Project Number: BS2 MQP 1301

MOBILE PROVISIONING INFRASTRUCTURE FOR TRUSTED COMPUTING

A Major Qualifying Project Report
submitted to the Faculty
of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the
Degree of Bachelor of Science

By
Dunia Abdulrazak, Jessica Pham, and Marc Michaud

Approved:

Professor Berk Sunar, Project Advisor

ABSTRACT

The purpose of this Major Qualifying Project was to address security concerns within the Android mobile platform with the help of our project sponsor, General Dynamics C4 Systems. The use of Trusted Platform Modules (TPM) was investigated to address security issues on mobile devices. In this project, a security architecture was developed in order to create a foundation for the use of TPM in Android devices.

TABLE OF CONTENTS

Table of Figures

GLOSSARY

Application Programming Interface (API) – protocol for interfacing software components

ARM TrustZone – hardware-based security that runs on two virtual processors; one runs the general operating system and the other runs more secure and sensitive data

Basic Input/Output System (BIOS) – set of built-in instructions located in firmware that loads an operating system from memory, and initializes and tests a system's hardware components

Confidentiality – limits the accessibility of information

Cloning – process of copying an existing system and its contents

Emulator – hardware and/or software that duplicates the function of one system in another system

Hash check – a cryptographic method that compares a derived value from a piece of data to its original in order to ensure it has not been changed

Integrity – to ensure that a system has not been tampered with

Intel Trusted Execution Technology (TXT) – type of hardware security for the Intel architecture that provides similar cryptographic functions as TPM

Log file – contains data of the events occurring on a device over time

Provisioning – grants users authorization to systems appropriate to their use of hardware or network resources

Pull download – connecting to a server at a scheduled interval/time in order to retrieve files

Push download – connecting to a device in order to download modified files

Remote Attestation - allows changes on a user's piece of hardware to be detected by authorized parties or systems

Root attack – allows the attacker to overcome the limitations set in place by the manufacturer of the device

Root of trust – the ability for the integrity of a system to be verified by ensuring that there have been no changes to the system's code

Security Enhanced (SE) Android – a version of the Android operating system developed by the U.S. National Security Agency that addresses security concerns within Android

Trusted Platform Module (TPM) – cryptographic processor capable of generating and storing cryptographic keys

1. INTRODUCTION

In modern society, mobile devices have become an essential part of users' lives. Despite the fact that mobile devices are widely used today, they still lack many of the security features offered by desktop computers. Due to the nature of wireless downloads and insufficient security, there are many potential areas where malicious programs or third parties could access the device without authorization. One of the major setbacks in mobile security is the lack of hardware-based security systems.

Hardware-based security is highly useful because it remains separate from the operating system of a given machine. This means that it does not possess the same vulnerabilities as a purely software-based security system. Hardware-based security systems, such as the Trusted Platform Module (TPM), provide methods of storing sensitive data by creating a root of trust. This root of trust ensures that the overall system has not been tampered with by verifying each layer of the system's boot process.

Research has been done to implement technologies similar to TPM into Android mobile devices, such as the ARM TrustZone, which also creates a root of trust. However, none of this research has yet led to the creation of a secure Android mobile device for distribution. The only Android-based mobile device that contains a root of trust is the Panasonic Toughpad tablet, which contains a hardware TPM chip.

In cooperation with General Dynamics C4 Systems, this Major Qualifying Project explores the development of a system that runs on the Android operating system and a dedicated TPM chip when its presence in mobile devices becomes more common. The developed system manages what applications are permitted to run on mobile devices based on a policy file created

by the administrator. This type of application is desirable in corporate environments, such as General Dynamics C4 Systems, where there is strict control of information. The integration of hardware-based security into this developed system would assist in creating a trusted environment for mobile devices. This would improve employee communication and productivity while decreasing the risk of malicious software or unapproved applications gaining access to the corporation's sensitive information. This requires that the software used within the developed system remains tamperproof.

The security architecture of the developed system for this project runs on the Android operating system and was created so that the integration of TPM would be possible. By ensuring this type of security is possible on an Android mobile device, it allows the developed system architecture to be ported onto other platforms in order to create root of trust within the system.

2. BACKGROUND

This section of the report will provide information on hardware-based technologies, the lifecycle of Android applications, and an overview of the project and developed system. The overall goal of this section is to provide a clear background for the understanding of the developed application's security architecture as well as useful technologies that could be integrated into the developed system.

2.1. HARDWARE-ENABLED SECURITY TECHNOLOGIES

The three main technologies developed in the field of hardware-enabled security are the Trusted Platform Module (TPM), ARM TrustZone, and Intel Trusted Execution Technology (TXT). These technologies all provide a way to store cryptographic information within hardware-based systems. The basis of security in these systems originates from using separate internal firmware that does not rely upon the operating system [1]. By creating a system that is separate from the operating system, it is possible to create root of trust. Root of trust is made of "hardware/software components that are inherently trusted," meaning that these systems must be able to perform security critical functions, such as software verification and device authentication [2].

2.1.1. TRUSTED PLATFORM MODULE (TPM)

The Trusted Platform Module (TPM) is the primary foundation for this project. TPM is a standalone crypto-based processor that is added into a computing system in order to provide additional security and root of trust. In terms of security features, the TPM provides "an RSA key generation algorithm, cryptographic functions like RSA encryption and decryption, a secure

random number generator (RNG), non-volatile tamper-resistant storage, and the hash function SHA-1" [3]. Figure 1 displays the different components that form the TPM.

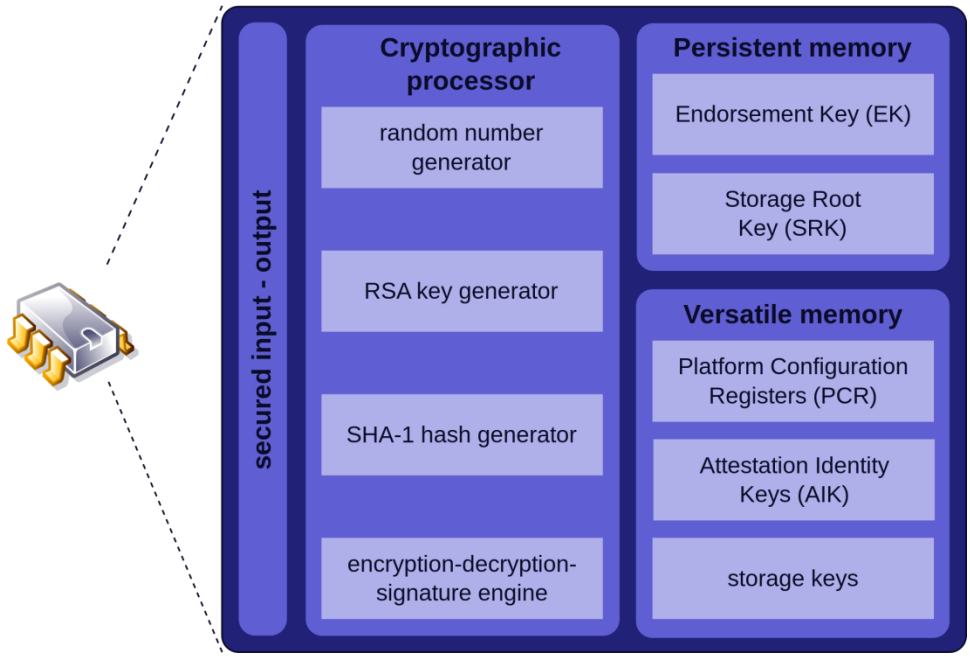

Figure 1: Trusted Platform Module security features [4]

When the Trusted Platform Module (TPM) becomes widely available to all mobile devices, it would allow for these devices and the provisioning application designed for this project, to become more secure. From a security standpoint, the major problem with storing files on any mobile device, such as on an Android smartphone, is that if rooted, the operating system would provide full access to all the files stored on the device. This means that any sensitive files are vulnerable to be read or manipulated. The TPM provides a way to generate and store RSA keys independently of the operating system. As a result, if TPM was available on mobile devices, a root attack would not compromise the TPM and the stored information would remain secure. TPM also provides the feature of verifiable attestation, which allows the Platform Configuration Registers (PCRs) to provide validation credentials. This is important for communication between

a mobile device and server for instance, in which the integrity of the device and connecting server is maintained. The verifiable attestation capability can be utilized to prove to the server that the mobile device can be trusted. This reduces the chance of unauthorized devices being able to connect to a server with malicious intent.

2.1.1.1. TRUSTED PLATFORM MODULE (TPM) ATTACKS

As mentioned previously, the Trusted Platform Module (TPM) is the primary foundation for this project in terms of hardware-enabled security. The TPM provides many security elements that could be utilized within a system, such as its self-contained cryptographic features and its ability to create a root of trust. Overall, the TPM appears to be a very secure platform, but in extreme cases, it can be compromised.

It has been determined that the key method for exposing the contents of a TPM is to reveal its Storage Root Key (SRK) or the private portion of the internal Endorsement Key (EK). These two components are essential to a TPM attack because if the SRK is revealed, then "all the keys in the TPM's key hierarchy are compromised. Knowledge of the private part of the EK enables the creation of software clones of a genuine hardware TPM" [5]. This means that if the SRK is discovered, all the cryptographic functionality of the TPM becomes ineffective and allows all the stored data to be decrypted. Additionally, if the EK is discovered, then the TPM can be cloned in software and the integrity of the system would collapse. In other words, there would not be a way to tell if there were any modifications in the TPM's stored data. However, obtaining the SRK and EK proves to be a difficult task since it is stored within the TPM and is never exposed to the rest of the computing system.

An alternative method to compromising the TPM would be to alter the integrity measurements. This requires false parameters to be inserted into the data stream as it is accepted by the TPM. This can be achieved by "modifying the CRTM [Core Root of Trust for Measurement] which is supposed to be immutable" [5]. If the CRTM was modified successfully, then the root of trust for the system would collapse, meaning that the reliability of the system will not be accurately assessed. However, in order to modify the CRTM, the attacker would need a way to intercept the commands that are being sent pre-BIOS.

The TPM can also be attacked through its communication bus with the CPU, which is known as the Low Pin Count (LPC) Bus. This communication is unsecured because "it is feasible to sniff the LPC bus and eavesdrop the TPM communication" [5]. Sniffing or eavesdropping on the LPC bus does not pose a direct threat to exposing the TPM's contents, but it does act as a gateway for other possible attacks to be carried out, such as the Reset Attack. The idea behind this type of attack is that it aims to reset the TPM, and as a result, clear its functionality. However, in order to successfully perform this attack, the attacker would need to gain physical access to the chip to "disconnect the LPC reset line from the TPM, either directly at the pin of the TPM-chip or at the connector of the TPM daughterboard" [5]. This would cause the TPM to reset without clearing the platform, allowing the TPM's data to be accessible.

One of the most difficult types of attacks that can be performed on the TPM is an invasive hardware attack. This type of attack involves physical intrusion into the chip, which would then allow details such as the SRK or other contents of the TPM's memory to be obtained. In order to attempt an attack like this, usually "a high budget, qualified specialists and expensive equipment such as a Focused Ion Beam (FIB), an electron microscope, a laser cutter, and/or micro probing station" would be required [5]. These tools would allow the material of the TPM

to be invaded, permitting a device such as a microscopic probe to read data off the chip. Additionally, an electron microscope could be utilized to directly read the EEPROM memory of the device. This type of attack exposes the contents stored within the TPM, but also poses a risk to damaging the TPM if the attack is not conducted correctly. In the case that an attacker successfully uses a microscopic probe, the TPM contains other security measures that obscures the data. This makes it harder for the attacker to interpret the data accessed on the TPM.

