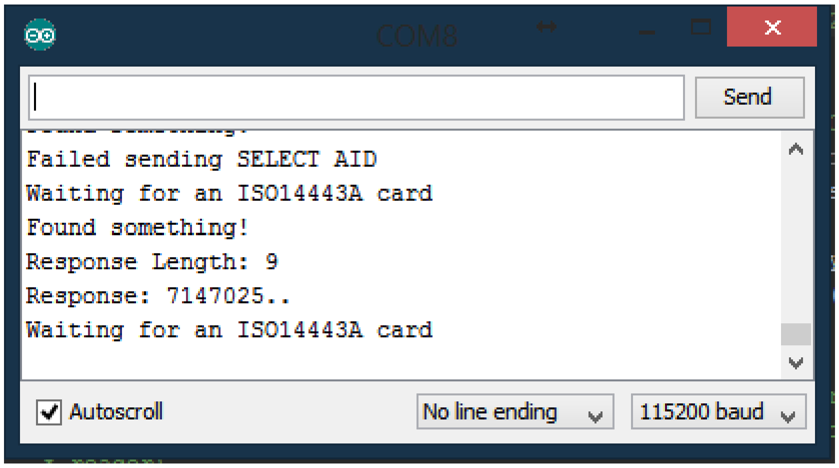
```
Ready
Received APDU: 00A4040005F222222222
```

Figure 25 - In-App Log for CardEmulation

This is interesting because the app is able to receive the AID request. However, because the request is incorrect, the app simply “ignores” it after evaluation.

Test Case #15

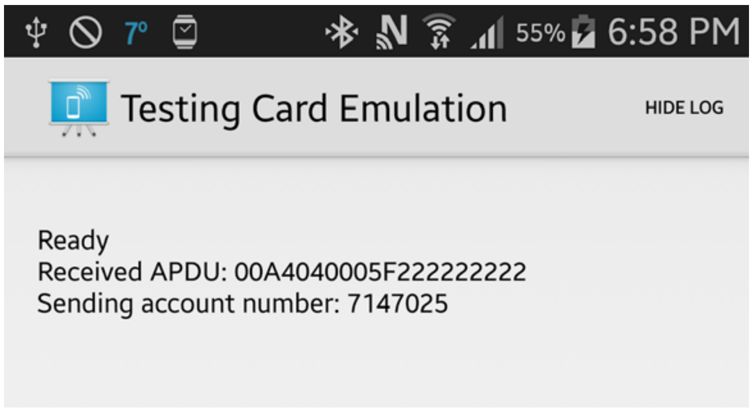
This time, we changed it back to the correct AID that is agreed upon between both the Arduino program and the CardEmulation app. Below is the screenshot after scanning the smartphone with the NFC reader:



```
Failed sending SELECT AID
Waiting for an ISO14443A card
Found something!
Response Length: 9
Response: 7147025..
Waiting for an ISO14443A card
```

Figure 26 - Consistent AID

As expected, the information is transferred to the Arduino, which is then able to output the serial values into the serial monitor. On the Android side, we are able to see a similar process. Refer to the screenshot below for details:



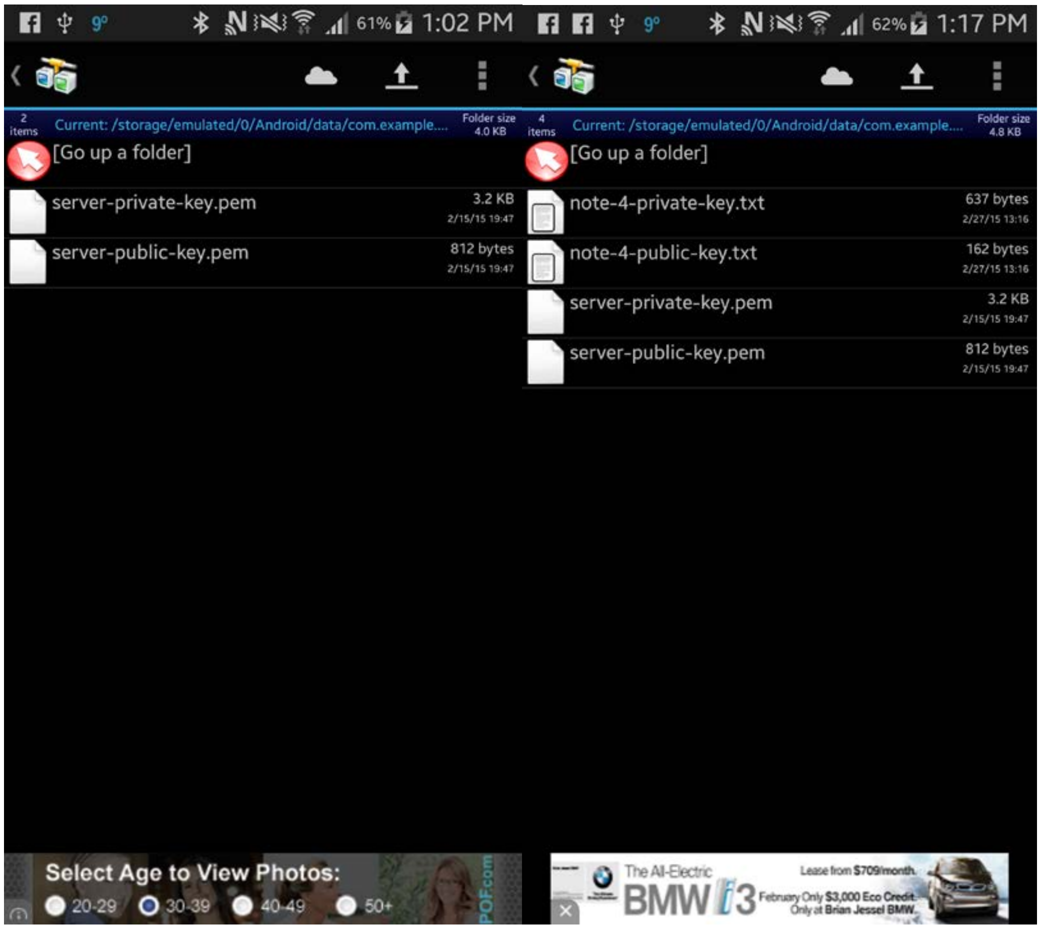
```
Ready
Received APDU: 00A4040005F222222222
Sending account number: 7147025
```

Figure 27 - App Sending Account Number

The noticeable difference here is the third line in the log output; it is missing in Test Case #14. Because the AID selection is correct (from “F2…” onwards), the app recognizes the AID selection, and processes the request by sending the account number over. We can confirm in this test case that the NFC reader is capable of reading the smartphone, and the smartphone is capable of sending NFC information to the reader. We are now ready to begin implementing and testing other components of our project.

Test Case #16

To continue with the following test cases, we shall remove the app from the smartphone and reinstall. This way, we can destroy pre-existing key pairs for the phone. After reinstalling the app onto the smartphone, we can see (through an FTP application) that the keys do not exist. Upon scanning the smartphone with the NFC reader, the smartphone generates a public key and private key based an account credentials. Consult the following two screenshots below:



```
2 items   Current: /storage/emulated/0/Android/data/com.example....   Folder size 4.0 KB
[Go up a folder]
server-private-key.pem                3.2 KB     2/15/15 19:47
server-public-key.pem                 812 bytes  2/15/15 19:47
```

```
4 items   Current: /storage/emulated/0/Android/data/com.example....   Folder size 4.8 KB
[Go up a folder]
note-4-private-key.txt                637 bytes  2/27/15 13:16
note-4-public-key.txt                 162 bytes  2/27/15 13:16
server-private-key.pem                3.2 KB     2/15/15 19:47
server-public-key.pem                 812 bytes  2/15/15 19:47
```

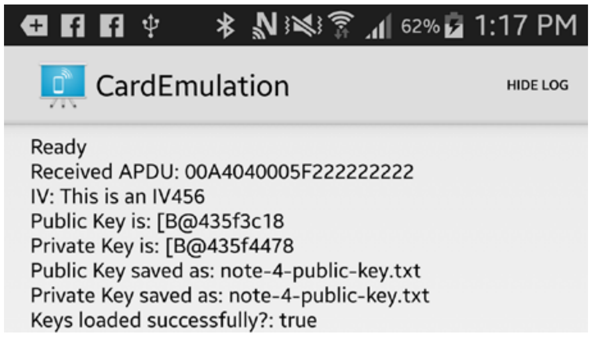
Figure 28.1, 27.2 - Creating New Key Pairs

We can see that the keys were generated by the app, because the keys reside in the app's native folder on the Android internal storage.

Test Case #17

Next, the app is required to load these keys if the files exist. To do that, every time the Android smartphone is scanned on the NFC reader, the app checks if these files exist by comparing the names of the files to the account they're associated with.

If they do not exist, they shall create a new set of keys based on the file names associated with the account. Otherwise, they load the external key files. In the logs of the application, we can see that the files were loaded successfully:

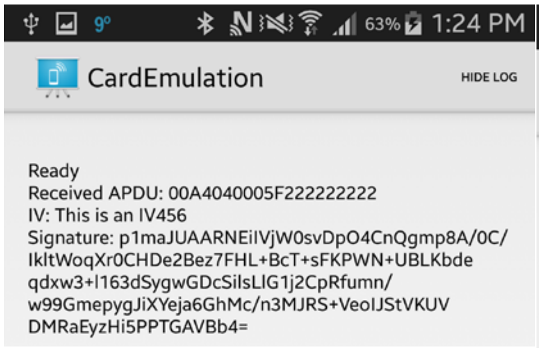


```
Ready
Received APDU: 00A4040005F222222222
IV: This is an IV456
Public Key is: [B@435f3c18
Private Key is: [B@435f4478
Public Key saved as: note-4-public-key.txt
Private Key saved as: note-4-public-key.txt
Keys loaded successfully?: true
```

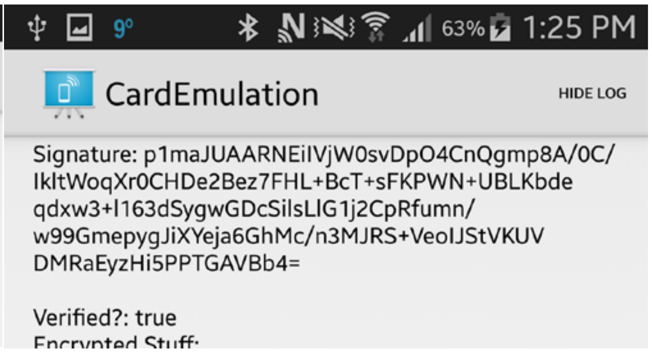
Figure 29 - Logs of Creating External Key Files

Test Case #18

Once the keys are loaded, the app transitions to creating a signature based on the account's private key that it loaded. It creates a digest after taking inputs of the account's username and number. Consult the following screenshot:



```
Ready
Received APDU: 00A4040005F222222222
IV: This is an IV456
Signature: p1maJUAARNEiIVjW0svDpO4CnQgmp8A/0C/
IkltWoqXr0CHDe2Bez7FHL+BcT+sFKPWN+UBLKbde
qdxw3+l163dSygwGDcSilsLIG1j2CpRfumn/
w99GmepygJiXYeja6GhMc/n3MJRS+VeoIJStVKUV
DMRaEyzHi5PPTGAVBb4=
```



```
Signature: p1maJUAARNEiIVjW0svDpO4CnQgmp8A/0C/
IkltWoqXr0CHDe2Bez7FHL+BcT+sFKPWN+UBLKbde
qdxw3+l163dSygwGDcSilsLIG1j2CpRfumn/
w99GmepygJiXYeja6GhMc/n3MJRS+VeoIJStVKUV
DMRaEyzHi5PPTGAVBb4=

Verified?: true
Encrypted Stuff:
```

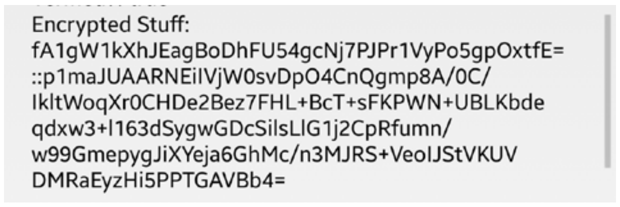
Figure 30.1, 29.2 - Signature Digest, Signature Verification

Test Case #19

The app then does a quality check on the signature and verifies its validity and authenticity based on its digest and the data. The app then logs the result of the check, as seen in the screenshot above, on the right. The result should always be true unless somehow the private key of the account holder suddenly changes, or there is a corruption in the generation of the signature.

