

Copyright
by
Ramiro Daniel Diaz
2015

**The Report Committee for Ramiro Daniel Diaz
Certifies that this is the approved version of the following report:**

S/MIME Client with Wireless Key Passing for Android

**APPROVED BY
SUPERVISING COMMITTEE:**

Supervisor:

Kathleen S. Barber

Co-Supervisor:

William Bard

S/MIME Client with Wireless Key Passing for Android

by

Ramiro Daniel Diaz, B.S.E.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2015

Abstract

S/MIME Client with Wireless Key Passing for Android

Ramiro Daniel Diaz, M.S.E.

The University of Texas at Austin, 2015

Supervisor: Kathleen S. Barber

As the value and vulnerability of personal data has increased, the security of digital communication and information has become more important. Currently, there exists no application that allows for a convenient transfer of secure digital communication on mobile devices as they continue to usurp the role of personal computers in the real world. This report will document my attempt to create one such application on the Android platform. This report will present a brief background of the algorithms intended to be used, as well as an overview of the evolution of the email protocols into the S/MIME protocol. It shall also cover related works that currently exist that we explored, or used in the creation of the application. In program requirements, the overall requirements of the application will be discussed in detail. In program design, we will describe the overall design and architecture of the application. Finally, program outcome will describe the difficulties faced in the process of implementing the work, the outcome itself and where it can be found, and future work that is needed for the application.

Table of Contents

List of Figures	ix
INTRODUCTION	1
BACKGROUND	3
Chapter 1: Encryption Algorithms.....	4
RSA	4
AES	5
SHA	7
Chapter 2: Email History	8
SMTP	8
MIME.....	9
S/MIME	9
Chapter 3: Used Components	11
X.509 Certificates	11
Bluetooth/BLE	11
RELATED WORKS	13
Chapter 4: Djigzo/Ciphermail.....	14
Pros	14
Cons	14
Reasoning.....	15
Chapter 5: Java Cryptography Architecture/ javax.crypto Package	16
Pros	16
Cons	16
Reasoning.....	16
Chapter 6: JavaMail API.....	17
Pros	17

Cons	17
Reasoning.....	17
Chapter 7: Javamail-android.....	18
Pros	18
Cons	18
Reasoning.....	18
Chapter 8: Bouncy Castle	20
Pros	20
Cons	20
Reasoning.....	20
Chapter 9: Spongy Castle.....	22
Pros	22
Cons	22
Reasoning.....	22
Chapter 10: Gmail API	23
Pros	23
Cons	23
Reasoning.....	23
PROGRAM REQUIREMENTS	24
Chapter 11: Environment.....	24
Hardware.....	24
Software	24
Connectivity.....	24
Prerequisites.....	24
Chapter 12: Functional Requirements	25
Email.....	25
Encryption.....	25
Contacts.....	25

Key Passing.....	25
User Interface.....	26
Chapter 13: Additional Requirements	27
Coupling.....	27
Source Code.....	27
Tests	27
PROGRAM DESIGN	28
Chapter 14: SMIME-CORE.....	29
Key Manager.....	29
Contact Manager	30
Crypto Manager	30
Mail Manager.....	31
Communication Manager.....	31
Chapter 15: App.....	34
User Interface.....	34
PROGRAM OUTCOME	36
Chapter 16: The S/MIME Client.....	37
Inbox	37
Read Mail.....	38
Compose Mail.....	42
Contacts.....	44
Chapter 17: Difficulties Faced	48
Dependency Issues.....	48
Spongy Castle	49
Missing Hardware	50
Additional 3 rd Party Issues	51
Chapter 18: Future Plans.....	52
Open Source Code	52

Javadoc.....	52
User Interface.....	52
Additional Account Types	53
Additional Encryption Algorithms	53
Automated Testing.....	53
Bug Fixes	53
Third Party Code.....	54
CONCLUSION	55
References.....	56