Currently, these attack methods pose a threat to the reliability of using TPM as a way to provide security features to a system. However, it is expected that the TPM will advance in the near future and gain the ability to counter these attacks.

2.1.2. ARM TRUSTZONE

The ARM TrustZone contains the ability to partition all of the SoC (System on Chip) hardware and software resources so that they "exist in one of two worlds – the Secure world for the security subsystem, and the Normal world for everything else" [6]. This means that the security critical code runs in a trusted environment while the other operating system routines run normally. Figure 2 displays the different elements that form the ARM TrustZone.

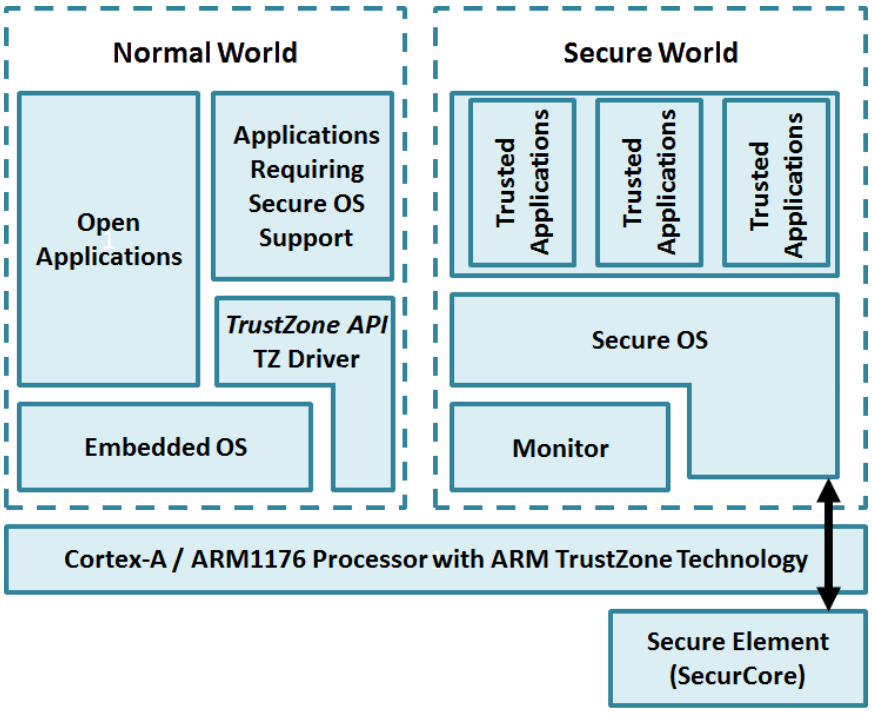

Figure 2: ARM TrustZone security architecture [7]

The functionality of the ARM TrustZone is based around the use of two virtual processors. The first processor runs the general operating system, whereas the second processor runs all the sensitive and secure elements. The ARM TrustZone is smaller and more restricted in the features it provides compared to the Trusted Platform Module (TPM). However, the use of this technology would still benefit this project by running the most sensitive aspects of the provisioning software within the secure world provided by the ARM TrustZone. This would allow for secured memory and crypto blocks on the device, greatly reducing the chance of tampering and ensuring that all files used by the provisioning application remains secure.

2.1.3. INTEL TRUSTED EXECUTION TECHNOLOGY (TXT)

The Intel Trusted Execution Technology (TXT) device focuses on creating a root of trust based on safe boot and attestation. However, the Android operating system runs almost

exclusively on ARM based processors and the TXT is meant for Intel chips. This technology could still benefit this project by improving server security in the case that the developed security architecture is ported onto a non-Android platform. The TXT device "creates a Measured Launch Environment (MLE) that enables an accurate comparison of all the critical elements of the launch environment against a known good source" [8]. This means that the TXT device contains the ability to validate the boot process of a piece of software. This feature would benefit this project by ensuring that the server has not been tampered with and contains safe software and code that would be distributed to mobile devices by comparing it to a known configuration. Figure 3 depicts the decision making process of the TXT system.

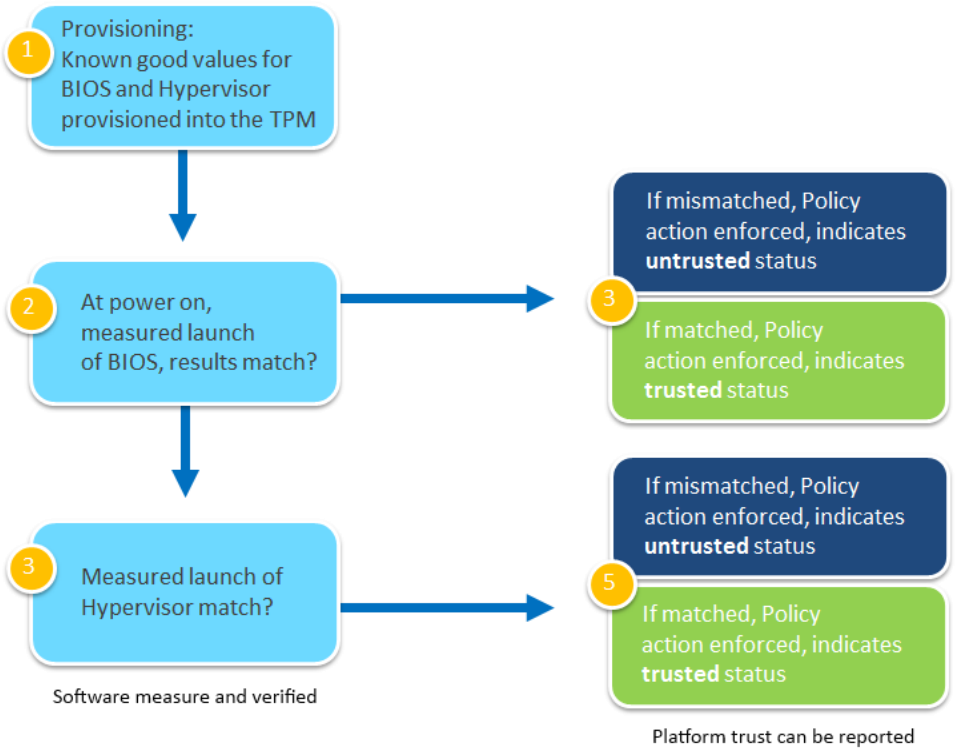

Figure 3: Trusted boot process for Intel Trusted Execution Technology [8]

The Intel Trusted Execution Technology device would further benefit this project due to attestation. This would allow a hardware-based method of verification during communication

between the server and mobile device. In other words, the TXT device would be able to identify and ensure the security of the server has not been compromised and is not running unauthorized software prior to Secure Sockets Layer (SSL) communication.

2.2. ANDROID ACTIVITY LIFECYCLE

In order to understand how the developed application for this project functions, it is necessary to know how the lifecycle of an Android application functions. Every Android application is comprised of activities, which allows the user to execute certain tasks. These activities enter a series of lifecycle states when the application launches. There are a total of six lifecycle states, but the three main states that are mentioned within this report are ***onResume()***, ***onPause()***, and ***onStop()***. The ***onResume()*** state is encountered when the activity is launched for the first time or when the user returns to the activity. The ***onPause()*** state is encountered when another activity comes into the foreground, and the ***onStop()*** state is encountered when the activity is no longer visible. Figure 4 depicts how each Android activity lifecycle state is encountered.

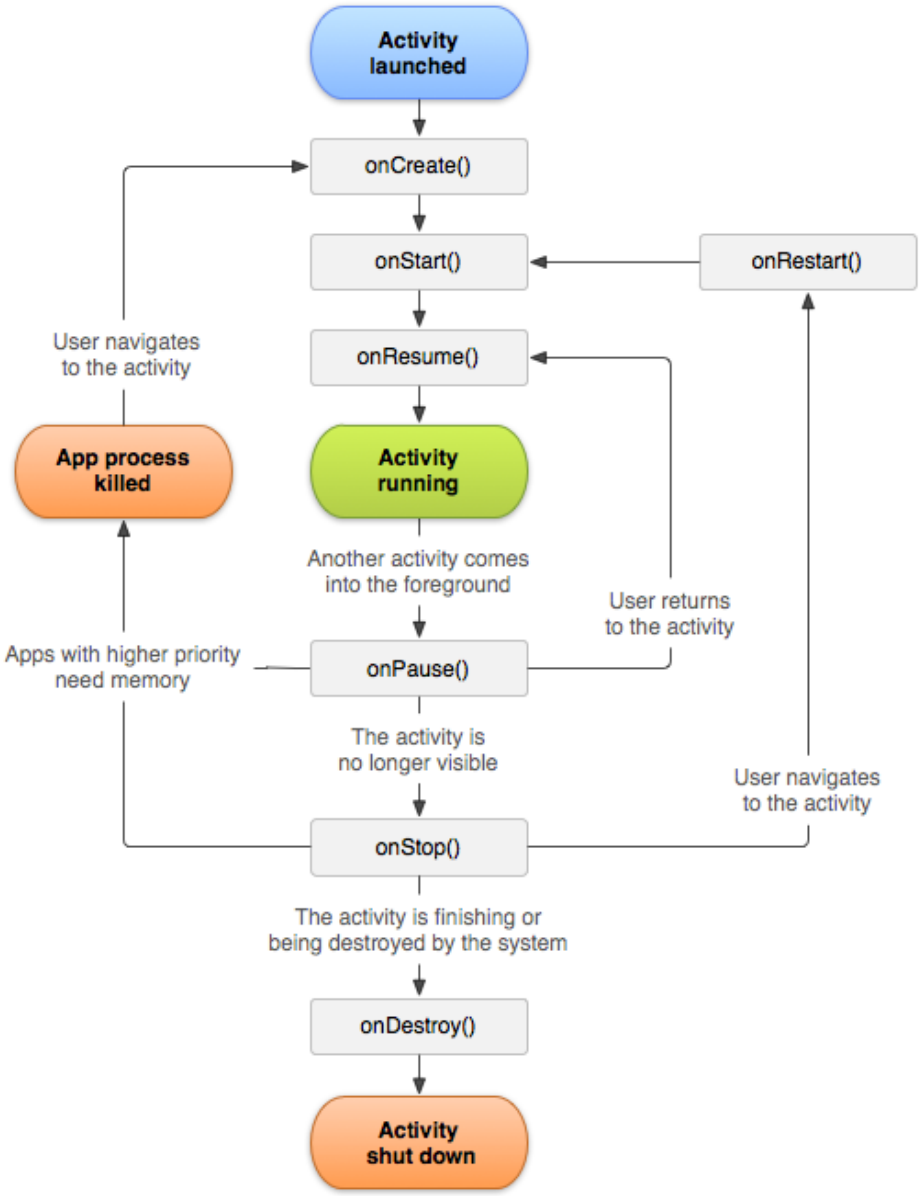

Figure 4: Activity lifecycle of an Android application [9]

2.3. PROJECT OVERVIEW

The main goal of this Major Qualifying Project is to provide a foundation for the implementation of the Trusted Platform Module (TPM) onto mobile devices. This was completed by creating an Android application and a security architecture that would easily allow

the integration of TPM when it becomes widely available on mobile platforms, such as tablets and smartphones. This Android application provides corporations with the ability to provision or moderate the applications that can operate on the company's mobile devices. This is crucial since many employees can exchange proprietary, confidential, and highly secretive information over corporate mobile devices, and if malicious software or third party gained access to this information, it could be detrimental to the corporation's operation.