Test Case #20

Next, the app encrypts the account's username and number string, as seen below:



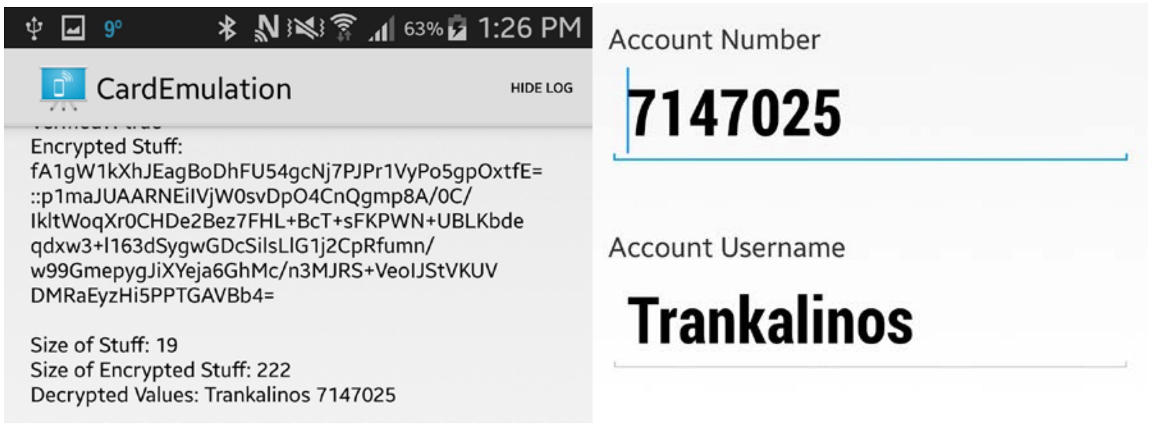
```
Encrypted Stuff:
fA1gW1kXhJEagBoDhFU54gcNj7PJPr1VyPo5gpOxtfE=
::p1maJUAARNEiIVjW0svDpO4CnQgmp8A/0C/
IkltWoqXr0CHDe2Bez7FHL+BcT+sFKPWN+UBLKbde
qdxw3+l163dSygwGDcSilsLIG1j2CpRfumn/
w99GmepygJiXYeja6GhMc/n3MJRS+VeoIJStVKUV
DMRaEyzHi5PPTGAVBb4=
```

Figure 31 - Encrypted Data

Note that after the double colon ("::") in the string, we see the signature from Test Case #19. Consult Test Case #22 for further details.

Test Case #21

Similar to the in-app signature verification, the app also conducts a decryption of the account's username and number and outputs it to the log window.



```
Encrypted Stuff:
fA1gW1kXhJEagBoDhFU54gcNj7PJPr1VyPo5gpOxtfE=
::p1maJUAARNEiIVjW0svDpO4CnQgmp8A/0C/
IkltWoqXr0CHDe2Bez7FHL+BcT+sFKPWN+UBLKbde
qdxw3+I163dSygwGDcSilsLIG1j2CpRfumn/
w99GmepygJiXYeja6GhMc/n3MJRS+VeoIJStVKUV
DMRaEyzHi5PPTGAVBb4=

Size of Stuff: 19
Size of Encrypted Stuff: 222
Decrypted Values: Trankalinos 7147025
```

Account Number
7147025

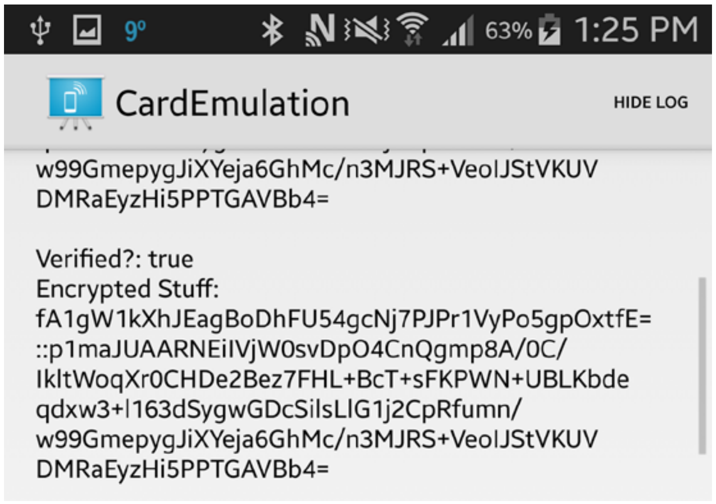
Account Username
Trankalinos

Figure 32.1, 31.2 - Decrypted Values & TextBoxes

It is imperative that the decrypted values (on the left) equal the username and account number as depicted on the right screenshot.

Test Case #22

Recall from Test Case #19 that the data was to be encrypted. In the screenshot below, the reasoning behind the double colon ("::") is to enable the server in the upcoming test cases to split the string by using a delimiter. For the purpose of our proof-of-concept, our delimiter is the double colon. The screenshot below shows the app concatenating the encrypted values of our data with our designated delimiter as well as the signature digest.

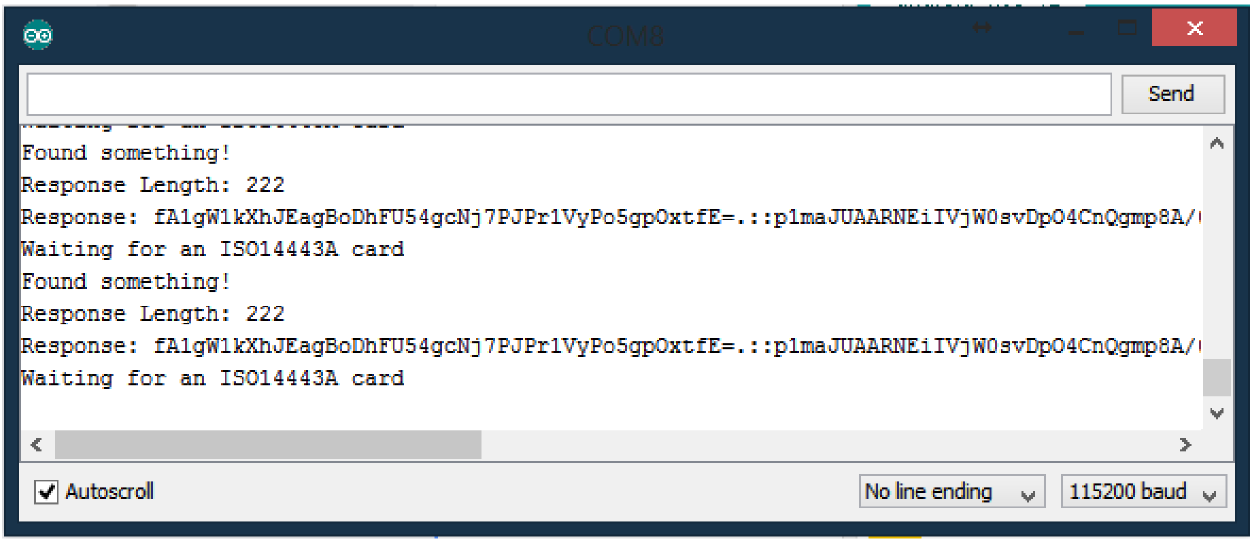


```
w99GmepygJiXYeja6GhMc/n3MJRS+VeoIJStVKUV
DMRaEyzHi5PPTGAVBb4=

Verified?: true
Encrypted Stuff:
fA1gW1kXhJEagBoDhFU54gcNj7PJPr1VyPo5gpOxtfE=
::p1maJUAARNEiIVjW0svDpO4CnQgmp8A/0C/
IkltWoqXr0CHDe2Bez7FHL+BcT+sFKPWN+UBLKbde
qdxw3+I163dSygwGDcSilsLIG1j2CpRfumn/
w99GmepygJiXYeja6GhMc/n3MJRS+VeoIJStVKUV
DMRaEyzHi5PPTGAVBb4=
```

Figure 33 - Double Colon Delimiter

Test Case #23

After all of the previous test cases, the app is ready to transmit the data (or "stuff") to the NFC reader. In the screenshot below, the Arduino serial monitor shows what it was able to receive from the app and smartphone:



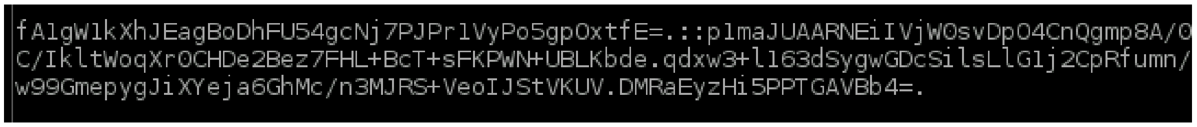
```
Found something!
Response Length: 222
Response: fA1gW1kXhJEagBoDhFU54gcNj7PJPr1VyPo5gpOxtfE=.::p1maJUAARNEiIVjW0svDpO4CnQgmp8A/
Waiting for an ISO14443A card
Found something!
Response Length: 222
Response: fA1gW1kXhJEagBoDhFU54gcNj7PJPr1VyPo5gpOxtfE=.::p1maJUAARNEiIVjW0svDpO4CnQgmp8A/
Waiting for an ISO14443A card
```

Figure 34 - Arduino Serial Monitor

It is important to see that the Response Length is 222, which is consistent with our app when it displayed how large the data being sent over was. We also see our double colon delimiter, and similar encrypted values and signature.