List of Figures

Figure 1:	Visualization of the AES encryption process	6
Figure 2:	Module breakdown of the S/MIME Client	28
Figure 3:	BLE Server/Client Flow	32
Figure 4:	Certificate Packet	33
Figure 5:	High Level view of the Application User Interface.	34
Figure 6:	Inbox view of SMIME Client.	38
Figure 7:	Read Screen of unencrypted email with headers and text part.	39
Figure 8:	Read Screen of unencrypted email with text part and HTML.	40
Figure 9:	Read Screen of encrypted email without proper certificate.	41
Figure 10:	Read Screen of encrypted email with proper certificate.	41
Figure 11:	Compose Screen in unencrypted mode.	42
Figure 12:	Secure contact selector in Compose screen.	43
Figure 13:	Encrypted mode Compose screen with contact selected.	44
Figure 14:	Contacts screen.	45
Figure 15:	View contacts popup.	46
Figure 16:	Select contact to send popup.	46
Figure 17:	Select Contact server peripheral.	47
Figure 18:	Server status page	47

INTRODUCTION

As the internet continues to grow, and mobile computing takes on a more ubiquitous role in the world, the amount of data being exchanged on a daily basis is booming and will only increase. The collection and selling of such information has become a very valuable industry in itself. However, as the value of personal information increases, so does the importance of data security.

Private information is continuously under attack by threats of all types. For example, in November 2014 tens of thousands of emails belonging to Sony were stolen by cybercriminals and published on the internet, causing several problems which hurt and embarrassed the company both financially and in reputation [1]. Threats to personal data, does not necessarily need to be from foreign entities. In June of 2013, former NSA contractor Edward Snowden leaked classified documents revealing the existence of the PRISM system [2]. PRISM gave the NSA the ability to collect private communication of targeted individuals from many of the country's top internet and telecommunication companies [3]. Emails, were among the communication types that were likely to be intercepted.

Although, in competition with mobile instant messaging and text messaging, email remains the predominant leader in online communication with over 190 Billion emails sent per day [4]. For one wanting to increase data security, it would only be natural to attempt to integrate it with email first. Luckily, there exists a security protocol for this already, S/MIME, which will be described in more detail later.

The S/MIME protocol enjoys widespread use in the business sector, and is supported by most of the online services already. However, the protocol is not used nearly as widely in the public sector [5]. While there may be several factors for this, I

believe that convenience is a significant one. The private sector is increasingly moving away from their desks, and is preferring to use their mobile devices. As of January 2014, more Americans used a mobile device to access the internet as opposed to a personal computer [6]. In order for S/MIME to increase adoption in the private sector, it is in its best interest to have a mobile application that supports it. Some gateway applications do exist, however, none integrate the receipt, decryption, and display of the end message. Neither do they allow for conveniently passing public credentials to a new party. The S/MIME Client for Android is intended to fill that void.

The S/MIME Client for Android will be an email application that allows sending and receiving encrypted S/MIME emails as well as the ability to share public credentials to another individual using the app. This application will use open source software, and will thus be open source itself. This report, will describe the background, requirements, and design of the application and will see the end product.

BACKGROUND

To further understand the decisions made in this report, it is necessary to understand some background of the components used. A brief description of the encryption/digest algorithms, history of email protocols, and description of other components used follows.

Chapter 1: Encryption Algorithms

At the core of data security is encryption. The ability to share data with and only with its intended recipient is necessary for communications to remain secure. There are several algorithms that have existed throughout time with varying levels of security. From the Caesar Cipher of the time of Julius Caesar to the relatively new Elliptic Curve Cryptographic ciphers, it is important to know about the methods of encryption that are being used. This chapter will describe the algorithms used in the S/MIME client, namely RSA, AES, and SHA-512.

RSA

RSA is a Public Key (asymmetric) encryption algorithm developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman at MIT. It was first published in 1978 and has since been the most widely accepted and implemented approach to public key encryption. [7]. RSA's security depends entirely on the difficulty of factoring large numbers which is found to be NP Hard. Put simply, the algorithm is as follows [8]:

- 1) Select a number, n , that is the product of two prime numbers p , and q .
- 2) A number e will be a number that is relatively prime to $(p-1)(q-1)$
- 3) The private key, d , can be found by $d = e^{-1} \text{mod}((p-1)(q-1))$
- 4) To encrypt: The cypher text, c , can be found by $c = m^e \text{mod} n$ where m is the message to be encrypted.
- 5) To decrypt: Message, m , can be found by $m = c^d \text{mod} n$

The security of RSA increases with the key size, however larger key size will bring about a performance penalty. Factorizations of 512, and 768-bit keys have occurred so those key sizes are deemed insecure [7], it also is considered wise to begin to phase out the 1024-bit key size as it may be factored within the next decade. In this project we

use a 2048-bit key size for RSA. In Android, use of RSA is limited to only be able to encrypt a message up to $(k/8 - 11)$ bytes where k is the key size, for the 2048-bit key size this would mean a max message size of 245 bytes [9]. This means that large messages need to be encrypted by using another algorithm, preferably a fast symmetric cipher. In our case, we will be using RSA to encrypt the shared symmetric key which encrypts the large messages.