2.3.1. ANDROID APPLICATION

The Android application developed for this project is titled the Application Manager. This application is able to regulate applications on corporate mobile devices by determining whether the launching application is on the approved list of applications.

When a third party application begins to launch on the mobile device, the Application Manager then launches and checks to determine if the third party application contains a valid lock file. The lock file is a temporary licensing file that was designed to speed-up the verification process and minimizes the system's impact on the device. If a valid lock file exists, the third party application continues operation. However, if a lock file does not exist or is expired, the Application Manager must then verify if the application is permitted to run using the policy file. The policy file contains the package name and signature of all the permitted applications that can run on the mobile device. If the application is listed in the policy file, the third party application resumes operation, but if it is not on the policy file, the Application Manager then prevents it from operating. The user is then prompted with the option to update the policy file or exit both applications (the third party application and the Application Manager).

3. METHODS

This section of the report will discuss the primary components for this project: server, security, and application. These components form the basis of the Application Manager, where the first two components form the security architecture of the system and the last component is essential for application control. This section will explain the purpose of each component as well as its implementation and encountered problems.

3.1. SERVER COMPONENT

The purpose of the server is to provide a way of distributing the policy file over a corporate network. Using a server allows every mobile device connected to the network to obtain a copy of the policy file automatically. This provides a more efficient method for distributing files compared to uploading the file to each individual phone at any given time.

The structure for intercommunication was composed of three parts: the server, mobile device, and the device's internal memory. The purpose of the server is to host the policy file for distribution to mobile devices. The mobile device is the host for the Application Manager and acts as a receiver, in which it receives the policy file from the server. The device's internal memory is used to store a local copy of the policy file since it is the most secure location for file management within Android applications. Figure 5 shows the three parts that form the intercommunication structure for the Application Manager.

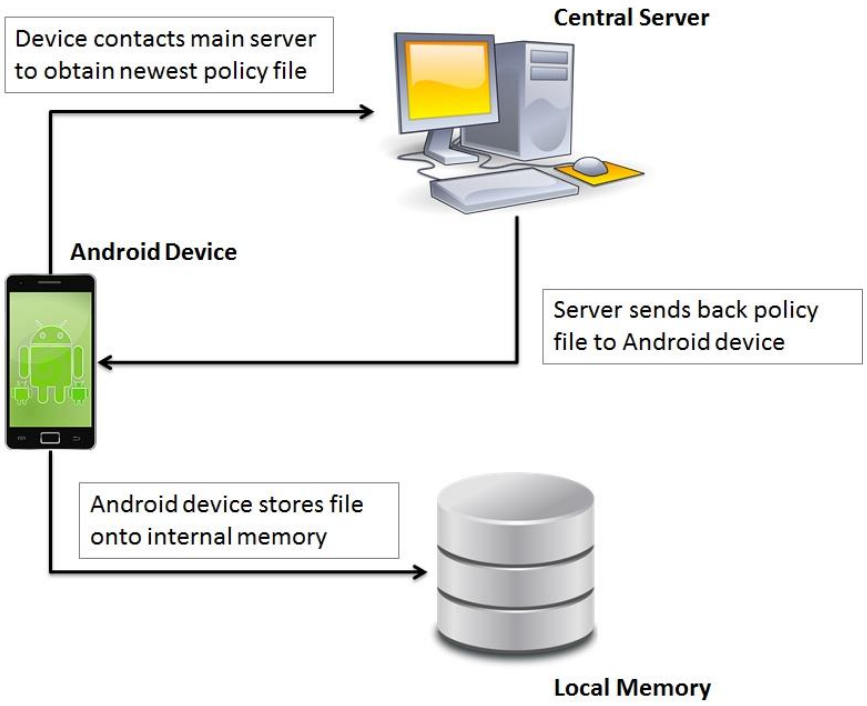

Figure 5: Main components of the intercommunication structure

The server hosts two files that are used by the Application Manager: the encryption key file and the policy file. Once per day the Application Manager attempts to connect to the server in order to download the latest versions of these files into the device's internal memory. Several security measures were taken in order to ensure the confidentiality and integrity of the download process and files. For example, when the mobile device connects to the server, the identity of the server is validated via a third party signed certificate. Additionally, during the download process, Secure Sockets Layer (SSL) protocol is utilized to prevent third parties from interfering with the download.

3.1.1. CHANGES/PROBLEMS ENCOUNTERED

3.1.1.1. FILE DISTRIBUTION

A server would be used to host and distribute files to a large number of devices connected to a corporate network. For this project, only a single device required access to the hosted files since the Application Manager was not distributed to anyone outside of the project team. As a result, it was determined that a website would be sufficient for hosting and distributing the files to a single device. A website was chosen because it could easily provide the desired layers for security and is faster to setup than a server.

The website was hosted through Worcester Polytechnic Institute (WPI) because of its provided resources. WPI provided access to the SSL protocol and a third party signed certificate. If the website was created by other means, the SSL credentials and third party signed certificate would have to be separately obtained which is costly.

Originally, the website was set up so that login credentials, in the form of a username and password, were required to access the site. This was achieved by using a hypertext access (htaccess) and a hypertext password (htpasswd) file. However, during the implementation of a login mechanism for the site, a series of permission denied errors were encountered. Upon further research and consultation with server administrators, it was determined that users are not permitted in implementing the functionality of password-based login services. As a result, WPI's standard login protocol was used.

WPI's login protocol requires a valid WPI login in order to access the website. In other words, no one outside of the WPI community would be able to access the website. Ideally, the server's login credentials would allow anyone connected to the corporate network to access the

files, but it was determined that WPI login credentials would suffice for this project. While implementing the WPI login credentials, it was determined that Uniform/Universal Resource Locator (URL) redirection was necessary. This proved to be problematic because the login appeared to be incompatible with the Hypertext Transfer or Transport Protocol (HTTP) post commands, which is utilized in order to post user login credentials to the website. As a result, login credentials were not implemented on the website since the implemented SSL protocols would ensure the confidentiality and integrity of the hosted files.

3.1.1.2. DOWNLOAD SYSTEM

Originally, a push download system was to be implemented in order to update the encryption key and policy files on the mobile device. A push system is the process where the server sends a signal to the device in order to notify it that files have been updated on the server. The push system would then enable the device to communicate with the server in order to obtain the updated files. Figure 6 shows the steps taken in a push-based download.

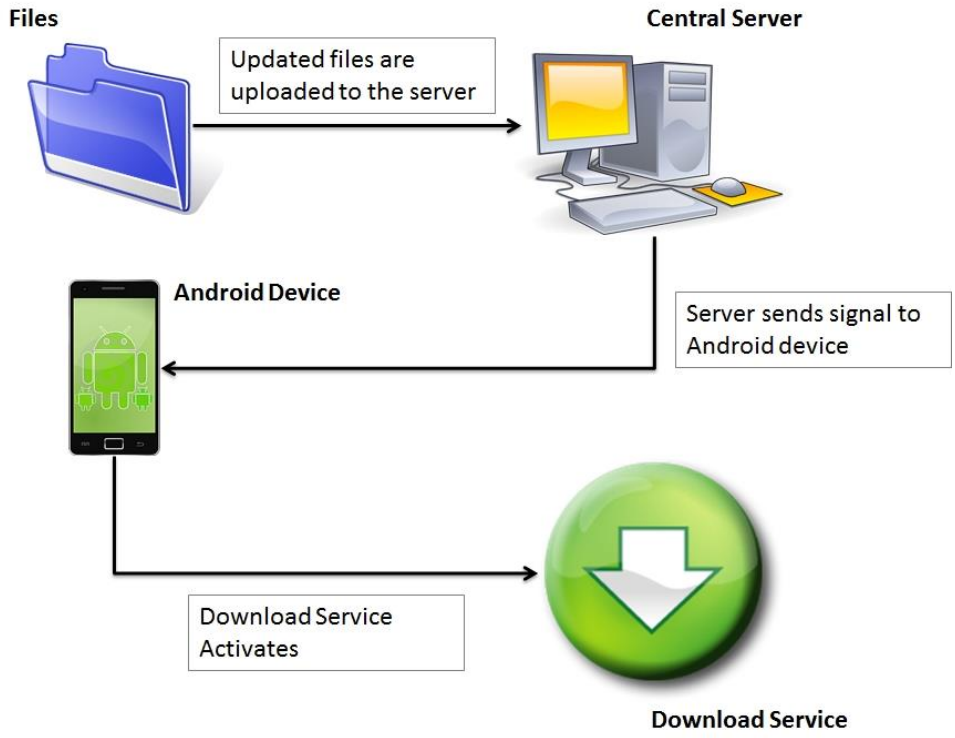

Figure 6: Push-based download

Due to the restriction of administrative privileges on WPI hosted websites, a push download system could not be implemented. As a result, a pull download system was used, in which the devices communicates with the server and downloads the hosted files based on a periodic schedule or when prompted by the user. Figure 7 shows the steps taken in a pull-download.

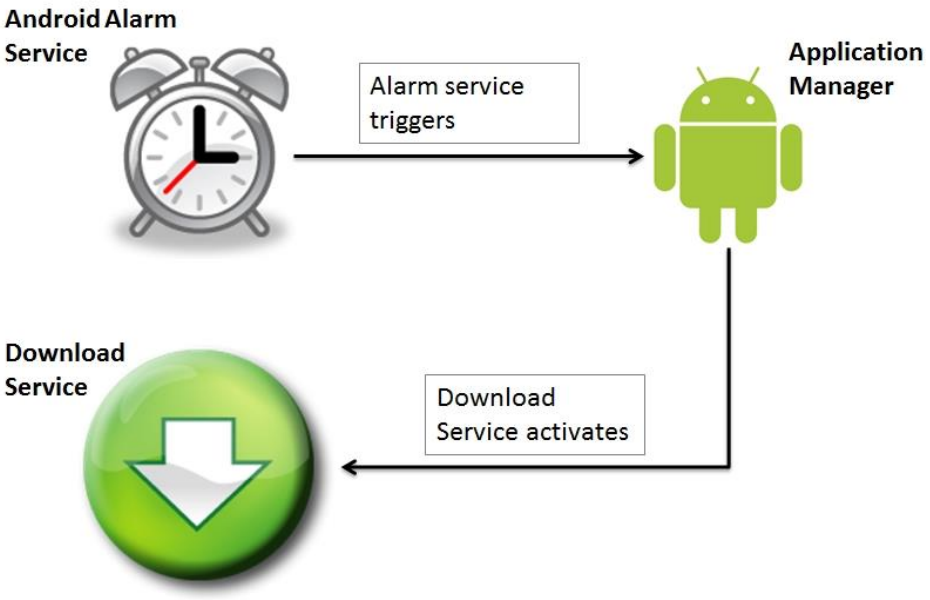

Figure 7: Pull-based download

The disadvantage to using a pull-based download system is that it is less battery efficient. However, this was taken into consideration during the design process and in order to minimize the drain on the device's battery, the pull download system is initiated only once per day. In other words, the device connects to the website and downloads the hosted files only once per day.