Test Case #24

Once the Arduino is able to retrieve the data from its NFC reader, the serial data is fed into a Python script that is actively listening on its USB serial port. In doing so, the following screenshot depicts the data that it is able to receive:



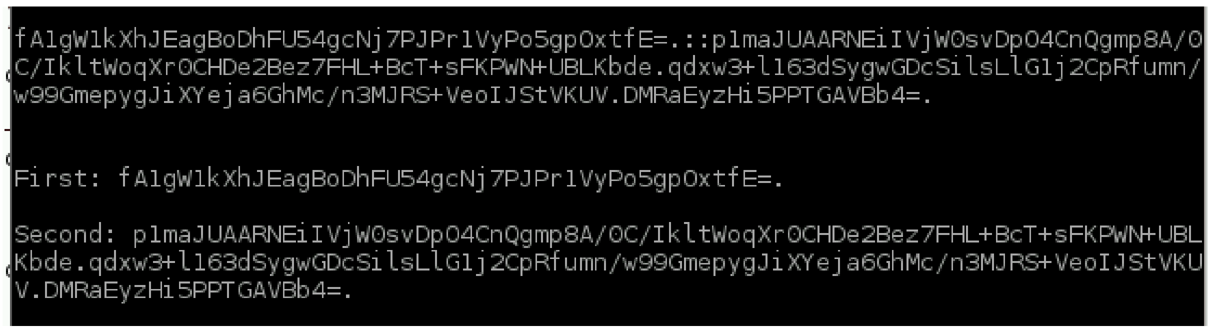
```
fA1gW1kXhJEagBoDhFU54gcNj7PJPr1VyPo5gpOxtfE=.::p1maJUAARNEiIVjW0svDpO4CnQgmp8A/
C/IkltWoqXr0CHDe2Bez7FHL+BcT+sFKPWN+UBLKbde.qdxw3+l163dSygwGDcSilsLlG1j2CpRfumn/
w99GmepygJiXYeja6GhMc/n3MJRS+VeoIJStVKUV.DMRaEyzHi5PPTGAVBb4=.
```

Figure 35 - Python Serial Dump

We can tell that this is the expected values when we compare the first few characters with Test Case #23.

Test Case #25

The next component is to utilize our delimiter to split the signature and the ciphertext. The next set of outputs from our Python script is as follows:



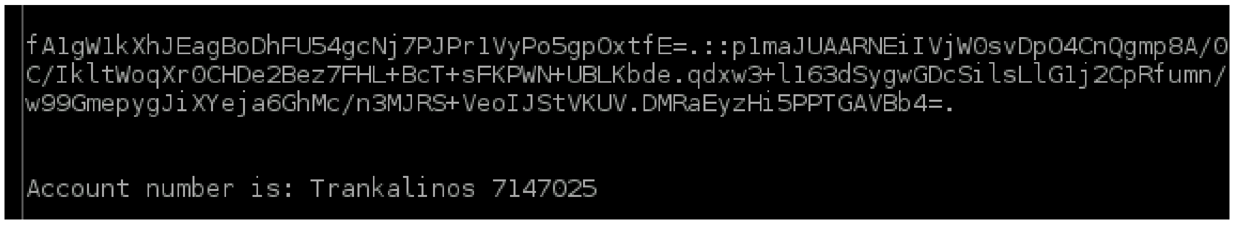
```
fA1gW1kXhJEagBoDhFU54gcNj7PJPr1VyPo5gpOxtfE=.::p1maJUAARNEiIVjW0svDpO4CnQgmp8A/0
C/IkltWoqXr0CHDe2Bez7FHL+BcT+sFKPWN+UBLKbde.qdxw3+l163dSygwGDcSilsLlG1j2CpRfumn/
w99GmepygJiXYeja6GhMc/n3MJRS+VeoIJStVKUV.DMRaEyzHi5PPTGAVBb4=.

First: fA1gW1kXhJEagBoDhFU54gcNj7PJPr1VyPo5gpOxtfE=.

Second: p1maJUAARNEiIVjW0svDpO4CnQgmp8A/0C/IkltWoqXr0CHDe2Bez7FHL+BcT+sFKPWN+UBL
Kbde.qdxw3+l163dSygwGDcSilsLlG1j2CpRfumn/w99GmepygJiXYeja6GhMc/n3MJRS+VeoIJStVKU
V.DMRaEyzHi5PPTGAVBb4=.
```

Figure 36 - Serial Data Split

At this point, we can distinctly see the two components of our data: (first), our ciphertext; and (second), our signature digest.

Test Case #26

The Python scripts requires the ciphertext to be decrypted in order to verify the account. It conducts that now and the result is found in the following screenshot:



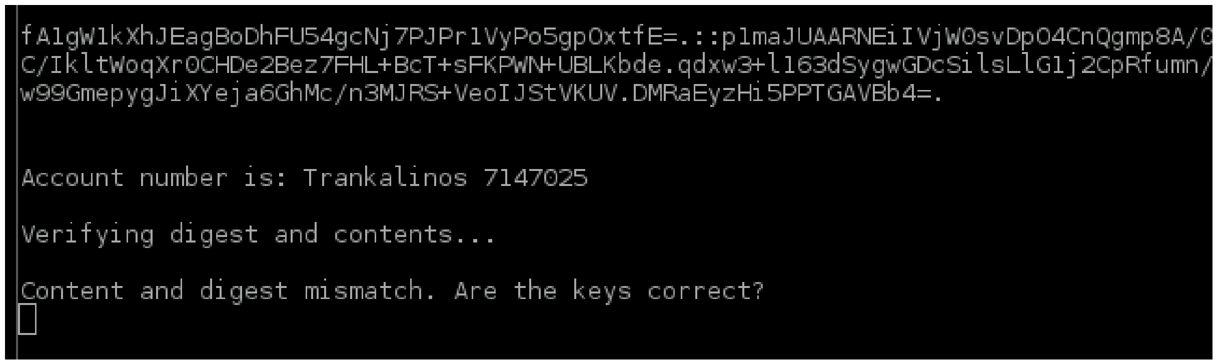
```
fA1gW1kXhJEagBoDhFU54gcNj7PJPr1VyPo5gpOxtfE=.::p1maJUAARNEiIVjW0svDpO4CnQgmp8A/0
C/IkltWoqXr0CHDe2Bez7FHL+BcT+sFKPWN+UBLKbde.qdxw3+l163dSygwGDcSilsLlG1j2CpRfumn/
w99GmepygJiXYeja6GhMc/n3MJRS+VeoIJStVKUV.DMRaEyzHi5PPTGAVBb4=.


Account number is: Trankalinos 7147025
```

Figure 37 - Python Decryption

As expected from our decrypted values in Test Case #21, the resulting plaintext values are human-readable and are consistent with our app.

Test Case #27

The final step is to ensure that there no instances of corruption, mismatches or unexpected values. To do this, the script’s next action is to verify the plaintext values with its signature digest. The screenshot below:



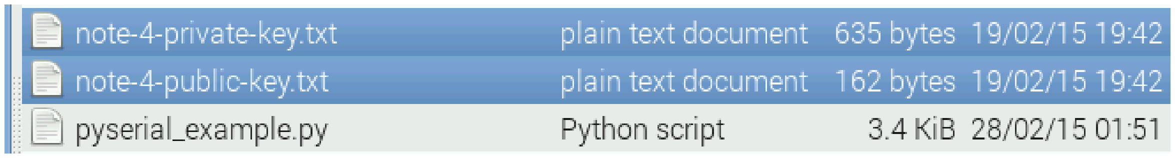
```
fA1gW1kXhJEagBoDhFU54gcNj7PJPr1VyPo5gpOxtfE=.::p1maJUAARNEiIVjW0svDpO4CnQgmp8A/0
C/IkltWoqXr0CHDe2Bez7FHL+BcT+sFKPWN+UBLKbde.qdxw3+l163dSygwGDcSilsLlG1j2CpRfumn/
w99GmepygJiXYeja6GhMc/n3MJRS+VeoIJStVKUV.DMRaEyzHi5PPTGAVBb4=.


Account number is: Trankalinos 7147025

Verifying digest and contents...

Content and digest mismatch. Are the keys correct?
```

Figure 38 - Content & Digest Mismatch

The “Content and digest mismatch” error is expected, because we did not update the Python script with our newly generated private key in Test Case #16 and Test Case #17. A simple prompt, “Are they keys correct?” is an easy reminder for the Administrator to re-register the keys with the server. Consult the next test case for further details.

Test Case #28

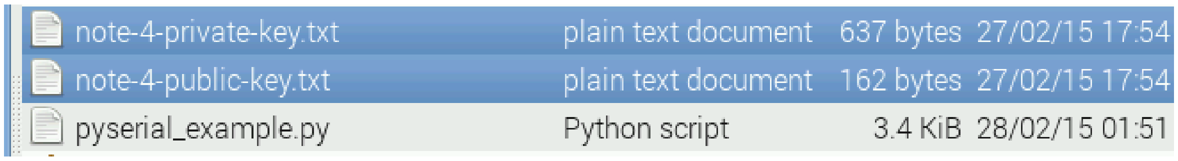
In this test case, we are required to manually update the keys for the Python script with the new ones generated in Test Case #16. To do so manually, and for the sake of testing, we would transfer both the public key and private key over into the Python script’s directory. On the following screenshot below, we see the old ones highlighted:



```
note-4-private-key.txt    plain text document   635 bytes  19/02/15 19:42
note-4-public-key.txt     plain text document   162 bytes  19/02/15 19:42
pyserial_example.py       Python script           3.4 KiB  28/02/15 01:51
```

Figure 39 - Old Client Key Pair

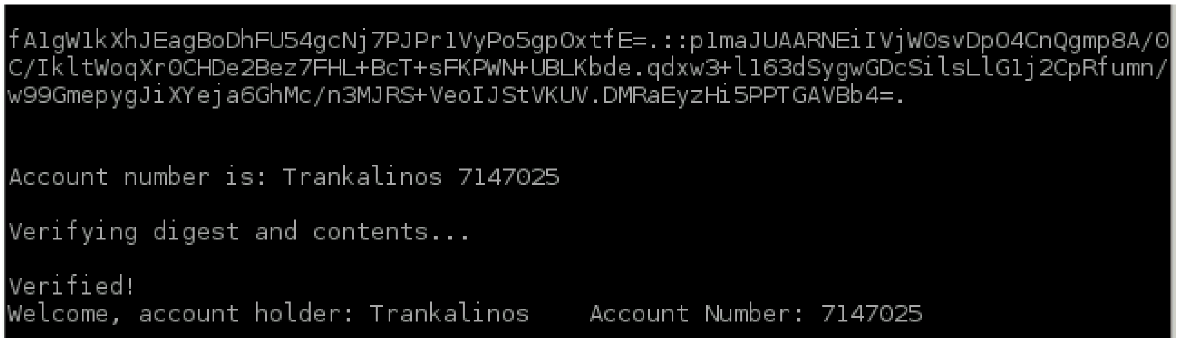
Note the date on the two files. Consult the following screenshot after replacing them with the new key pair files:



```
note-4-private-key.txt    plain text document   637 bytes  27/02/15 17:54
note-4-public-key.txt     plain text document   162 bytes  27/02/15 17:54
pyserial_example.py       Python script           3.4 KiB  28/02/15 01:51
```

Figure 40 - New Client Key Pair

Again, note the date. Once these files were replaced, the following screenshot depicts the verification results after the smartphone was scanned again on the NFC reader:



```
fA1gW1kXhJEagBoDhFU54gcNj7PJPr1VyPo5gpOxtfE=.::p1maJUAARNEiIVjW0svDpO4CnQgmp8A/0
C/IkltWoqXrOCHDe2Bez7FHL+BcT+sFKPWN+UBLKbde.qdxw3+l163dSygwGDcSilsLlG1j2CpRfumn/
w99GmepygJiXYeja6GhMc/n3MJRS+VeoIJStVKUV.DMRaEyzHi5PPTGAVBb4=.


Account number is: Trankalinos 7147025

Verifying digest and contents...

Verified!
Welcome, account holder: Trankalinos    Account Number: 7147025
```

Figure 41 - Account Verified

As expected, once we updated the public key (and private key) generated in Test Case #16 and Test Case #17, the verification was successful.