AES

The National Institute of Standards and Technology began the search for the replacement of the Data Encryption Standard (DES) on January 2, 1997 after it was found that DES could be cracked using specialized machines. The replacement needed to have 128-bit block sizes with supported key sizes of 128, 192, and 256 bits. Also the standard needed to be available worldwide on a royalty free basis. The search yielded the Rijndael algorithm. Invented by the Belgian researchers Vincent Rijmen and Joan Daemen, Rijndael was evaluated based on security, cost, and algorithm and implementation characteristics and found to be superior to its rivals thus enabling it to be chosen as the Advanced Encryption Standard or AES in 2001 [10].

AES is comprised of nine, 4-stage rounds and one 3-stage round. The four stages used are as follows [7]:

- Substitute bytes – A byte by byte substitution of the block using a predefined S-box.
- ShiftRows – A permutation of the rows
- MixColumns – A substitution using the finite field $GF(2^8)$
- AddRoundKey – A bitwise XOR of the block with the expanded key.

Every step is easily inverted mathematically, enabling the standard to be used in one direction for encryption, and in the other direction for decryption. The expanded key previously mentioned is produced when the 128-bit key is expanded to 44 32-bit words of which for are used in each round. Figure one describes the AES encryption process.

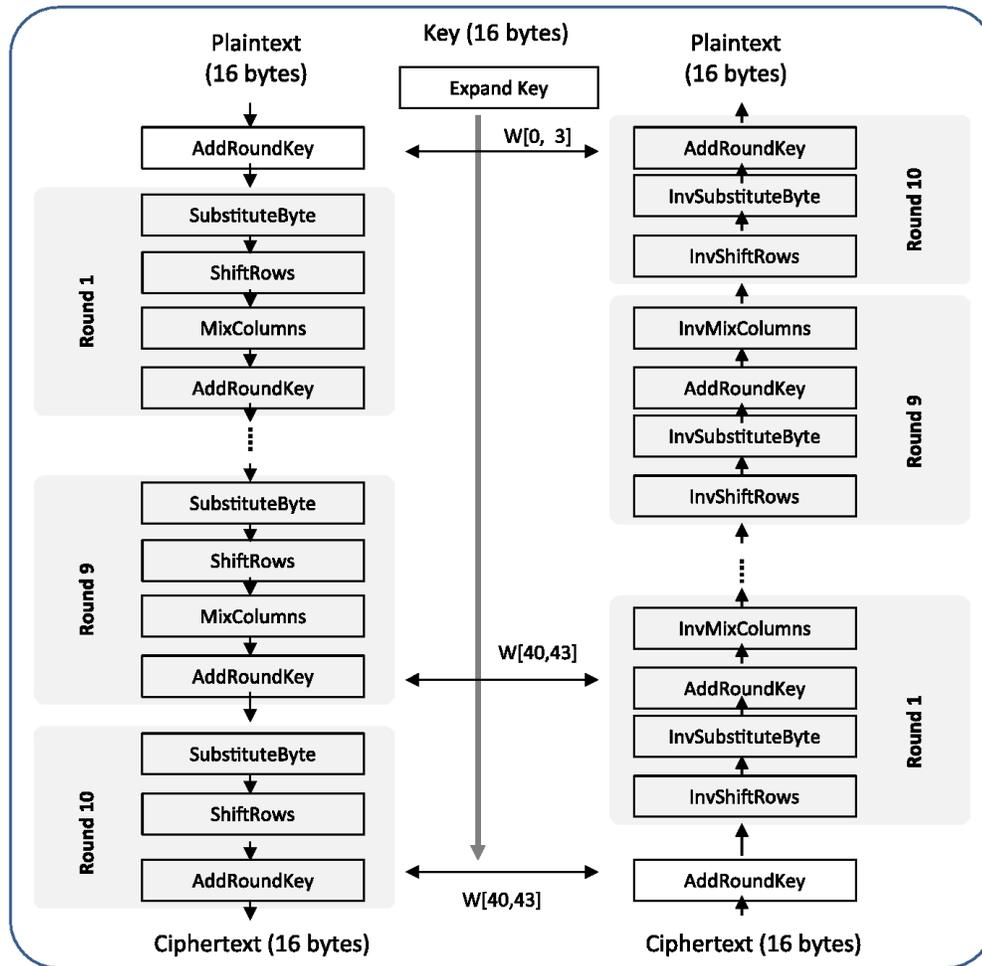


Figure 1: Visualization of the AES encryption process [11]. The left flow describes the process of encrypting plaintext. The right flow describes decrypting ciphertext.

SHA

The Secure Hash Algorithm (SHA) is the most widely used hash function, primarily because every other hash function has been found to have cryptanalytic weaknesses. SHA was developed by NIST and the National Security Agency (NSA) and published in 1993. The original SHA algorithm (SHA-0) was found to have weaknesses and replaced by SHA-1 in 1995. Once again in 2002, NIST found that the probability for SHA-1 to produce messages with the same hash was greater than expected, prompting it to create three new standards, collectively known as SHA2. SHA-2 consists of three versions of SHA which produce hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA384, and SHA-512 respectively. For this project we will be using SHA-512.

SHA-512 takes an input message of at most 2^{128} bits and produces a hash of 512 bits. The message is padded so that it can be broken up into 1024 bit blocks after having the original message length appended to it as a 128-bit number. The message is then put through 80 rounds of processing to produce a 512-bit value that can be used for signatures or testing message uniqueness. Unlike encryption functions, hash functions are intended to be one way and irreversible, meaning that you cannot derive the original message from its hash.