The pull system was implemented within the Application Manager by creating a repetitive service, where an Android service is a function that starts, completes its given task, and then stops. Figure 8 shows the process flow for the implemented pull-based download system. The benefit of implementing the download system as a service is that it will continuously run even if the Application Manager is not currently active. In other words, as long as the phone is on, the service will run. This download service is initiated when the Application Manager is launched for the very first time on a given device.

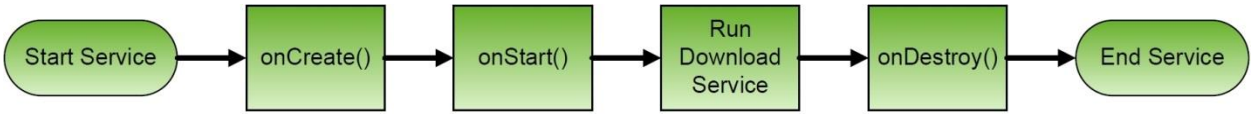

Figure 8: Life cycle for download service

Within the implemented service, a method is called that is responsible for the download process. The download process itself utilizes the ***HttpConnection*** interface within the ***org.apache.http*** Android Application Programming Interface (API) in order to establish a connection with the website. Once the connection is established, a file stream is opened and a file is created within the base directory of the device's internal storage. The information to be transmitted is then read through a secure stream and written to the file. Once this process completes, a new method is called in order to setup the next call to the service (for the following day). After the time for the next download is set, the service terminates, and the process repeats itself once the service is called again (the following day).

3.2. SECURITY COMPONENT

3.2.1. SECURE SOCKETS LAYER (SSL)

One of the largest concerns during this project regarded ensuring that the communication between the mobile device and the server remained secure. In order to reduce the risk of third parties accessing the download or manipulating the download procedure, Secure Sockets Layer (SSL) was implemented. SSL is a protocol that ensures the integrity of the download.

SSL is used to protect against eavesdropping during the download procedure and to protect against tampering of downloaded files. If SSL was not implemented and a third party was eavesdropping on the download, they would then be able to acquire a copy of the policy file. If the third party then managed to decrypt the policy file (originally encrypted), they could modify

it to allow any applications to run on the mobile device. The use of SSL provides a layer of security to ensure the integrity of the policy file, and without it, the Application Manager would not be able to determine if the third party tampered with the policy file. Additionally, if SSL was not implemented, a third party could tamper with the download connection between the server and mobile device. This would allow the policy file to be replaced with a malicious file or tampered with, as previously mentioned.

The download connection is a secure process that utilizes a "handshaking" procedure to provide security credentials to the mobile device before the download occurs. SSL allows for the communication between the device and server to be encrypted during this "handshaking" session. In other words, during the download procedure, any data that is transferred between the server and mobile device becomes unreadable without the session key. If a third party manages to intercept the download, they would not be able to read the data and as a result, the integrity of the data is maintained throughout the download.

During the download procedure, the information that is to be sent from the server to the mobile device is divided into smaller portions called packets. The server signs each packet that is transmitted during the download, which is crucial in ensuring the integrity of the files. Once a file is signed, any changes made by a third party would be detected by the receiving end of the download (the mobile device). As the mobile device receives packets, a hash check occurs to ensure that the signature matches the corresponding packet. If the signature does not match, it could indicate that the file became corrupted during the transmission or that a third party could have tampered with the files. Figure 9 depicts the flow of communication and the steps taken during the SSL download procedure.

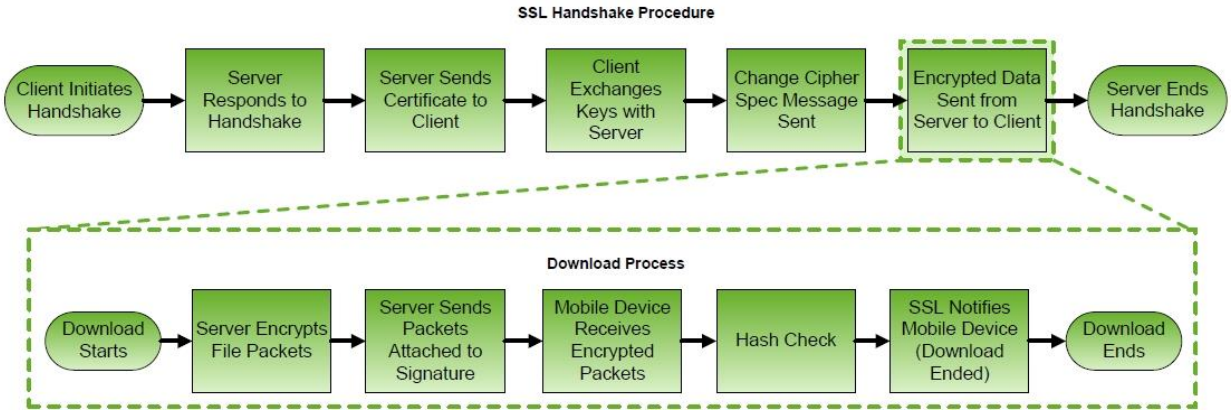

Figure 9: Secure Sockets Layer download procedure

Ensuring the integrity of files is critical to the architecture of the Application Manager because it greatly reduces the chance of malicious software affecting a corporate network. The architecture of the Application Manager is also designed such that the policy file is also encrypted. This provides an additional layer of security to the policy file because encryption also occurs naturally within SSL.

3.2.2. ENCRYPTION

In order to ensure that the Application Manager could not be easily altered in a way that could allow any application to run, all files used by the Application Manager were encrypted with a 128-bit AES (Advanced Encryption Standard) key. AES encryption was chosen over other types of encryption because it is widely used among government and business organizations.

The purpose of encrypting files used by the Application Manager is to ensure the confidentiality of the policy file by making it harder for potential hackers to add applications to the file. In other words, if the policy file was not encrypted, a potential hacker could open the policy file and easily make modifications to it in order to allow any application to run on the mobile device. Additionally, the use of AES encryption added a layer of security to the lock file

by making it more difficult for potential hackers to modify how long an application could run without being verified by the Application Manager.

3.2.3. APPLICATION VERIFICATION

The Application Manager validated that an application could run by cross-referencing both its package name and signature to the policy file. The reason the package name is used for verification is because it is an application specific identifier. In other words, no two applications can exist on a single mobile device and contain the same package name. The application signature is also used for application verification because each released Android application must be signed, and the signature used to sign an application is unique to its developer. Meaning, every released application is signed with a unique key unless the same developer created the applications.

The purpose of using both the package name and signature for validation was to protect against unreleased applications from running. An unreleased application is an application created in the Android SDK (Software Development Kit) that has not been released to the public or application store, and as a result, it can be created to have the same package name as an existing application. Therefore, if the Application Manager verified applications solely by the package name, unreleased applications would be able to "get around" the Application Manager's security protocols.

3.2.4. CHANGES/PROBLEMS ENCOUNTERED

3.2.4.1. ENCRYPTION

Originally, the type of encryption applied to the files used by the Application Manager was chosen to be DES (Data Encryption Standard). In order to perform DES file encryption,

research was done to determine how to generate keys within the Android SDK environment, as well as determine how to use the generated key for file encryption and decryption. After some research, it was determined that the ***javax.crypto*** API (Application Programming Interface) contained a ***SecretKeyFactory*** class that could generate a 56-bit DES key, and that the ***Cipher*** class would allow for file encryption/decryption.

Once the functionality of the file encryption/decryption code was verified, the next goal was being able to store the DES key in a file and then reading the DES key back out. After attempting to write the raw bytes of data representing the DES key to a file, it was determined that the decryption functionality of the code was no longer functioning correctly. This meant that when the file containing the DES key was read, it was either not reading in the correct data from the file, or it was unable to successfully convert the ***byte*** array to a ***SecretKey***. After some debugging, no easy solution could be found to solve this issue.

As a result, further research was done leading to resources recommending the use of the ***KeyStore*** class located within the ***java.security*** API in order to store the DES key. However, when testing this method, errors occurred stating that the ***KeyStore*** class could not store Secret keys, which was the data type of the DES key. After some debugging, further research was performed to try to find a different method to store a DES key to a file. It was then determined that the DES key could be converted to a string, and then stored in a file. Once the code was written, testing was then performed to ensure that the DES key was being written to a file and correctly read back out, and that the code for file encryption and decryption was still functioning.

Eventually, the type of encryption applied to the files used by the Application Manager was changed from DES (Data Encryption Standard) to AES. The main reason for this change was that AES encryption is most commonly used today. Additionally, AES keys are harder to

"crack" than DES because of its complex cipher, which allows the files used by the Application Manager to be as secure as possible if encrypted with an AES key.

This change was implemented by performing further research into the generation of AES keys, and how to perform AES encryption. After some research, it was determined that AES keys are generated very similarly to DES keys, except that AES keys require a password in order to actually generate the key . Once the functionality of AES key generation was verified and that files were successfully being encrypted and decrypted using the key, the next step was to store the AES key. This was done by reusing the code previously written to store a DES key.

3.2.4.2. APPLICATION VERIFICATION

Originally, the Application Manager validated that an application could run by comparing the application's assigned RSA (Rivest–Shamir–Adleman) key to its corresponding key stored in the policy file. This was done by conducting research on how to generate a RSA key pair and how to store the private key in a file. Research into this area revealed that the RSA key pair could be generated using the ***java.security*** API and the ***KeyPairGenerator*** and ***KeyPair*** classes.

After verifying that the RSA key pair was generated correctly, the private key was then extracted, converted into a ***byte*** array, and then written to a file. Once the functionality of writing the private key, and then later reading the key, was confirmed to be working correctly, the next goal was to store multiple Key Pairs in a single file (one ***KeyPair*** for each application on the policy file). However, after conducting some research, it became clear that storing multiple key pairs in a single file would prove to be difficult since a private key contains additional text besides the actual key (private keys contain headers).

As a result, instead of generating an RSA key pair for each application allowed to run on the mobile device, it was decided that a single RSA key pair would be used. In other words, all of the applications allowed to run would be assigned the same RSA key, rather than different ones. In order to assign the RSA keys to the applications permitted to run, the package name of each application needed to be obtained. This would allow the Application Manager to be able to distinguish which application could run (the package names would be implemented in the policy file). The package names were acquired by using the ***android.content.pm*** API and the ***PackageManager*** class.

Eventually, the method used to verify applications by the Application Manager was changed from RSA key authentication to package name and signature validation. The main reason for this change was that in order to verify the application by its assigned RSA key, the package name of the application would need to be verified first. As a result, this particular method of application verification seemed redundant because the application could just be verified by its package name. Research into this area then revealed that unreleased applications could be created to have the same package name as an existing application. Therefore, it was decided that instead of validating applications by using an assigned RSA key, the applications would be verified by the package name and signature.

This change was implemented by obtaining the package name and signature of each application permitted to run. Once the functionality of opening the policy file and reading it was verified to be working correctly, the next step was to confirm that the package name and signature of the running application exists on the policy file. This was done by modifying the previously written code to obtain an application's package name, which would then allow the application's signature to be obtained as well.

3.3. APPLICATION COMPONENT

3.3.1. POLICY FILE

The policy file is an encrypted file containing the package name and associated signatures of applications approved for use on the corporate network. A list of approved applications was chosen since it would be much easier to manage because the number of permitted applications would most likely be less than number of prohibited applications. At a specified time interval (daily), the policy file is updated by a service running within the Application Manager, in addition, there are instances where the user can manually update the policy file. The Application Manager uses the policy file by verifying that it contains the obtained package name and signature of the launching application. After this verification, the Application Manager either permits the application to operate or prevents it from running.