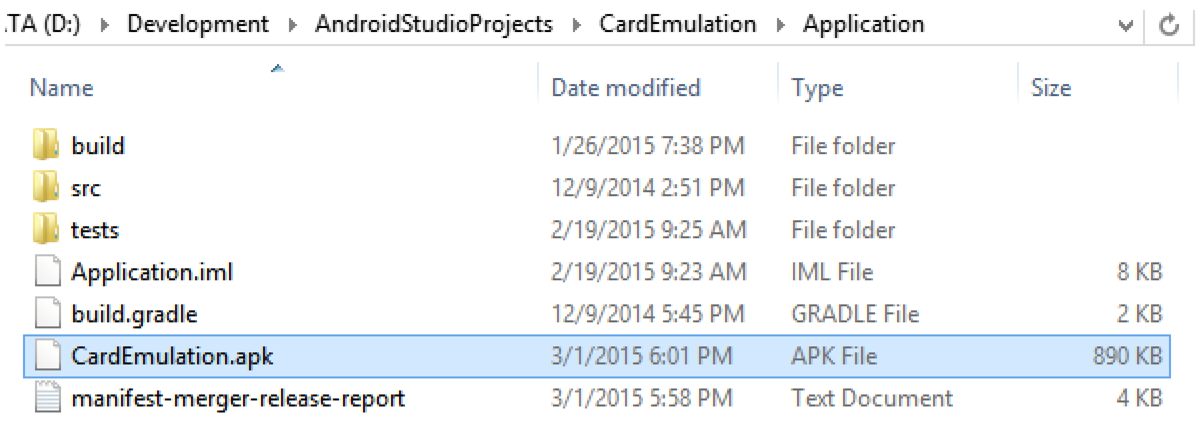
Penetration Testing

In order for the attacker to create and masquerade as a legitimate NFC reader for our app, the attacker must acquire the APDU header of the NFC communication transaction between the NFC reader and Android app, as well as the AID of the app itself. Although the two values are in the source code, they are not directly visible to the average user.

By using some Java tools for penetration testing, we can easily retrieve the raw code from the app assuming that the attacker is able to retrieve the application's .apk file. These tools are:

	Tool Name	Description
1	Baksmali.jar	A .jar file that decompiles the Android .apk by utilizing its .dex file, and allows the user to see and modify the raw assembly language code
2	Smali.jar	A .jar file that recompiles the Android .apk by utilizing a modified .dex file

For the sake of attempting to penetrate this project, we shall only use baksmali.jar as it is unnecessary to modify any of the source code. To perform this penetration test, we had to generate our own .apk for our CardEmulation app. Android Studio does this for us with relative ease. Here is the file directory where our .apk was generated:

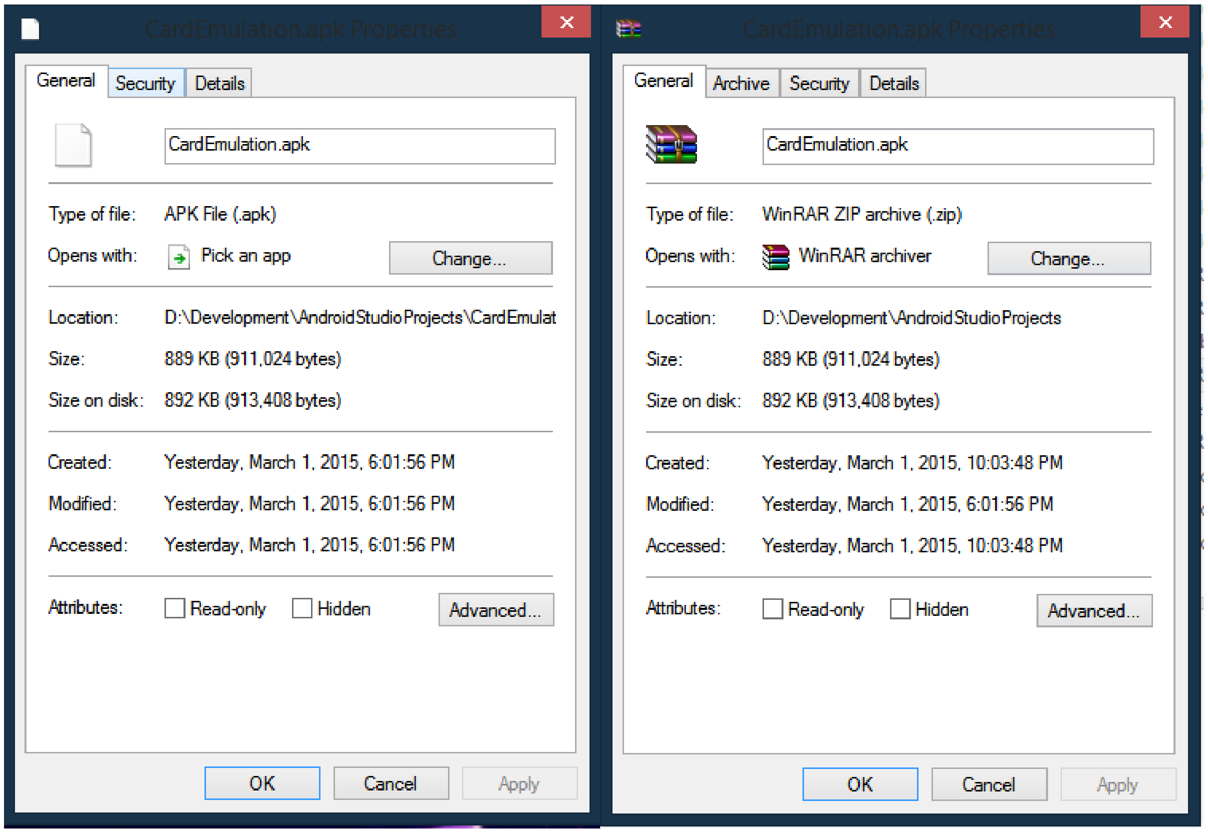


.TA (D:) ▸ Development ▸ AndroidStudioProjects ▸ CardEmulation ▸ Application

Name	Date modified	Type	Size
build	1/26/2015 7:38 PM	File folder	
src	12/9/2014 2:51 PM	File folder	
tests	2/19/2015 9:25 AM	File folder	
Application.iml	2/19/2015 9:23 AM	IML File	8 KB
build.gradle	12/9/2014 5:45 PM	GRADLE File	2 KB
CardEmulation.apk	3/1/2015 6:01 PM	APK File	890 KB
manifest-merger-release-report	3/1/2015 5:58 PM	Text Document	4 KB

Figure 42 - APK file generated

For safety reasons, a copy of the CardEmulation.apk was made in a different directory where our java tools are located. Once copied, we renamed the .apk by appending another file extension, .zip. On the left is the original .apk file while the right depicts the file with the .zip extension:



CardEmulation.apk Properties

General | Security | Details

CardEmulation.apk

Type of file: APK File (.apk)
Opens with: Pick an app
Location: D:\Development\AndroidStudioProjects\CardEmulat
Size: 889 KB (911,024 bytes)
Size on disk: 892 KB (913,408 bytes)
Created: Yesterday, March 1, 2015, 6:01:56 PM
Modified: Yesterday, March 1, 2015, 6:01:56 PM
Accessed: Yesterday, March 1, 2015, 6:01:56 PM
Attributes: Read-only Hidden

CardEmulation.apk Properties

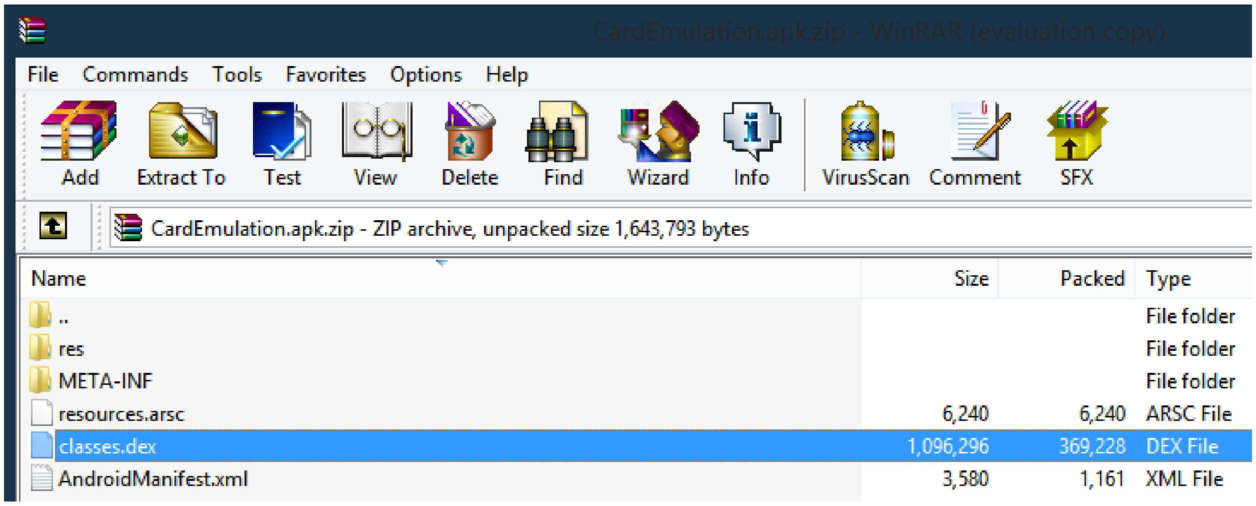
General | Archive | Security | Details

CardEmulation.apk

Type of file: WinRAR ZIP archive (.zip)
Opens with: WinRAR archiver
Location: D:\Development\AndroidStudioProjects
Size: 889 KB (911,024 bytes)
Size on disk: 892 KB (913,408 bytes)
Created: Yesterday, March 1, 2015, 10:03:48 PM
Modified: Yesterday, March 1, 2015, 6:01:56 PM
Accessed: Yesterday, March 1, 2015, 10:03:48 PM
Attributes: Read-only Hidden