Chapter 2: Email History

Email is the most prevalent form of communication used today, with over 190 Billion emails sent per day. When choosing to implement the S/MIME client, some investigation and knowledge of how the underlying protocols of emails have evolved is necessary. This chapter will attempt to shine a light on the stages of email's evolution leading to the S/MIME protocol.

SMTP

The Simple Mail Transfer Protocol (SMTP) was first defined by RFC 821 in 1982. Today, it is the most widely used mail transfer protocol. SMTP is part of the application layer in the TCP/IP protocol and transfers messages using the store-and-forward process to travel across networks [12]. SMTP is a mail transfer protocol only, meaning that it is used to send mail to its respective server, not to the intended individual. Individual servers themselves implement methods to deliver the message to the individual via other protocols such as the Post Office Protocol (POP3) or the Internet Message Access Protocol (IMAP). An SMTP session between an SMTP client and server usually consist of three command and response sequences:

- 1) MAIL – Establishes the return address of the messages in case of error.
- 2) RCPT – Establishes all of the recipients of the messages.
- 3) DATA – Confirms that the receiving server is ready to accept the message data, and signals whether or not to accept or reject a message once received.

In the earliest iteration of SMTP, the protocol was purely ASCII text based. The protocol was not able to handle binary data, or even international language character sets. To remedy this, Multipurpose Internet Mail Extensions were developed which are explained in the next section. Email addresses are still ASCII text only [13].

MIME

Multipurpose Internet Mail Extensions (MIME) was created by Nathaniel Borenstein and Ned Freed and drafted by the Internet Engineering Task Force (IETF) as a standard in 1993 [14]. MIME was intended to address some of the problems and limitations of SMTP in compliance to the Internet Message format defined in RFC5322 [7], [15]. The most important limitations of SMTP that MIME addresses are its inability to transmit binary objects and the inability to use 8-bit international characters instead of 7-bit ASCII.

MIME adds five new message header fields providing information regarding the body of the message. Among these fields is Content type, also known as MIME type. MIME types are widely used to define standardized ways to handle a wide variety of multimedia information. Additionally, MIME also allows the ability for emails to have multiple parts. Most emails that are encountered today are MIME emails.