3.3.2. APPLICATION REGULATION

The Application Manager determines that a launching application is permitted to operate when the package name and signature of the third party application matches the package name and signature stored on the policy file. Utilizing the activity lifecycle of Android applications, the Application Manager simply exits itself, permitting the third party application to continue operation. However, if the third party application is not on the policy file, the Application Manager prevents its operation. Furthermore, the Application Manager is not able to fully terminate an unrelated application (putting the application to lifecycle state ***onDestroy()***), the Application Manager creates an infinite loop between the third party application and itself. This is implemented by the third party application (the developed test application) calling the Application Manager when it enters the ***onCreate()*** or ***onRestart()*** states in the Android

lifecycle. When the third party application is not on the policy file, the user has the option to exit both applications via the exit button on the device (the user is taken to the device's home screen) or updating the policy file via the update button. Figure 10 outlines the lifecycle process of preventing a blocked application from operating.

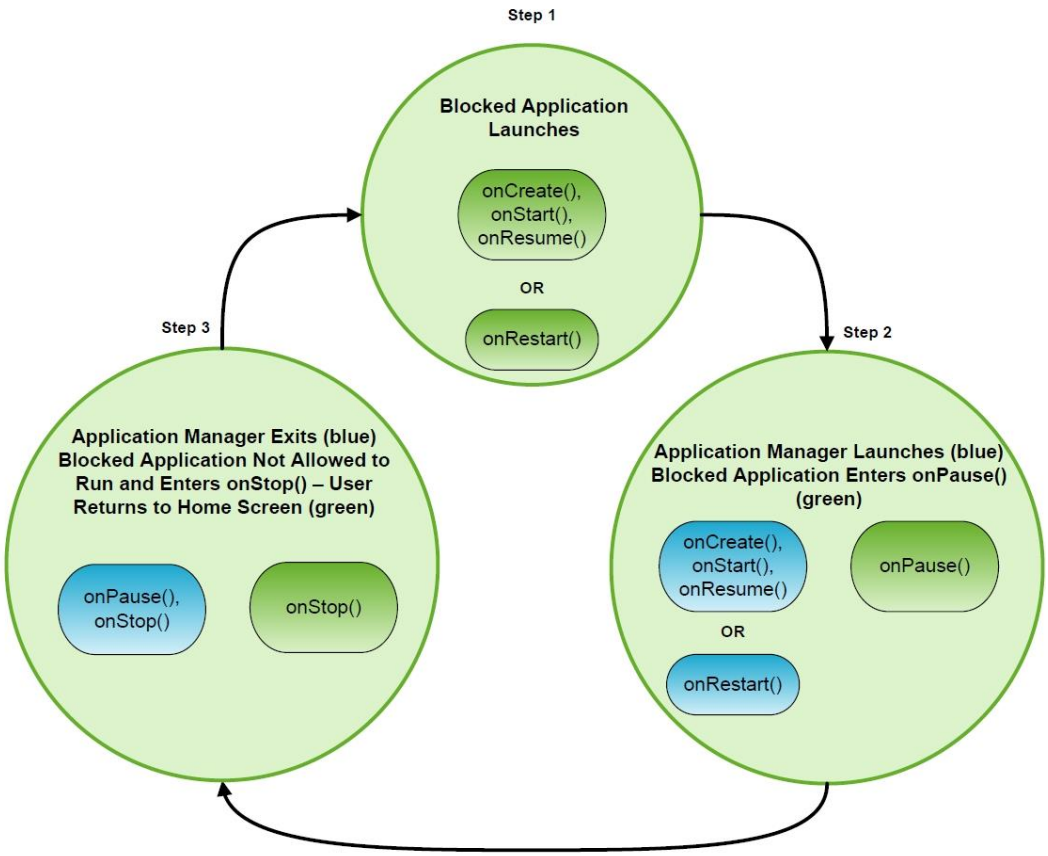

Figure 10: Blocked application lifecycle process

3.3.3. CHANGES/PROBLEMS ENCOUNTERED

3.3.3.1. LAUNCHING THE APPLICATION MANAGER

Originally, the Application Manager would launch once a third party application's launch was detected. However, research into this area did not indicate that there as a way for the Application Manager to interrupt the lifecycle of a third party application's launch. As a result, a

different method was implemented in order to initiate the Application Manager's launch. Research into the ***android.content.pm*** API and the ***PackageManager*** class led to the discovery of an Android command that would permit an application to initiate the launch of another application using its package name. In other words, test applications were created that programmatically launched the Application Manager when it entered the ***onCreate()*** or ***onRestart()*** lifecycle states.

3.3.3.2. OBTAINING THE PACKAGE NAME OF THE LAUNCHING APPLICATION

Once the test applications are able to launch the Application Manager, the Application Manager needed to identify and verify the third party application. The best way to identify the test application was by its package name, since it is unique to every application that is publicly distributed. Originally, the Application Manager would be able to query and obtain the package name of the third party application independent of the third party application (no hard coding). However, upon research and attempts at accomplishing this task, it was determined that the Application Manager would be unable to obtain the package name of the third party application without hard coding it within the test application. The final solution for the Application Manager to be able to obtain the package name of the test application was to have the test application include its package name when it launches the Application Manager.

3.3.3.3. TERMINATING AN UNAPPROVED APPLICATION

A large component of the Application Manager is terminating an unapproved application. Research into the activity lifecycle of an application indicated that there are ways for an application to be terminated (reach the ***onStop()*** or ***onDestory()*** lifecycle states). However, when attempting to have the Application Manager terminate a specified application, the task was not

completed as expected. Attempts at terminating the application led to research into how current task killer applications terminate other applications.

Research into terminating applications indicated that it is best not to terminate unrelated applications, and to allow the Android system and lifecycle handle the termination of applications. Using kill commands from the ***android.os*** API and ***Process*** class, or from the ***android.app*** API and ***ActivityManager*** class, appeared to be the solution. However, implementing these commands and further research illustrated that these commands do not completely terminate an application. In the end, the Application Manager was designed to simply prevent the user from using the blocked application instead of actually terminating the application. The Application Manager accomplished this by returning the user to the home screen of the device and placing the blocked application into the ***onStop()*** lifecycle state. As a result, if the blocked application was launched again, it had to enter the ***onRestart()*** lifecycle, where the Application Manager would be called. This created an infinite loop between the two applications and the home screen in order to prevent the user from using the blocked application.

4. TESTING

This section of the report will discuss how the Application Manager was developed by integrating the previously mentioned components: server (download), security (application verification), and application (application termination). This section will also discuss the procedure used for testing and encountered problems.

4.1. INTEGRATION

Once the code for all three components was written and tested for functionality, they were integrated to form the Application Manager. The integration of all three portions was completed in two steps: combining the application verification and application termination methods, and then incorporating the downloading method. Once all three portions were successfully integrated, exhaustive testing was performed to ensure the Application Manager functioned as expected. Figure 11 shows the steps taken during the integration process.

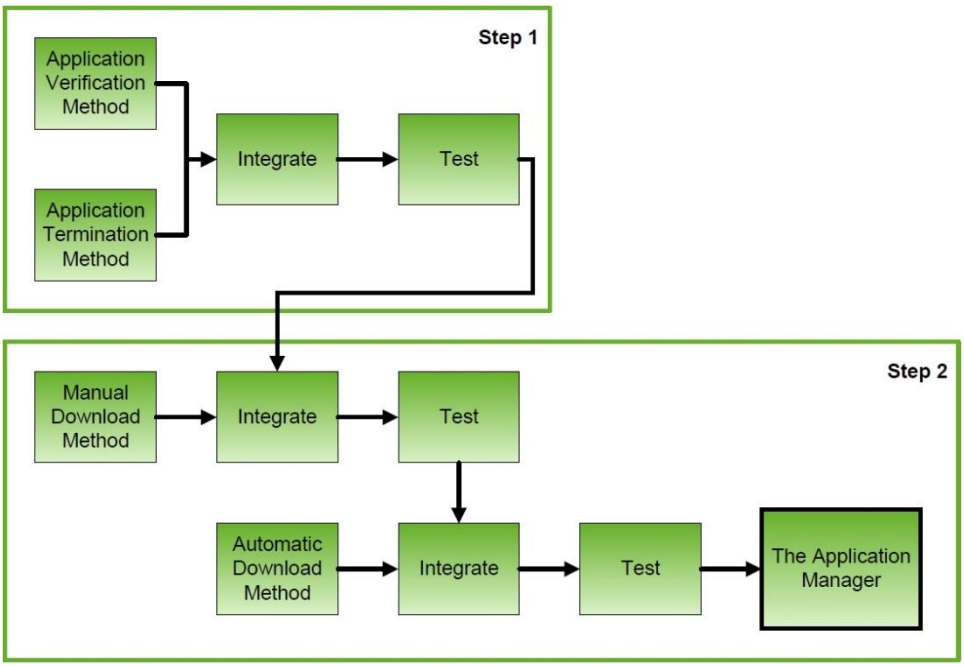

Figure 11: Steps taken to create the Application Manager

4.1.1. STEP 1: APPLICATION VERIFICATION & TERMINATION

The first step in creating the Application Manager consisted of combining the first two methods: application verification and application termination. This was done by simply incorporating the code written to terminate an application within the code written to verify an application. In other words, the code for terminating an application was not modified, and was just simply added into the code written for application verification.

Once these two methods were incorporated into a single piece of code, testing was done to ensure the code was incorporated correctly. This was done by verifying that the functionality of both pieces of code was not affected once they were combined into a single piece of code.

4.1.2. STEP 2: DOWNLOAD

After it was determined that the first step of the integration process was functioning correctly, the download method was included. This was done in two parts: adding the manual download method and adding the automatic download method. In order to incorporate the manual download method, the code for performing a manual download of the policy and key files was simply added to the existing code (produced in first step of the integration process). This was completed by making minor modifications to the manual download code and the existing code.

Once the manual download method was incorporated into the existing code, testing was done to ensure the code was functioning as expected. This was completed by verifying that the files were downloaded once the download button was pressed, and that the functionality of the application verification and termination methods were unaffected.

After it was determined that the code was working correctly, the automatic download method was incorporated into the code. This was done by making minor modifications to the automatic download code before it was added to the existing code. Once the automatic download code was incorporated, testing was completed to make sure the code was functioning correctly. This was accomplished by verifying that the first step of the integration process still functioned, that the manual update still functioned, and that the automatic download was working by downloading the files at a regular interval.

4.2. TESTING PROCEDURE

Once all three portions were integrated into a single application, testing needed to be done to ensure that the Application Manager functioned correctly. This was executed by creating a checklist of the different situations the Application Manager would encounter.

The first scenario that was tested dealt with only launching the Application Manager (not trying to launch an application). There were two expected outcomes for this situation. The first was that when the user clicked the update button, the policy and key files would be downloaded and stored within the device's external storage. The second expected outcome was that the policy and key files would be updated every two minutes. Figure 12 shows the process flow for the case in which the user tries to launch the Application Manager only. In addition, the automatic update interval was set for two minutes only for testing purposes; in the final product, the interval was changed to once a day.