Figure 43.1, 42.2 - APK and APK.zip

This “hack” allows the user to traverse inside the .apk file. Below is a screenshot that highlights the file that we are interested in, in order to decompile the application:



CardEmulation.apk.zip - WinRAR (evaluation copy)

CardEmulation.apk.zip - ZIP archive, unpacked size 1,643,793 bytes

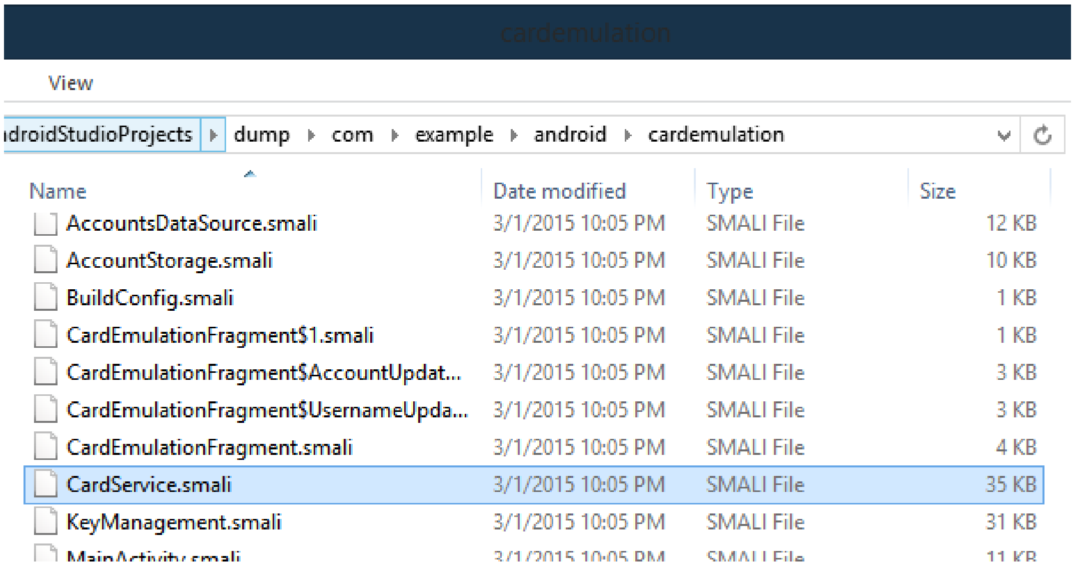
Name	Size	Packed	Type
..			File folder
res			File folder
META-INF			File folder
resources.arsc	6,240	6,240	ARSC File
classes.dex	1,096,296	369,228	DEX File
AndroidManifest.xml	3,580	1,161	XML File

Figure 44 - Classes.dex from .zip file

The classes.dex file was "extracted" from this pseudo-zip file and stored on the same directory as our decompiler. The next step is to open a command prompt, change directories to the working directory where the classes.dex file and the baksmali.jar could be found, and enter the following command:

```
java -jar baksmali-2.0.5.jar -o ./dump ./classes.dex
```

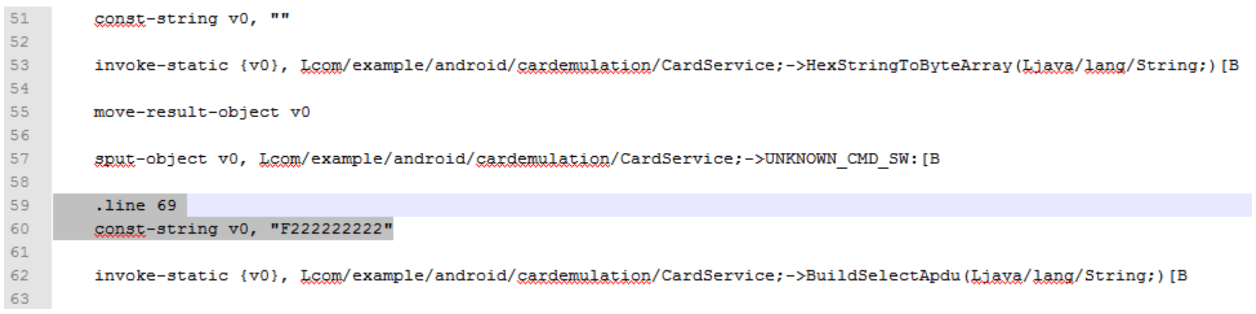
Where, "./dump" is the directory where the decompiled files shall be stored, and "./classes.dex" is the classes.dex from our pseudo-zip file that was extracted. Inside the *dump* directory is where the source code is. After traversing through com, example, android, and cardemulation, we have successfully located the raw assembly code from our application as seen below:


cardemulation
View
idroidStudioProjects › dump › com › example › android › cardemulation

Name	Date modified	Type	Size
AccountsDataSource.smali	3/1/2015 10:05 PM	SMALI File	12 KB
AccountStorage.smali	3/1/2015 10:05 PM	SMALI File	10 KB
BuildConfig.smali	3/1/2015 10:05 PM	SMALI File	1 KB
CardEmulationFragment$1.smali	3/1/2015 10:05 PM	SMALI File	1 KB
CardEmulationFragment$AccountUpdat...	3/1/2015 10:05 PM	SMALI File	3 KB
CardEmulationFragment$UsernameUpda...	3/1/2015 10:05 PM	SMALI File	3 KB
CardEmulationFragment.smali	3/1/2015 10:05 PM	SMALI File	4 KB
CardService.smali	3/1/2015 10:05 PM	SMALI File	35 KB
KeyManagement.smali	3/1/2015 10:05 PM	SMALI File	31 KB
MainActivity.smali	3/1/2015 10:05 PM	SMALI File	11 KB

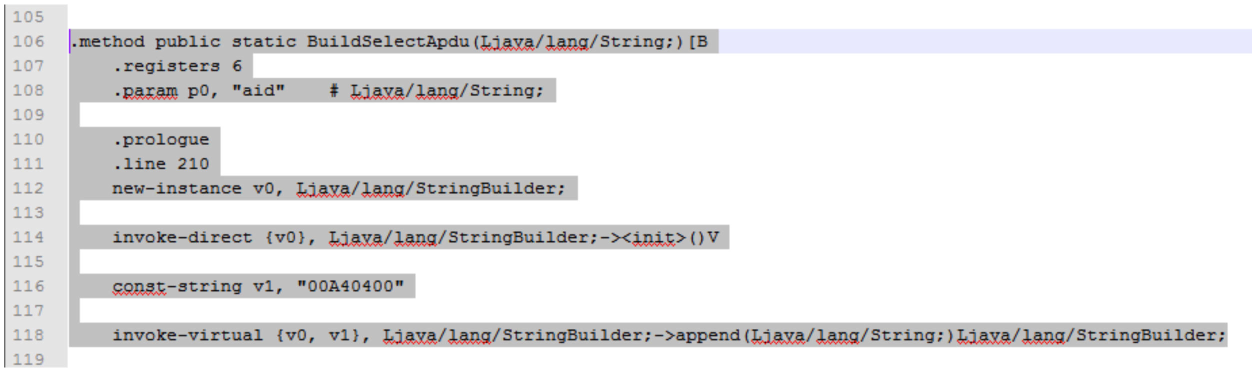
Figure 45 - .smali dumps

We are interested in the CardService.smali file, because that is where we store the APDU header and AID as static strings. These .smali files can be opened with Notepad++ as seen below:



```
    const-string v0, ""

    invoke-static {v0}, Lcom/example/android/cardemulation/CardService;->HexStringToByteArray(Ljava/lang/String;)[B

    move-result-object v0

    sput-object v0, Lcom/example/android/cardemulation/CardService;->UNKNOWN_CMD_SW:[B

    .line 69
    const-string v0, "F222222222"

    invoke-static {v0}, Lcom/example/android/cardemulation/CardService;->BuildSelectApdu(Ljava/lang/String;)[B
```

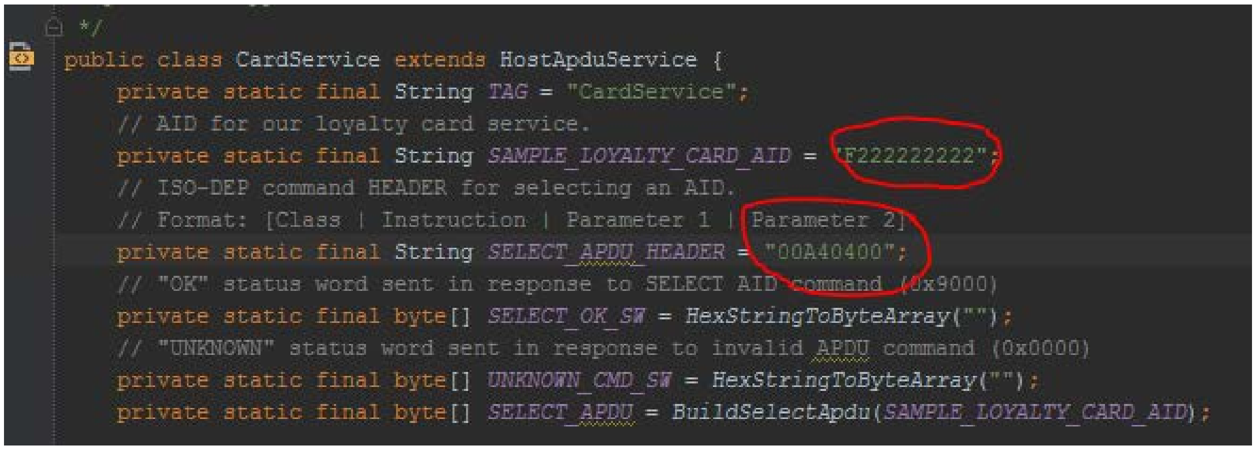
Figure 46 - CardService.smali (AID selector)



```
.method public static BuildSelectApdu(Ljava/lang/String;)[B
    .registers 6
    .param p0, "aid"    # Ljava/lang/String;

    .prologue
    .line 210
    new-instance v0, Ljava/lang/StringBuilder;

    invoke-direct {v0}, Ljava/lang/StringBuilder;-><init>()V

    const-string v1, "00A40400"

    invoke-virtual {v0, v1}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

Figure 47 - CardService.smali (APDU header)

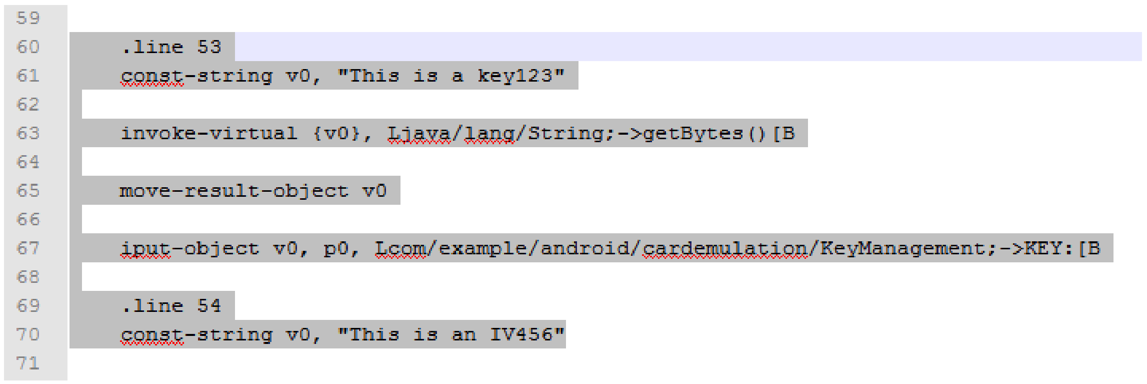
Note the highlighted, and compare to the actual Java code of our application:



```
 */
public class CardService extends HostApduService {
    private static final String TAG = "CardService";
    // AID for our loyalty card service.
    private static final String SAMPLE_LOYALTY_CARD_AID = "F222222222";
    // ISO-DEP command HEADER for selecting an AID.
    // Format: [Class | Instruction | Parameter 1 | Parameter 2]
    private static final String SELECT_APDU_HEADER = "00A40400";
    // "OK" status word sent in response to SELECT AID command (0x9000)
    private static final byte[] SELECT_OK_SW = HexStringToByteArray("");
    // "UNKNOWN" status word sent in response to invalid APDU command (0x0000)
    private static final byte[] UNKNOWN_CMD_SW = HexStringToByteArray("");
    private static final byte[] SELECT_APDU = BuildSelectApdu(SAMPLE_LOYALTY_CARD_AID);
```

Figure 48 - Android Studio Source Code I

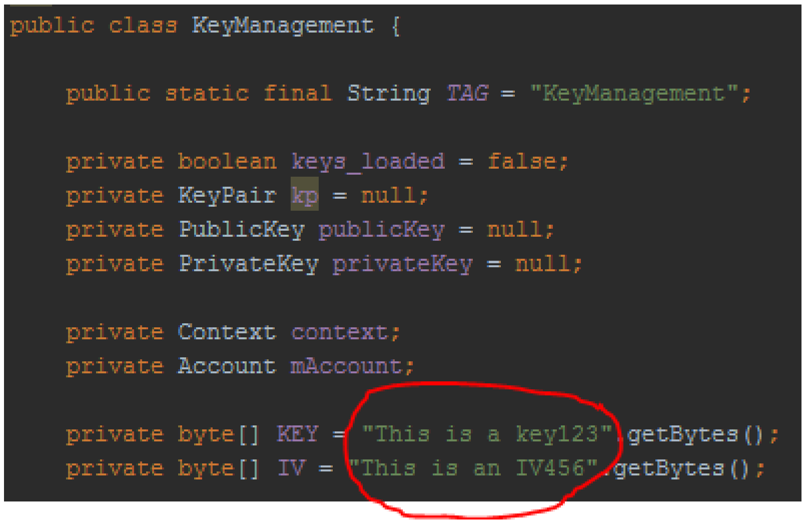
By simply using baksmali.jar, the attacker is able to extract the two of four key pieces of information it needs to perform a relay attack. In addition, consult the following two screenshots; the first is a screen capture of the KeyManagement.smali file and the second is the Android source code of the KeyManagement.class:



```
    .line 53
    const-string v0, "This is a key123"

    invoke-virtual {v0}, Ljava/lang/String;->getBytes()[B

    move-result-object v0

    iput-object v0, p0, Lcom/example/android/cardemulation/KeyManagement;->KEY:[B

    .line 54
    const-string v0, "This is an IV456"
```

Figure 49 - KeyManagement.smali (static key & iv)

As you can see, the code displays what we hardcoded into the source.



```
public class KeyManagement {

    public static final String TAG = "KeyManagement";

    private boolean keys_loaded = false;
    private KeyPair kp = null;
    private PublicKey publicKey = null;
    private PrivateKey privateKey = null;

    private Context context;
    private Account mAccount;