S/MIME

Secure/Multipurpose Internet Mail Extension (S/MIME) is a security standard developed in 1995 for encrypting and signing MIME data. When S/MIME 1 was released, there was no single standard for secure messaging, and had to compete with several other standards. In 1998, S/MIME version 2 was submitted to IETF for consideration as a standard, and has since become it [16]. S/MIME provides two main abilities to MIME, message encryption and message signing.

Message encryption in S/MIME is achieved by using both symmetric and asymmetric ciphers. Typically, the desired message is encrypted using a shared secret key and an algorithm such as AES. The secret key itself is encrypted using an asymmetric

cipher such as RSA and the recipient's public key. When the recipient receives the email message, they decrypt the shared key using their private key, and then decrypts the message using the shared key.

Message signing is achieved by taking a hash of a message using an algorithm such as SHA-256. The hash is then signed by being encrypted by the sender's private key. That signature is attached to the message and sent. When the recipient receives the message, they use the sender's public key to unencrypt the hash. They then take a hash of the message themselves and compare it to the hash received. If the hashes match, then the receiver can be assured that the message was not altered in transit.

S/MIME is supported by most servers that support MIME, however its requirement to use certificates and certificate authorities serves as a bar from the general populace from adopting it. For the purposes of this project, we will be using self – signed certificates to pass keys. However, it is advised that for really secure transactions, true trusted certificate authorities be used.

Chapter 3: Used Components

While not specifically encryption algorithms or tied directly to emails, the following components are necessary for the S/MIME client to be realized.

X.509 CERTIFICATES

In S/MIME the public encryption keys are transferred via X.509 certificates. X.509 certificates were created to avoid Man-in-the middle attacks where a third party, or a “middle man” has both the recipient’s and the sender’s public key. In this situation, the middle man will claim to both parties that they are the intended recipient and sender. The parties would falsely believe they are securely sending each other messages, when in fact they are sending it to the middle man who is decrypting the messages, reading or altering it, then re-encrypting it with their own key and sending it to its intended recipient.

X.509 certificates protect against man in the middle attacks by adding a trusted certificate authority to the mix. In an X.509 certificate, along with the public keys of the sender or the recipient, is the same key signed by the trusted certificate authority. The recipient can at any time compare the key with the signed key by using the certificate authority’s public key to verify that the key has not been tampered with. In S/MIME all keys are sent by way of X.509 certificates.

BLUETOOTH/BLE

One of the main points of this application involves the ability to wirelessly transmit a certificate to someone you wish to engage in secure conversations with. With the exception of wireless internet and telephone, Bluetooth is perhaps the most ubiquitous wireless standard today. Bluetooth version 1 was introduced in 1999 and made it onto its first cellular phone one year later. Bluetooth has many different types: Bluetooth classic, Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR), Bluetooth High Speed (HS), and

the type we will be using Bluetooth Low Energy (BLE). BLE is becoming the de-facto standard for the Internet of Things due to its ubiquity and power efficiency. BLE consist of a server and a peripheral, where the server is the device providing data and the peripheral is consuming the data. In the case of the S/MIME client, the application will have to serve as both, depending on whether they are sending or receiving the certificate. The ability for a phone to become a server and advertise their existence in BLE is a relatively new addition to Android, implemented starting with version 21 of the operating system. As such, not all phones have the hardware capable of advertising over BLE.

RELATED WORKS

When embarking on any new software program, it is important to survey all of the readily available tools which are related. Researching related works ultimately reduces development time by allowing you to not have to reinvent the wheel every time. As the S/MIME client will ultimately be open source, the use of other open source software will allow me to skip development of core features that have already been developed by the software. Additionally, software quality should increase since core features in related works have already been scrutinized and tested by other eyes. This additional scrutiny is important for cryptography, as it will continually test the algorithm's compliance with Kerckhoff's principle. To Paraphrase, Kerckhoff's principle states that the security of an algorithm should not be based on the attacker's knowledge of the algorithm but on the mathematics of the algorithm itself. An attacker should not gain any advantage from knowing the algorithm that is being used. Open source software allows many people to attempt to find vulnerabilities in the implementation and report them. Software that has been around longer without being cracked can be generally thought of as more secure.

Chapter 4: Djigzo/Ciphermail

Ciphermail is an S/MIME gateway which allows for sending s/MIME emails from an Android phone. It has strong support for S/MIME and is available via the android marketplace. Its origins can be traced back to when it was the open source project Djigzo. It is still being updated, however it no longer is open sourced for the Android platform.