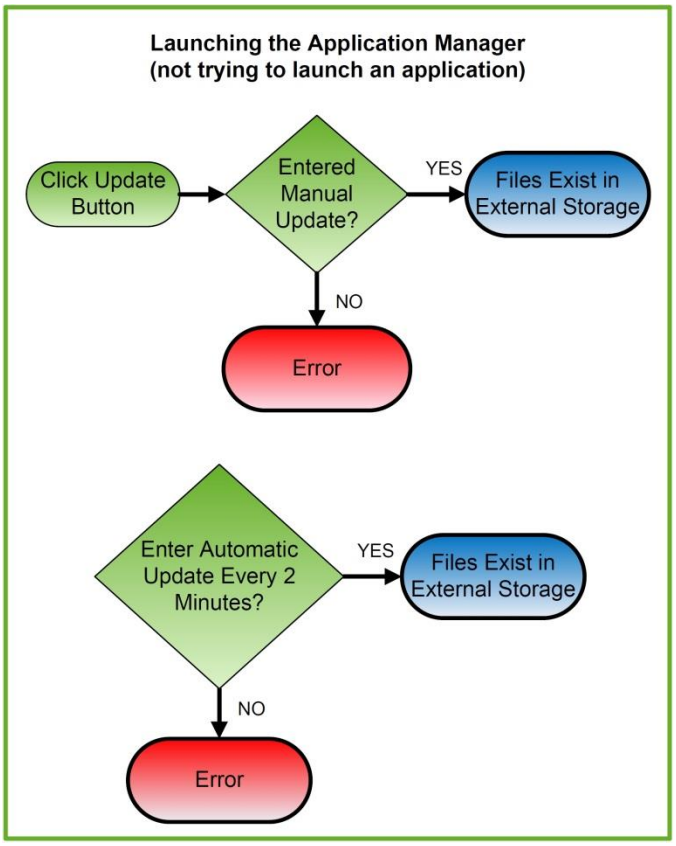

Figure 12: Launching the Application Manager

The second scenario that was tested dealt with trying to launch an application that is on the policy file for the first time (or after its lock file has been deleted). The expected outcome for this situation was that the application would be on the policy file and as a result, the application would be permitted to run. Figure 13 shows the steps taken by the Application Manager after the application is launched for the first time. The Application Manager checks to see if the application has a lock file, which it should not because this is the first time the application is launching. The Application Manager then locates and reads the policy file, which should be stored in internal memory. If the application is listed in the policy file, a lock file is then created and encrypted, and the application is resumed.

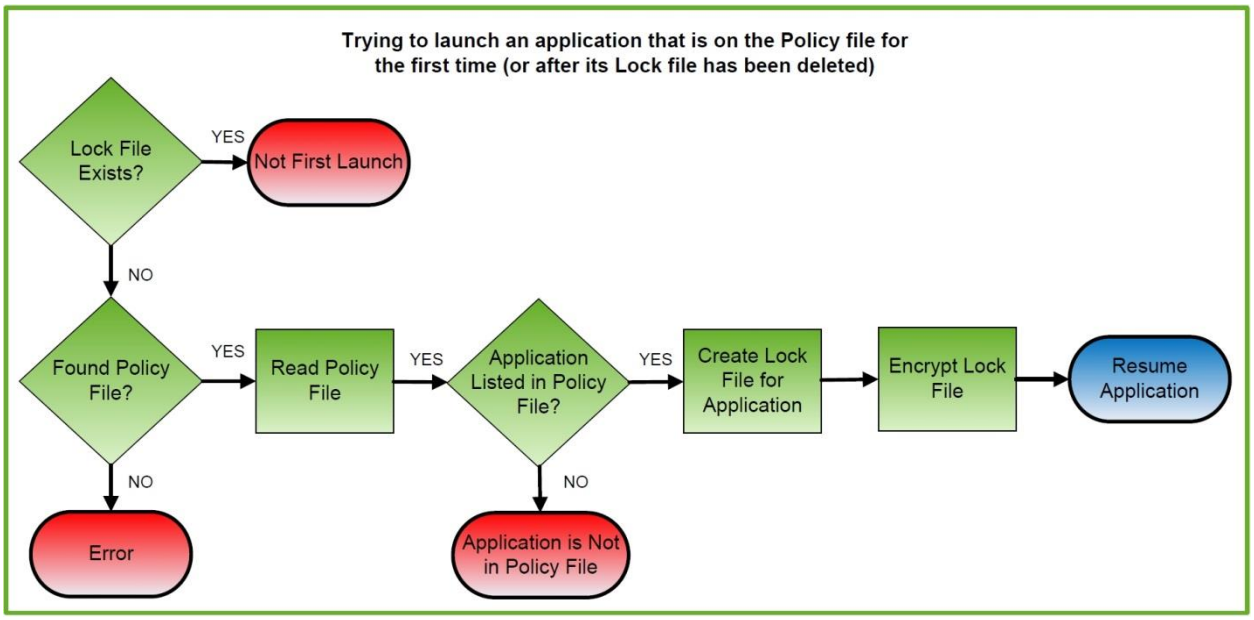

Figure 13: Launching a permitted application for the first time (or after its lock file has been deleted)

The third scenario that was tested dealt with trying to launch an application that is on the Policy file (not first launch). There were two possible outcomes for this situation. The first is that the lock file would not be expired and the application would be permitted to run. The second outcome is that the lock file has expired which would cause the application to be terminated (user would be taken to their device's home screen). Figure 14 shows the steps taken by the Application Manager after the application is launched.

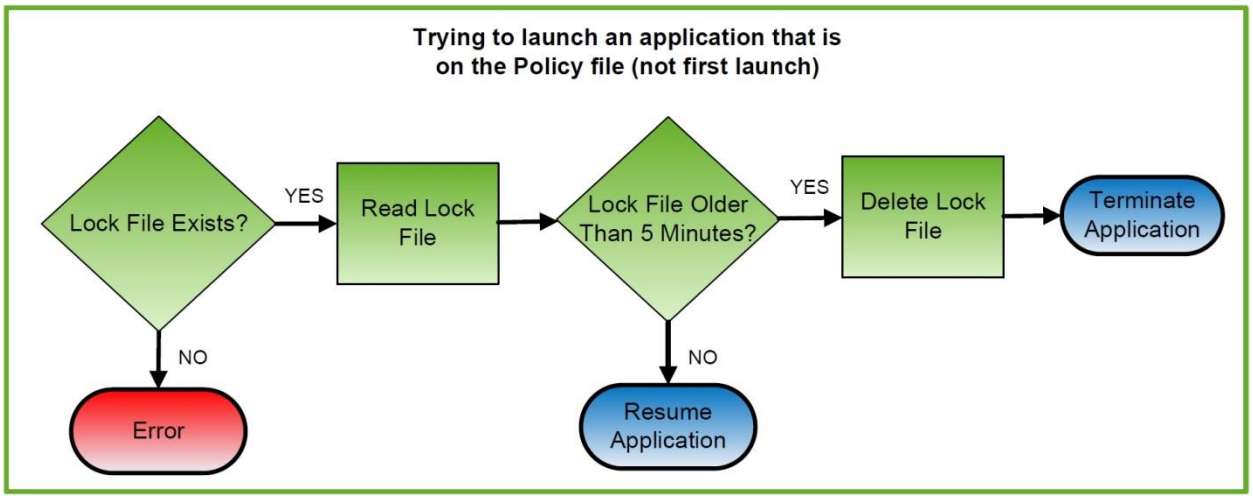

Figure 14: Launching a permitted application (not first launch)

The fourth and final scenario that was tested dealt with trying to launch an application that is not on the policy file. The expected outcome for this situation was that the application would not be on the policy file and as a result, the application would not be able to run. Figure 15 shows the steps taken by the Application Manager after the application is launched for the first time. The Application Manager checks to see if the application has a lock file, which it should not because this application should never be permitted to run (no lock file should ever be created). The Application Manager then locates and reads the policy file. If the application is not listed in the policy file, the application would then be terminated.

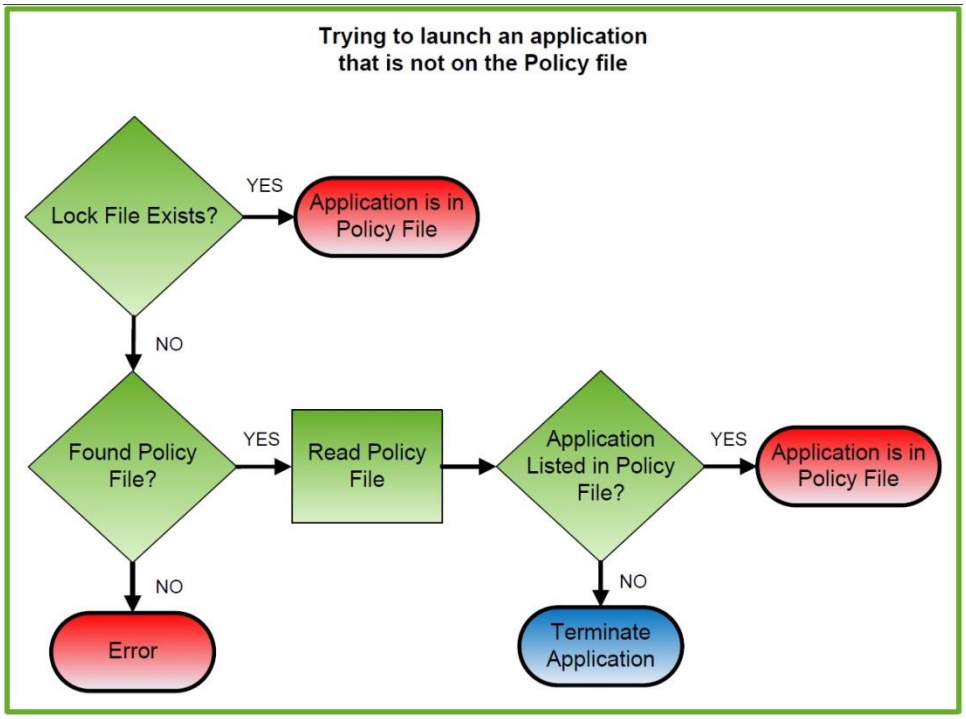

Figure 15: Launching a prohibited application

While performing this testing procedure, it was determined that the Application Manager was fully functioning. In other words, no problems were encountered during the testing procedure. Additionally, to ensure that the application was indeed working, the test was repeated a number of three times to ensure the same results were obtained.

4.3. DESIGN RENOVATIONS

After determining that the Application Manager was fully functioning, some design changes were made to the application. These changes included transition from Secure Digital (SD) card storage to internal storage, and a user interface for the application. These design renovations were implemented in two phases: modifying the existing code to function within internal storage and adding additional code for a user interface.

The first phase consisted of modifying the existing code to function within the internal storage of the device. This was done by modifying the code in three steps: modify the application verification method, modify the application termination method, and modify the downloading method. These steps were achieved by only modifying the code affecting where files were stored and loaded. After each step was completed, testing was completed in order to verify that the code was modified correctly at each step, and that the overall functionality of the Application Manager was not affected.

The second phase consisted of creating a user interface within the application. This user interface was completed in two steps: implement a dialog box and implement a toast notification. The reason for implementing a dialog box was to alert the user when a lock file was expired or when the application attempting to run is not on the policy file. Additionally, the reason for implementing a toast notification was to notify the user that a network connection was not detected in the case that a manual update for the policy file failed. After each step was completed, testing was done in order to verify that the overall functionality of the Application Manager was not affected.