    private byte[] KEY = "This is a key123".getBytes();
    private byte[] IV = "This is an IV456".getBytes();
```

Figure 50 - Android Studio Source Code II

It is important to know that the attacker could easily use this method to extract the static secret key and initialization vectors to complete the four components for a successful and uncontested relay attack, as seen in the previous screenshots. Therefore, it is imperative to implement some form of randomization in the future for the initialization vector, as well as encrypting the secret key with our private key, and storing it in our SQLite database only to be retrieved when needed.

Implications of the Implementation

Discussion: Vulnerability to Relay Attacks

Although the NFC protocol is still new when it comes to host-card emulation, and even though this project attempts to add more security in the transaction, the data exchange is still vulnerable to a relay attack. Consider the following scenario with the following assumptions:

	Assumptions
1	The victim is completely unaware of the attacker
2	The victim's smartphone has NFC enabled when his or her screen is off through open source software
3	The victim has a location where his or her smartphone can be used for access

	Condition	Likelihood	Impact
1	The attacker is able to determine the Android ID (AID)	Medium	High
2	The attacker is able to determine the APDU request	Medium	High
3	The attacker is close enough to a vulnerable smartphone to intercept an encrypted message	Low	High
4	The attacker knows where the encrypted message could be forwarded for access	Medium	High

In this scenario, the attacker has spent sufficient amount of time determining the AID of our app, as well as the unique APDU header in order to initiate the data exchange (see **Penetration Testing** section). The attacker is then able to stalk the person, at a proximity of nearly touching them and then waves his NFC reader to retrieve the encrypted message in our app. In this scenario, we also assume that the victim's smartphone has his or her screen *on.* When the smartphone's screen is off, the likelihood of a successful interception is extremely low.

At this point, we can see that there are only a limited amount of solutions that makes every single transaction unique. And even though the software is still susceptible to a relay attack, the chances of such an attack happening is quite low, and requires the user to be nearly incompetent in order to be a victim of such an attack. In the rare case where there is a victim of such an attack, the probability of the victim being a victim of another targeted, but unrelated attack is also quite high.

Discussion: Security Methodology & Implementation

The initial approach to this project was to utilize the public and private keys of both the client and server for the data exchange over NFC. In essence, the following set of criteria “builds” on the encryption implementation between the two:

Security Feature	Security Implementation
Server-Only Asymmetric Encryption	The smartphone only utilizes the server’s public key to encrypt
Server & Client Asymmetric Encryption	The smartphone uses its private key to perform a first level encryption, and then the server’s public key to perform a secondary level encryption
Signature Verification	The smartphone sends its contents in plaintext, appended with a signature digest of the contents, where the server performs a signature verification
Signature Verification & Server-Only Asymmetric Encryption	The smartphone encrypts its contents with the server’s public key, and appends its ciphertext with a signature digest, where the server decrypts the contents with its private key and performs a signature verification

However, there were difficulties with loading the server’s public key on the client application. According to some research, the Android application requires the server’s public key to be in a .der format. Unfortunately, after exporting the public key to the client in said format, the Android application threw an unexpected error: *InvalidKeySpecException*.

For purposes of testing, we decided to copy the public key into the Android application as a string, and removed the RSA headers and footers. When the application was compiled, the loading of the public key did not produce the previously aforementioned error. Instead, at the point where the app was to encrypt the contents, it produced a new error: *ArrayIndexOutOfBoundsException: too much data for RSA block*.

A suggestion was proposed to use a hybrid encryption method after the previous error occurred. This method utilized asymmetric encryption to securely transfer the symmetric encryption system’s secret key and in this proof-of-concept, its initialization vector. In turn, the following set of criteria is then modified from its initial intended implementation:

Security Feature	Security Implementation
Client-Side AES Encryption	The smartphone only utilizes symmetric encryption, knowing an agreed-upon secret key and initialization vector
Signature Verification	The smartphone sends its contents in plaintext, appended with a signature digest of the contents, where the server performs a signature verification
Client-Side AES Encryption & Signature Verification	The smartphone encrypts its contents, and the ciphertext is appended with a signature digest of the contents, where the server decrypts the ciphertext and performs a signature verification

This method of implementation needs to be improved by adding a randomized 16-byte initialization vector that is prepended to the front of the application. The server is required to take the first 16-bytes of the response to decrypt the contents. Regardless, it appears that NFC readers and host-card emulation applications are easily able to adopt means of encryption in their communication.

Research Topics to Refine the Solution

Because the proposed solution is a proof-of-concept, there are still questions that need to be addressed before the solution could be released as an off-the-shelf product or solution. In turn, the following questions are topics that need to be sought into further detail:

What are potential solutions for other smartphones such as iOS, Blackberry and Windows OS?

A key drawback to this project is that it is specifically made for Android smartphones. While Android-based smartphones dominate the majority of the smartphone market, it is unwise to delimit our product to just Android smartphones. Further research should be targeted towards other smartphones that have a large portion of the smartphone market. At this point, while Apple iOS-based smartphones recently received NFC support, but the company has severely inhibited its use by restricting it to just Apple Pay (CNET, 2014). It may be best to then develop a non-NFC solution for Apple smartphones until their NFC capabilities are unrestricted.

Are there any beneficial differences for this proposed system once Windows 10 is made available for the Raspberry Pi?

As of February 2015, the Raspberry Pi Foundation released the Raspberry Pi (Model) 2 and announced that they are working closely with Microsoft to produce a Windows 10-based operating system for the new Raspberry Pi. (Microsoft, 2015) In turn, instead of continuing to produce a *nix-based solution for this implementation, what kind of benefits would a Windows-based operating system provide to the system? Initial impressions of a Windows-based solution includes taking advantage of Active Directories, an external Microsoft SQL Server, integration with Exchange server, etc. The potential synergies and high cohesion should be a strong rationale to research into an alternative solution to the proposed.

Which sizes and modes of AES encryption are the most efficient?

The project development stages of this project encountered different modes of encryption with respect to AES symmetric encryption methods. Although the project elected to utilize AES-128 with cipher-block chaining (CBC) and PKCS5 padding, there were other candidates that were as qualified to be implemented as well. These included AES-192 and AES-256, which required a key size of 24- and 32-bytes respectively as opposed to our 16-byte key with AES-128. Modes for encryption included cipher feedback (CFB), output feedback (OFB), counter (CTR) and OpenPGP.

There was no rationale behind choosing AES-128 with CBC and PKCS5 padding. However, it is important to identify if there is a significant difference in security and is there a trade-off when implementing other modes of symmetric encryption. For example, by eliminating padding in encryption schemes, we can reduce the overhead of the data to be transferred. Questions and peculiarities like so should be addressed in future research iterations of this project.

What are optimal use-cases for the administrator to register new smartphones?

In this project, we had not envisioned a use-case where we would perform the registration of the smartphone. To achieve this “registration” component in our testing and proof-of-concept, external public and private keys were exchanged over FTP clients. Looking forward, an exchange over FTP may work but it still requires manual user input. An optimal solution would require the administrator to use minimal effort for the transfer of keys and a technology that can take advantage of this would be Peer-to-Peer NFC.

While we have criticized the vulnerabilities of Peer-to-Peer NFC in the previous sections, it serves the purpose here because the registration use-case is conducted under a controlled environment. It also utilizes the smartphone’s NFC capabilities and already has a large API for both client and server. Registration optimization, however, should require to external input and research for “ergonomics” and ease of use.

Future Scope Enhancements

Due to financial restrictions and time constraints imposed on this project, the following components shall not be addressed. However, for expansion and implementation for future prospective researchers and students alike, these components may be considered innovative and can be included in the future.

Centralized Server Integration & Access Control Groups

While we have an embedded system that is a standalone server that works alongside our NFC reader, we can “lighten the load” of the embedded system and have it forward the information to a centralized server. This centralized server will monitor locations for authorized access for example.

Once a foundation for a home network is established using a centralized server and multiple users, we can then begin to implement access control groups to allow only certain users access to certain places.

Implement Access Control Logic for Door Locks

The Arduino is flexible for prototyping electronics. While this project has not discussed or attempted to develop logic for opening and closing door locks due to financial restrictions and unfamiliarity with hardware engineering, the rationale is enough for future implementations to include this within the Arduino source code, or when the actual product is developed.

Include Email & SMS Notification

The inclusion of email and SMS notification can allow users, homeowners especially, to know whether there are suspicious activity occurring around areas where NFC authentication is in place. This is due to the fact that, once a centralized server is configured, networking notifications becomes a simple matter of connecting all NFC readers using our embedded systems together in a switch or wireless network.

However, it is unrealistic to configure a mail server for each embedded system; instead, it will be much easier to have one centralized mail server that resides on the centralized server that we have already mentioned. An SMS server can only be attained by enlisting the help of a designated telecommunications service provider.

Standardizing an HCE Protocol

Create a document based on IEEE standards that depicts the requirements and constraints of communication between host-card emulated apps and NFC readers. This way, the protocol will be easily implemented across a multitude of platforms once developers and engineers can agree on a protocol.

The protocol itself should be similar to the TCP 3-way handshake. However, the protocol should adopt a throw-catch protocol: the NFC reader should send a request (the “throw”) to the client, in which the client retrieves the request (the “catch”) and sends back the same “packet”, only this time it is populated with data (the “throw”). The NFC reader then receives the response (the “catch” again) and parses the data as necessary.

The standardizing of this protocol should also be placed in a Request for Comments where other like-minded developers and engineers can provide more valuable insight, better design work, and stronger mutual agreement on the standard.

Conclusion

The proof-of-concept experimentation between Android smartphones and NFC readers proved to be feasible. The key ingredient was to find the hardware that was compatible reading from the smartphone’s NFC chip as well as having a compatible NFC library built. After some trial and error with the hardware component, the development of the Android app and the server proved to be reasonable.

Ultimately, the key difficulties after solving the hardware problem was to find a common ground between the software. Extensive research and understanding of the host-card emulation protocol provided and developed by Google were helpful in establishing the first means of communication between client and reader. When the NFC reader was able to extract information from the smartphone, it was simply a matter of testing and trying which encryption approaches worked.

There were multiple choices on how to implement our encryption modes. For the sake of experimentation, we were only interested if encryption was even possible for NFC. If it was possible, why hasn’t encryption become a staple standard for host-card emulation communication? In the beginning, our approach was to only utilize RSA encryption between client and server. However, after running into technical difficulties, a hybrid encryption method was introduced. While there were some road bumps during the experimental component of our project, after implementing AES-128 encryption utilizing CBC mode and PKCS5 padding, we were finally able to decrypt the message on the server side.

Not all software become bullet-proof to attacks. No matter how many levels of encryption and security is implemented into our proposed system, there will always be a way to break the system. However, the discussion for attacks on our system have illustrated that while the system can be attacked, the likelihood of these attacks being successful is extremely low. We have also said that, because of how our smartphones are designed with security in mind, a potential relay attack can only occur if the victim allows the attacker to be within range, masquerading their NFC reader as a legitimate reader for the app. Even then, we have identified that the victim’s smartphone must have their screen active in order to intercept the data exchange. While the relay attack is simple in theory, the actual attack being conducted has a minimal chance of being successful.

And like all proof-of-concepts, there is always room for improvement. We identified areas for future scope improvements and implementation to make our solution more robust. With the announcement of the Raspberry Pi 2 and its collaboration with Microsoft for Windows 10, there is a whole other area to research and experiment with. Also, even though our proof-of-concept speaks about access control for doors and locks, we have not yet implemented that component. It is imperative to keep in mind that there are many uses for this project, and considering how NFC and host-card emulation are still in their infant stages, this is a prime time to jump onto their bandwagon and develop revolutionary and innovative technologies.

Lastly, as a culmination of my four years at the British Columbia Institute of Technology, I would like to express my gratitude to all of my professors. I hope that this final project is enough to satisfy your requirements. I believe that I have done the best of my ability to demonstrate everything that I have accumulated in my diploma program and bachelor's degree. And finally, I hope that the suggestions that I have discussed today can be used as a stepping stone for future budding students that are enthusiastic with learning the NFC protocol, developing Android applications, and creating servers that does their bidding.

References

Lady Ada. (2012, December 30). Adafruit PN532 RFID/NFC Breakout and Shield. Retrieved March 1, 2015, from https://learn.adafruit.com/adafruit-pn532-rfid-nfc

Microsoft. (2015, February 1). Windows 10 for Raspberry Pi 2. Retrieved March 1, 2015, from https://dev.windows.com/en-us/featured/raspberrypi2support

Reilly, C. (2014, September 16). Apple Locks iPhone6 NFC to Apple Pay. Retrieved March 1, 2015, from http://www.cnet.com/news/apple-locks-down-iphone-6-nfc-to-apple-pay/

Wang, R. *ElecFreaks NFC Video Tutorials* [Motion picture]. (2013). YouTube.

Appendix I – Project Timeline

As per the project's guidelines, the minimum amount of hours spent on the project must be at least 405 man-hours. The project will begin on January 5th 2015 with an initial due date by May 1st 2015, and an extension lasting to January 1 2016. Below is a breakdown of the milestones that shall be achieved through this project, and their projected hours:

Milestone 1: Configuring Embedded Systems & NFC Reader

This project milestone includes the setup and familiarization of the Raspberry Pi, its operating system, and some preliminary form of NFC reader configuration. Some form of testing shall be conducted and the test cases and results shall be documented in the final report of the research project. For now, some example test cases can be referenced in the former of this document. After testing the waters with the Raspberry Pi and its NFC reader, we can move on to the next milestone.

Components	Description	Estimated Hours	Actual Hours	Accumulated Hours
Familiarization of Raspberry Pi and "Wheezy" Operating System	Understanding the hardware limitations, how things are implemented, and where things should go. Understanding Debian environment over Redhat / SUSE / Fedora.	20 hours	20 hours	20 hours
Installation and configuration of RFID Reader	Understanding the hardware with relation to software. Basically, a period to play around with the components of the system hardware architecture.	20 hours	10 hours	30 hours
Testing RFID Reader with included RFID tags, cards, tokens, etc.	Testing the hardware components by verifying Raspberry Pi's software components. Results shall be documented as testing progresses.	20 hours	10 hours	40 hours
Installation of NFC Reader: Adafruit NFC Breakout Board - PN532	Understanding the hardware with relation to software. Basically, a period to play around with the components of the system hardware architecture; properly installing them onto Arduberry and breadboard wirings	20 hours	30 hours	70 hours
Installation of NFC Reader: Adafruit NFC Shield - PN532	Understanding the hardware with relation to software. Basically, a period to play around with the components of the system architecture.	10 hours	20 hours	90 hours

Installation of NFC Reader: NXP EXPLORE-NFC Module for Raspberry Pi	Understanding the hardware with relation to software. Basically, a period to play around with the components of the system hardware architecture.	15 hours	20 hours	110 hours
Installation of NFC Reader: Seeed Studio NFC Module V2	Understanding the hardware with relation to software. Basically, a period to play around with the components of the system hardware architecture.	15 hours	20 hours	130 hours
Testing all NFC Readers with MiFare Classic 1K NFC tags, Android smartphone, etc.	Testing the hardware components by verifying Raspberry Pi's software components. Results shall be documented as testing progresses.	10 hours	10 hours	140 hours
Documentation of Project Report	An ongoing component of the research project, whereby each milestone shall be tested according to test cases. Test results shall be documented accordingly and to the best of the researcher's ability.	10 hours	10 hours	150 hours

Milestone 2: Develop Android App to Communicate with NFC Reader

Once we are done experimenting with the embedded systems, we can transition to working with our medium of choice. In this research project, we have delimited our medium to be an Android operating system, and to use either the Google Nexus 5 or the Samsung Galaxy Note 4. Using the Android Development Environment, we shall develop an app that can turn on NFC capabilities of the smartphone and to communicate using NFC standard protocols. Some preliminary test cases shall be applied to the app developed.

Components	Description	Estimated Hours	Actual Hours	Accumulated Hours
Develop and design GUI and application core	Enlist some roommates that are good in design work and ask for ideas on how to make the app intuitive.	20 hours	20 hours	170 hours
Program and code NFC and NFC-compliant components	Some preliminary coding that will allow the smartphone to be recognized by the NFC reader.	40 hours	40 hours	210 hours
Test application and code to some degree	Testing the smartphone is recognized. These "dumps" gauge how difficult the next milestone will be.	20 hours	20 hours	230 hours

Document all preliminary findings	An ongoing component of the research project, whereby each milestone shall be tested according to test cases. Test results shall be documented accordingly and to the best of the researcher's ability.	10 hours	10 hours	240 hours

Milestone 3: Ensure Android App is ISO-14443-3A Compliant

In this component, we refer to our literature review for ideas on improving security between the two components. We shall try to implement as many of the security features as possible between the application and the NFC reader, and then record if the attempts failed or passed. This component of testing and development is exploratory in nature and is used to see if other means of security can be used for authentication.

Components	Description	Estimated Hours	Actual Hours	Accumulated Hours
Implement Public Key Infrastructure	Configuring between the Raspberry Pi and the Smartphone app, to adopt a Public Key Infrastructure.	40 hours	40 hours	280 hours
Implement Digital Signatures	Configuring either the NFC reader or the Smartphone to provide correct and incorrect digital signatures.	40 hours	40 hours	320 hours
Implement Digital Certificates	Configuring either the NFC reader or the Smartphone to provide recognized and unrecognized digital certificates.	40 hours	0 hours	320 hours
Testing each security measure	Each security measure shall be tested against a series of test cases.	30 hours	20 hours	340 hours
Penetration Testing (if possible) with Kali Linux	Using Kali Linux and its suite of penetration testing tools, to test our application.	30 hours	30 hours	370 hours
Documentation of all security findings, lessons learned, and identification of weaknesses	An ongoing component of the research project, whereby each milestone shall be tested according to test cases. Test results shall be documented accordingly.	30 hours	30 hours	400 hours

Milestone 4: Finalize Project Deliverables

Once we reach this milestone in the research project, it becomes a matter of finishing up the final project report and adding any missing components to the report. We shall use any time during this milestone to make adjustments to our project as necessary.

Components	Description	Estimated Hours	Actual Hours	Accumulated Hours
Finish Project Report	Adding details, diagrams, etc. that are required as part of the final project deliverables.	15 hours	20 hours	420 hours
Other miscellaneous components that have not been addressed.	Any component that have not been addressed by the project or have not been encountered by the researcher.	20+ hours	At least 20 hours	440+ hours

Appendix II – Installation & User Manuals

Installing the Android App to Smartphone

On-disk includes the APK file of the CardEmulation app, as well as the source code that can be compiled using the latest Android Studio IDE. Before attempting to install to the Android smartphone, ensure that:

1. Android smartphone drivers are properly installed on the PC
2. USB Debugging is enabled on the smartphone

To install using the IDE, open Android Studio and go to File > Import Project:

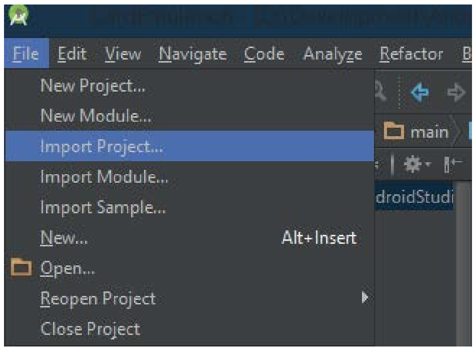
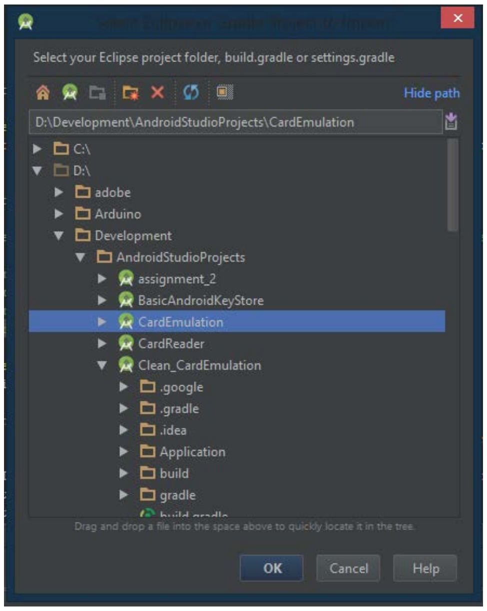


Figure 51 - Import Project in Android Studio

A new window will appear. Navigate to where the project is located and select it to open in Android Studio:



Select your Eclipse project folder, build.gradle or settings.gradle

D:\Development\AndroidStudioProjects\CardEmulation

Figure 52 - Select CardEmulation Android Studio Project

After the project is loaded in the IDE, connect the Android smartphone to the PC and run the program through Run > Run ‘Application’:

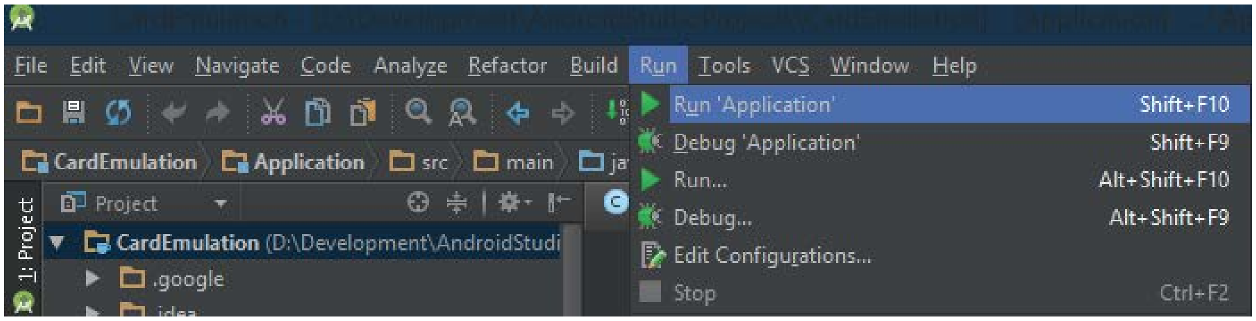
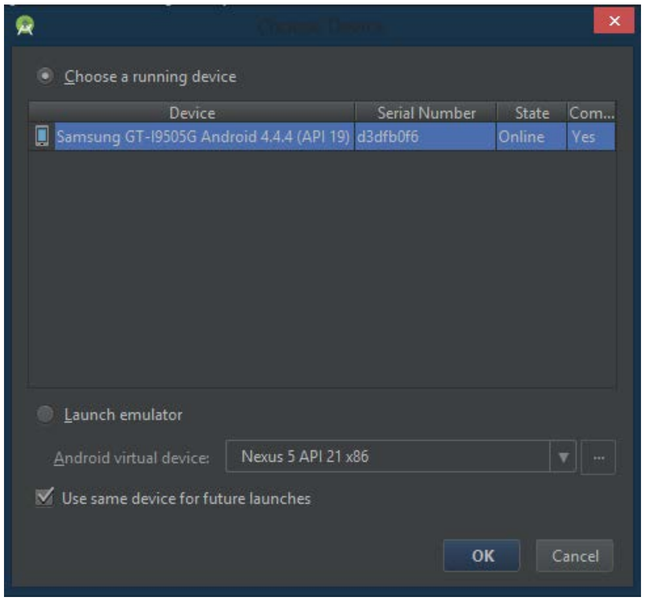


Figure 53 - Android Studio: Run Application

Another window will appear, prompting to select the device to run the application on. It is recommended to use a physical Android smartphone over an emulator as this project does not have a solution for soft NFC readers for the emulator:



Choose a running device

Device	Serial Number	State	Com...
Samsung GT-I9505G Android 4.4.4 (API 19)	d3dfb0f6	Online	Yes

Launch emulator

Android virtual device: Nexus 5 API 21 x86

Use same device for future launches ✓

Figure 54 - Android Studio: Select Device

The application should now launch in the smartphone. To launch the application via the APK file, the CardEmulation.apk file should be placed in an easy-to-access location on the smartphone. Once placed in there through whatever means, use a File Explorer app to locate it and run the application.

Priming the Raspberry Pi

Assuming that the Raspberry Pi's operating system (Debian "Wheezy") has been imaged into its microSD card and some prerequisite configurations were completed, the Raspberry Pi is ready to be updated. Ensure that the Raspberry Pi is connected to an internet source, then boot up the Raspberry Pi by providing a micro USB power source to it with 5 volts and at least 1 amp. When the Raspberry Pi is finished booting up, open a command terminal and enter the following command:

```
sudo apt-get update
```

Allow the Raspberry Pi to update itself. Once finished, install the Arduino IDE in order to install the Arduino USB drivers:

```
sudo apt-get install -y arduino
```

Other miscellaneous tools for development included Geany IDE:

```
sudo apt-get install -y geany
```

Fetching the Arduino Libraries for the NFC Readers

After the Arduino IDE is installed on the Raspberry Pi, navigate to the following using a terminal console:

```
cd /etc/arduino/libraries
```

In this directory, run the following commands to fetch the appropriate libraries for our Seeed Studio NFC Shield:

```
git clone –recursive https://github.com/Seeed-Studio/PN532.git NFC
ln –s NFC/PN532 ./
ln –s NFC/PN532_SPI ./
ln –s NFC/PN532_I2C ./
ln –s NFC/PN532_HSU ./
ln –s NFC/NDEF ./
```

This shall prime the Arduino IDE with all the libraries necessary for the NFC Shield to work.

Configuring Python & Python Libraries for Server

On the Raspberry Pi, install the following dependencies for the Python scripts provided in this project document using a Terminal application:

```
sudo apt-get install -y python python-crypto python-serial
```

Once the above installation is finished, run the next command in a Terminal application:

```
sudo apt-get install -y mysql-server python-mysqldb
```

If the installation prompts for a root password, please enter a root password now. For the sake of installation, let us use the password, **“uest1onQ?”**. Once the SQL server and Python libraries are finished, start the SQL server daemon if it has not already done so, and log into MySQL:

```
sudo service mysqld start
mysql -u root -p
```

After the second command, the Terminal shall prompt for the root password. Enter the password now. Once logged into MySQL, create the database for our application, and then exit.

```
mysql> CREATE DATABASE IF NOT EXISTS nfc_account;
mysql> exit;
```

Navigate to where the Python Scripts are; this directory shall have the SQL dump that is prepopulated with data for demonstration purposes. Once in the proper directory, run this command, and then enter the root password:

```
mysql -u root -p nfc_account < nfc_account.sql
```

There should be no errors if all the commands were done properly up until this stage. To check, log into MySQL, provide your password, and execute the following commands.

```
mysql -u root -p
mysql> SHOW DATABASES;
```

Expected results should look similar to this:

```
mysql> USE nfc_account;
```

```
mysql> SELECT * FROM Account;
```

Prepopulated data should look like this:

Once done, to run the Python server, execute the following command after exiting MySQL:

```
sudo python python_example.py
```

Appendix III – Supervisor & Sponsor Sign-Off

Please contact Aman Abdulla, or Program Head for further information regarding the Supervisor & Sponsor Sign-Off.

Appendix IV – Project Listings & Manifest

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+
+     Manifest and Code Listing
+     Author: David Tran
+     Course: COMP8045 9-Credit Half-Practicum
+     Project: NFC & Smartphone Security
+
+     Located on-disk are the following files:
+
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

[ROOT DIRECTORY]
+ Arduino Code (Directory)
      - Arduino Libraries (Directory)
            + *Adafruit-PN532 (Directory)
            + *PN532 (Directory)
            + *PN532_HSU (Directory)
            + *PN532_I2C (Directory)
            + *PN532_SPI (Directory)
            + *NDEF (Directory)
      - android_hce_v2.ino (can be opened with Notepad++, etc.)

+ CardEmulation (Directory) <-- Import into Android Studio if possible…
      - Application (Directory)
            + src (Directory)
                  - main (Directory)
                        + java (Directory)
                              - com.example.android.cardemulation (Directory)
                                    + Account.java
                                    + AccountsDataSource.java
                                    + AccountStorage.java
                                    + CardEmulationFragment.java
                                    + CardService.java
                                    + KeyManagement.java
                                    + MainActivity.java
                                    + MySQLiteHelper.java
                                    + SecurityConstants.java

+ Python-Server (Directory)
      - Miscellaneous (Directory)
            + aes_example.py
            + aes_test.py
            + nfc_accounts.sql
            + pyserial_example.py
            + serial_reader.py
            + simple_serial.py
            + test.py
```

- AesCrypt.py
- AesCrypt.pyc
- C8045Util.py
- C8045Util.pyc
- NFC_Server.py
- note-4-public-key.txt
- note-4-private-key.txt
- server-public-key.pem
- server-private-key.pem

+ David Tran - COMP 8045 Final Report.pdf
+ David Tran - COMP 8045 Project Proposal.pdf
+ CardEmulation.apk

*Note: Libraries for Arduino shall be listed as whole directories with their content inside them. The Manifest shall not list all the individual files located within. Refer to the actual files themselves for further details.