4.4. PROBLEMS ENCOUNTERED

While implementing the design renovations to the Application Manager, two problems were encountered. The first problem was encountered after the code for the downloading method was modified to function within internal storage. Once the code was modified, coding errors were received indicating that the method used in order to pass variables between two Java classes (the class for application verification and termination, and the class for automatic download) was incorrect. In order to resolve this problem, a new function was created in order to

pass the variable between the two classes. Testing was then performed to verify that the automatic download process was indeed functioning correctly.

The second problem was encountered when implementing the code to display the toast notification for no network connection being detected. Once the code was written, an exception was thrown indicating that the code was not written correctly to implement the toast notification. In order to resolve this problem, a different class found within the Android SDK environment and Java libraries was used. Testing was then performed to verify that the toast notification did actually appear on the device when a network connection did not exist.

Once these design renovations were implemented, testing needed to be done to ensure the Application Manager was still functioning. This was done by making modifications to the checklist used for testing. The renovated testing procedure can be seen in Appendix A of this report.[1]

While performing the exhaustive testing procedure, a problem was encountered in which the Application Manager would stop executing its code if it was interrupted by the automatic download procedure. In other words, if the automatic download was initiated in the middle of the application verification procedure, then the Application Manager would freeze and the user would have to restart the process (launch the Application Manager again). This did not pose a problem in terms of security since the user would not be able to run any application; however, it did pose a problem in terms of the user's convenience. As a result, it was decided to make the automatic download process occur every day at midnight. This would ensure that the automatic download process would occur at a time that would not affect the other processes executed by the Application Manager.

[1] The procedure for the second scenario was not changed.

5. RESULTS

The final product developed for this project resulted in a fully functioning Android application titled the Application Manager. This application was created to ensure the security of a corporation through employee mobile devices. In order to ensure the Application Manager was as secure as possible, it was tested for a certain number of scenarios that it would encounter when someone within the corporation or a potential third party tried to "hack" the application.

The first scenario deals with a third party attempting to "listen in" or intercept the download procedure. In order to prevent the third party from either obtaining or replacing the files during the download, SSL was implemented, as previously mentioned. This prevents a third party from eavesdropping and ensures the integrity of the files during the download process using a private session key and hash checks. Additionally, the implementation of SSL prevents a third party from intercepting the download connection because of certificate authentication.

The second scenario deals with a third party attempting to replace the hosted files on the server. This scenario would be the least likely to occur since the third party would need administrative privileges in order to replace the files. This means that the person who manages the server would need to provide the third party with administrative privileges or that they themselves would need to be the hacker. Additionally, if the third party managed to get administrative privileges, they would still need to be able to determine how the original files were encrypted. If the new files are not encrypted in the same exact manner as the originals, the Application Manager will not be able to read the files, which would prevent all the applications on the mobile device from running.

The third scenario deals with a third party somehow managing to get a copy of the encryption key or policy files. If the third party had access to these files, they would not be able to manipulate them since they would still need to know how the files were encrypted. This prevents the third party from modifying the policy file to allow any applications to run on any mobile device.

The fourth scenario deals with the possibility of a third party cloning another mobile device that contains the Application Manager. In this scenario, the third party would copy all the content of one mobile device into another device or into an emulator. In the case that the third party managed to clone all the files for the Application Manager onto their physical or emulated device successfully, they would still be unable to access the files. The reason for this is that all the files used by the Application Manager are stored in the internal memory of mobile devices. This means that no Android application, third party, or user of the device, can access or view the files. In other words, the files used by the Application Manager are inaccessible to any application or person besides the Application Manager itself.

The fifth and final scenario deals with a third party rooting the mobile device containing the Application Manager. Rooting in terms of Android is when the user gains "root access" within the Android system, in which the user overcomes the limitations set in place by the manufacturer of the device. This means that the user would have access to the internal memory of the device, and would be able to obtain the necessary files in order to read and modify the policy file. Although the Application Manager is not protected against rooting as it currently stands, the use of a Trusted Platform Module (TPM) or the implementation of Security Enhanced (SE) Android would enable the Application Manager to become secure against root attacks. The use of TPM would allow all the files used by the Application Manager to be stored in secure

storage, which is separate from the operating system (Android). As a result, if the operating system were compromised by a root attack, the files used by the Application Manager would remain secure. Additionally, SE Android, which was developed by the US National Security Agency, has been tested to withstand root attacks [10]. Therefore, the implementation of SE Android within the Application Manager would ensure that it was not vulnerable to root attacks.

6. FUTURE WORK

Although an Android application and security architecture was successfully developed within this project for the future integration of hardware-enabled security technologies, many extensions could be implemented.

6.1. PUSH DOWNLOAD

The largest benefit to improving this project in relation to its server component would be to implement a push-based download system. This would reduce the need for unnecessary downloads and power consumption.

One possible method that can be used to implement the push-based system is to utilize a third party source, such as Urban Airship. This service would host the key and policy files and send push notifications to mobile devices each time the files were updated. An alternative way to implement the push-based system would be to make a server that allowed the creator to have administrative functionalities. This differs from the website hosted by WPI because the owner of the webpage does not contain administrative rights to the server. Both of these options would require registering with Google's Cloud Messaging for Android service and applying all relevant code provided by Google into the application. Google code is necessary to register the devices running the application with the push service and for notifying the device when files are updated.

The main benefit of implementing this download protocol is the reduction of power usage within the mobile device. This helps to minimize the impact of the Application Manager on the battery life of the user's mobile device. The currently implemented pull-based system is set up to download the hosted files according to a daily scheduled time. This affects the power consumption of the device because the files would be downloaded even if they were not updated.

The push-based system would allow for lower power consumption because the files would only download if they were updated.

The second benefit of implementing a push-based system is it provides an increase in security. This is due solely to the fact that there would be fewer chances for a third party to connect and interfere with the download because it would occur much less frequently than the pull-based system. Overall, the present pull-based system is functional for allowing the system to maintain up-to-date key and policy files, but a push-based service would improve the overall efficiency of the system.

6.2. ENCRYPTION

Another method of improving this project would be to upgrade the security used for file management. The most important aspect of the Application Manager is the security features it provides, otherwise its functionality would fail. In other words, if the Application Manager were not secure, it would not be able to ensure a corporation's security.

One feature that can be implemented in order to upgrade the security of the Application Manager is to change the type of encryption used from 128-bit AES encryption to 256-bit AES encryption. This change would allow all files encrypted with a 256-bit AES key to become harder to "crack." The reason for this is that a 256-bit AES key contains 1.1×10^{77} possible combinations, whereas a 128-bit AES key contains 3.4×10^{38} possible combinations. As a result, it is much harder to guess the contents of a 256-bit AES key than a 128-bit AES key.

In order to implement the change from using 128-bit AES encryption to 256-bit AES encryption, access to a specific Java library is required. Once the library is included in the project files for the Application Manager and the size of the encryption key is changed, all the files

utilized by the Application Manager would be encrypted with a 256-bit AES key. This upgrade in security was not implemented originally since access to sources outside of the native Android SDK environment was limited within the General Dynamics facility. As a result, the required Java library could not be obtained.

6.3. LOG PARSING

A method that would further improve the operation of this project would be to implement log parsing. This would lessen the Application Manager's dependency on other Android applications by allowing the Application Manager to detect and intercept the launch of third party applications without the need for hard code.

This feature could be implemented by analyzing the contents of the Android device's log file, which is a file that contains recorded data of all events that have previously occurred on the device. As previously mentioned, attempts at permitting the Application Manager to detect and intercept the launch of other applications failed. The temporary solution was to hard code the test applications to send a launch intent which allowed the Application Manager to launch any time the test application entered the ***onCreate()*** or ***onRestart()*** lifecycle states. Implementing log file parsing would remove the previously inserted hard code since the launch of all Android applications would be recorded within the log file. Log file parsing would also remove the hard code previously needed to obtain an Android application's package name, which is needed for application verification.

In order to implement log file parsing, a service would be created to constantly read the log file of the mobile device. When the launch of an application was detected, the service would then set a flag, which would launch the Application Manager. This would cause the application

to enter the ***onPause()*** lifecycle state, and allow the Application Manager to obtain the application's package name and verify it against the policy file.

Although this implementation removes the need for hard coding Android applications, it is not the ideal method for detecting and intercepting the launch of applications. The reason for this is log parsing would be implemented as a background service that would operate constantly and as a result, contribute to draining the battery on the mobile device. Although, the overall impact of this log parsing service is not expected to be detrimental to the device, it may inconvenience the user. Considering the fact that there exists constraints on how Android applications can interact with each other, specifically intercepting and terminating other applications, this is the best solution to remove the need for hard coding other than incorporating the Application Manager into the Android operating system.

6.4. TRUSTED PLATFORM MODULE (TPM)

The main benefit to improving this project in relation to its security component would be to integrate the use of a Trusted Platform Module (TPM). This would improve the overall security of the server, Application Manager, and mobile device by creating a trusted environment. One method of improving the security of the server could be to use the feature of remote attestation in order to validate to the server that the mobile device it is connecting to is secure.

In order to improve the security of the Application Manager, the encryption key and policy files could be stored within the TPM's sealed storage. The use of sealed storage would mean that the files would be encrypted and they would never leave the TPM, even during encryption/decryption. The TPM would also allow the integrity of the files to be verified. This is

done by comparing the values of the Platform Configuration Registers (PCR) to the values stored in the PCRs at the time of storage/encryption.

7. CONCLUSION

The overall goal of this Major Qualifying Project was to provide a foundation for the implementation of the Trusted Platform Module (TPM) onto mobile devices. This was accomplished by developing an Android application and security architecture that mimics the security features provided by the TPM.

The developed Android application served as a proof-of-concept implementation that it is possible to create a security architecture on the Android platform. This provided a secure method for corporations to regulate applications on employee devices by utilizing server security protocols, AES encryption, dual methods of application verification, and the device's internal memory. The security architecture ensured the security of the developed application, the Application Manager, and that it could easily be ported onto different platforms or operating systems. This allows the developed architecture to be applied to various hardware-based security technologies such as the Trusted Platform Module (TPM), the ARM TrustZone, and the Intel Trusted Execution Technology (TXT).

APPENDIX A

This appendix contains the renovated testing procedure.

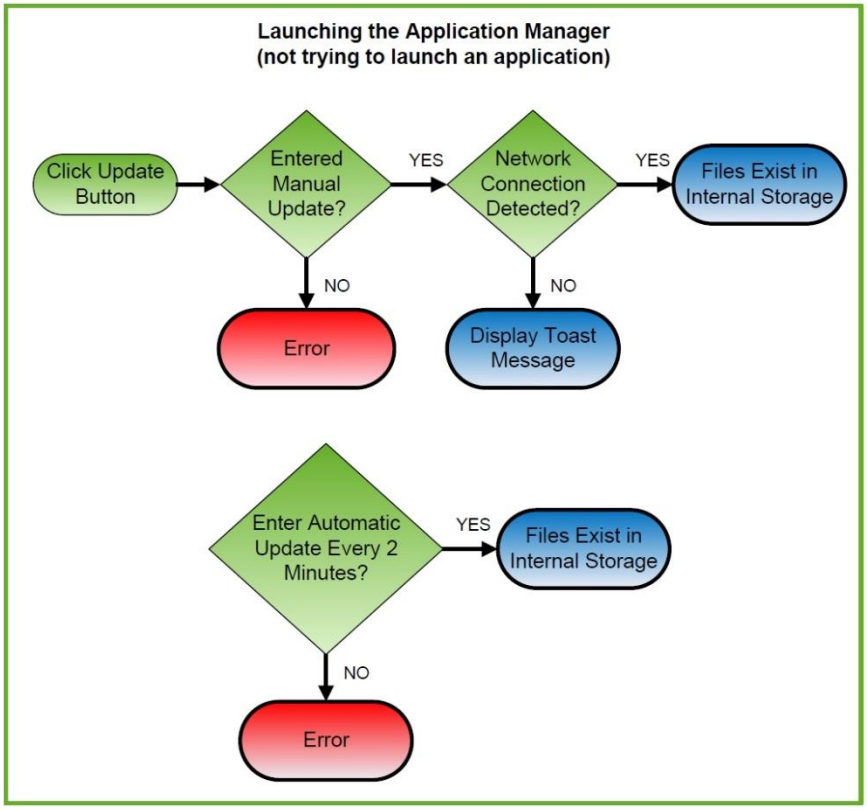

Figure 16: Launching the Application Manager – renovated scenario

For the renovated first scenario, there were two expected outcomes. The first was that when the user clicked the update button, the policy and key files would be downloaded and stored within the device's internal storage if a network connection was detected. If no connection was found, then the user would be notified that the download failed through a pop-up message (toast). The second expected outcome was that the policy and key files would be updated every two minutes. Figure 16 shows the process flow for the case in which the user tries to launch the Application Manager (Figure 12 shows this same scenario before design changes).

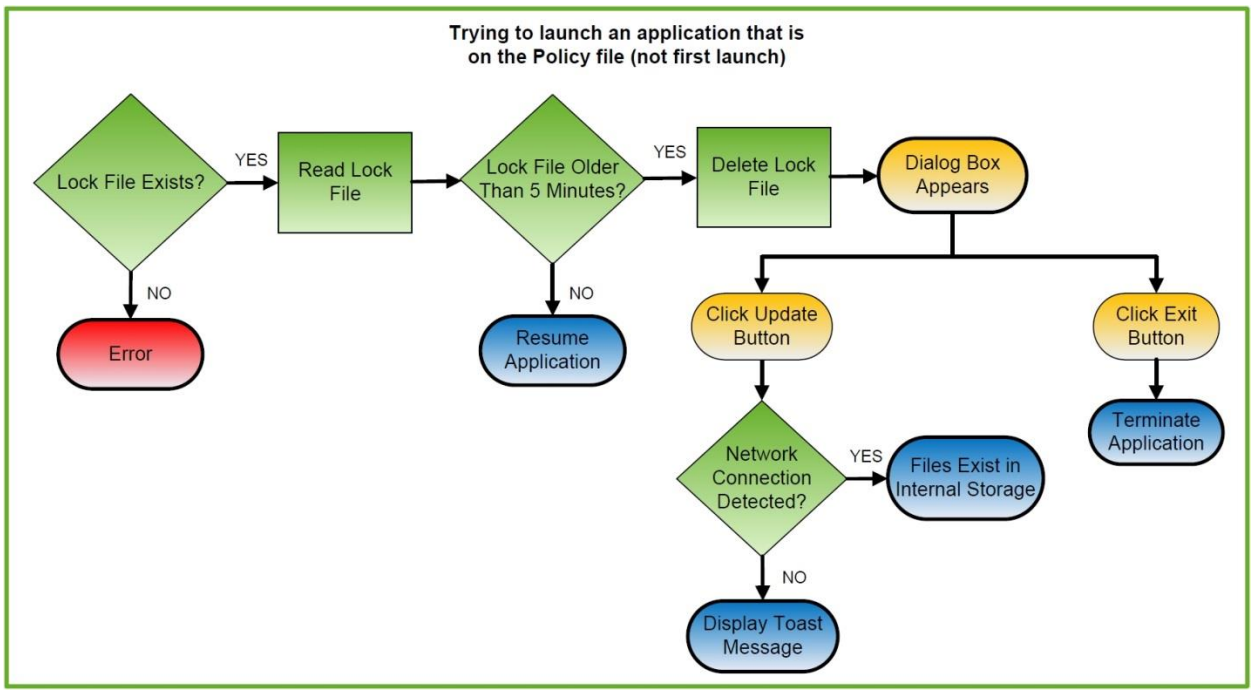

Figure 17: Launching a permitted application (not first launch) – renovated scenario

The renovated third scenario dealt with trying to launch an application that is on the policy file (not first launch). There were two possible outcomes for this situation, where the first is that the lock file is not expired and the application would be permitted to run. The second outcome was that the lock file had expired which would cause a dialog box to appear that would notify the user that the lock file for the current application has expired. This would provide the user with two options: update or exit. If the user clicked on the update button, the policy and key files would be downloaded and stored within the device's internal storage if a network connection was detected. However, if the user clicked on the exit button, then the application would be terminated and the user would be taken to their device's home screen. Figure 17 shows the steps taken by the Application Manager after an application is launched (Figure 14 shows this same scenario before design changes).

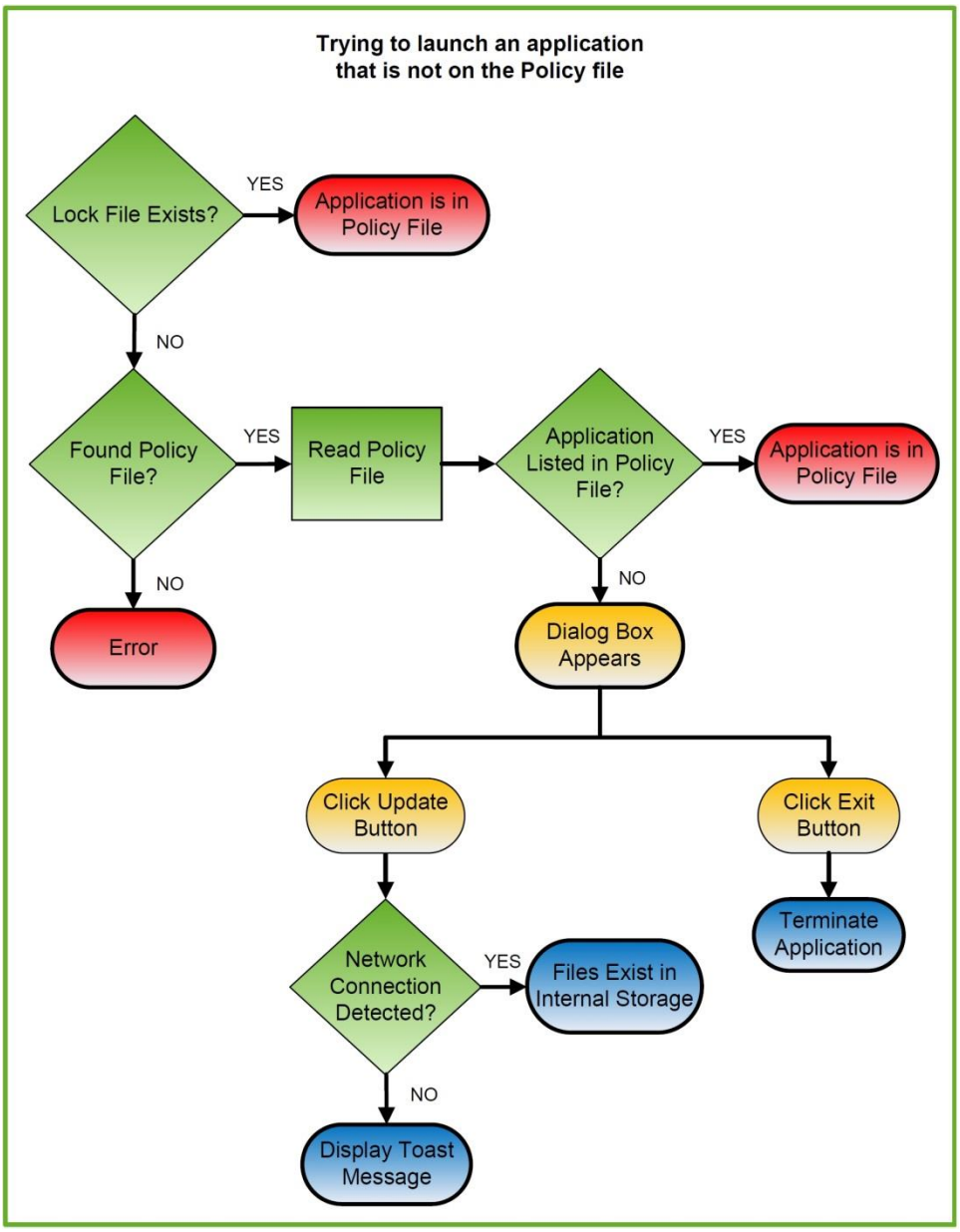

Figure 18: Launching a prohibited application – renovated scenario

The renovated fourth scenario contained one expected outcome: the application would not be listed in the policy file and as a result, the application would not be able to run. This would cause a dialog box to appear which would notify the user that the current application is not permitted to run. The user would then be provided with two options: update the policy and key files or exit the application and return to the device's home screen. Figure 18 shows the steps

taken by the Application Manager after a prohibited application is launched (Figure 15 shows this same scenario before design changes).

BIBLIOGRAPHY

[1] Trusted Platform Module (TPM). (n.d.).*Tablet PCs and Mobile Computing Solutions*. Retrieved April 13, 2013, from http://www.motioncomputing.com/choose/spec_TPM.htm

[2] Regenscheid, A. (n.d.). Roots of Trust in Mobile Devices. *Computer Security Division*. Retrieved April 13, 2013, from csrc.nist.gov/groups/SMA/ispab/documents/minutes/2012-02/feb1_mobility-roots-of-trust_regenscheid.pdf

[3] Trusted Platform Module (TPM) Specification Overview. (n.d.). *Study on the Impact of Trusted Computing on Identity and Identity Management*. Retrieved April 14, 2013, from www.fidis.net/resources/deliverables/hightechid/int-d37002/doc/9/

[4] Piolle, G. (2008, September 18). File:TPM english.svg. Trusted Platform Module. Retrieved April 14, 2013, from https://upload.wikimedia.org/wikipedia/commons/thumb/0/0b/TPM_english.svg/2000px-TPM_english.svg.png

[5] Schellekens, Dries. (2012, December). *Design and Analysis of Trusted Computing Platforms*. Retrived April 22, 2013, from https://www.cosic.esat.kuleuven.be/publications/thesis-219.pdf

[6] TrustZone - ARM. (n.d.). ARM - The Architecture For The Digital World. Retrieved April 14, 2013, from http://www.arm.com/products/processors/technologies/trustzone.php

[7] Software Architecture. (n.d.). TrustZone. Retrieved April 14, 2013, from www.arm.com/images/TrustZone_Software_Architecture.jpg

[8] Greene, J. Intel Trusted Execution Technology. (n.d.). White Paper - Intel Trusted Execution Technology. Retrieved April 15, 2013, from www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf

[9] Activities | Android Developers. (n.d.).Android Developers. Retrieved April 15, 2013, from http://developer.android.com/guide/components/activities.html

[10] Smalley, S. (n.d.). Security Enhanced (SE) Android. Build Security In. Retrieved April 14, 2013, from https://buildsecurityin.us-cert.gov/swa/presentations_032612/SE%20Android%20(Panel%20on%20Securing%20Mobile%20Operatins%20Systems)%20-%20Stephen%20Smalley.pdf