PROS

Ciphermail provides a solid library that has been in use for a few years. As such it can be a trusted one. It supports multiple servers and accounts and allows for the import of certificate via email with password protected files [17].

CONS

The largest detractor for Ciphermail is that its android client is no longer open source, however it does provide packages for other distributions [18]. As mentioned above, any cryptographic application can only be more secure and trusted if made open source. It is also important to note that the open source packages provided for other distributions are not well documented online. Additionally, Ciphermail does not receive emails, making the average user have to download two applications in order to read emails. This is a major inconvenience. Finally, at the time of initial development of the S/MIME application, Ciphermail did not have support for sending anything other than text. This addition is recent, and while too late for the initial draft of the S/MIME client, it would merit additional consideration for future versions.

REASONING

The pros of using Ciphermail in the creation of the S/MIME client were incredibly strong, however the cons of not being open sourced, and at the time not having support for extensions other than text ultimately disqualified it. However, with the recent updates adding support for extensions it will deserve a fresh look for integrations in future versions of the S/MIME client.

Chapter 5: Java Cryptography Architecture/ javax.crypto Package

The Java Cryptography Architecture (JCA) is bundled within the Java JDK for use with applications. It provides the classes, interfaces and APIs for doing cryptography on java [19]. Its functionality is delivered via the javax.crypto package [20] and has an extendable implementation through the use of providers.

PROS

Since Android is based largely on Java, it is great to find an architecture that was already embedded in the language. JCA and the javax.crypto package are very well documented and widely exercised. The additional use of providers proved to be incredibly useful for the S/MIME application.

CONS

As a package, javax.crypto did everything it was supposed to do. However, It was missing functionality needed for the S/MIME project, which meant we would have to search out additional providers to supplement the package. The architecture was designed to support this, however, so it caused no issue.

REASONING

Being that the JCA was already built in to the Android platform, was widely used, and greatly documented; its inclusion into the S/MIME client was a given. The cons, simply meant we had to search for additional providers to add to the package.

Chapter 6: JavaMail API

The JavaMail API, is the open source implementation adding support for creating a mail system in java [21]. This support includes MIME which is at the core of S/MIME [22].

PROS

The JavaMail API is open sourced and widely used adding support for MIME emails in java. It is very well documented and available via Maven Repositories, that are easy to add to Android projects.

CONS

At the time of development of the S/MIME Client, it was found that the JavaMail APIs were not fully supported on Android. During the parsing MIME email attachments, a runtime exception occurred for missing libraries that were referenced in the API. It happens that Android contained a cut down version of the javax.awt package that was missing the required classes.

REASONING

The JavaMail API, had everything we wanted out of a mail package on paper. It was open sourced, well documented, and well supported. Unfortunately, the dependency issue made it unusable at the time, prompting a search for a package that could resolve the error. It is also important to note, that the newest development release, now claims support for Android [23]. This is a change that is desirable for future releases of the S/MIME client.

Chapter 7: Javamail-android

Upon hitting dependency issues with the built in javax packages where classes within javax.awt were referred to but missing. It was necessary to find alternatives to the JavaMail API that fixed the breakage. This is the role javamail-android filled. Javamail-android is an open sourced package whose goal is to provide a 100% functional lightweight JavaMail port for android [24].

PROS

This package is open sourced and achieved its goal in fixing the dependency issues resident in the JavaMail API on Android for the time. Documentation was lacking, however, upon inspection of the code, it was found that it simply contains the missing classes in javax.awt and modifies the dependent classes to point to these new classes. So, the JavaMail API documentation would also work for these classes.

CONS

As mentioned before, documentation was lacking, however the JavaMail documentation worked for this package. Unfortunately, this project was also quite old and was now archived on the google code site it resided on. I encountered an issue where Android was not referring to the libraries in this package, but to the old one of its own. The fix to this required me to have to export the repository and rename the package. Adding the headache of needing to repackage the entire project to use the Android studio build system, as this was built on an antiquated Eclipse IDE.

REASONING

This package, did not have too much going for it except that it solved a problem that was a functionality blocker. If given a choice, which I now have for future releases, I

would not have used this package. Unfortunately, at the time of development this was the only option to use. Sun setting this library for the new JavaMail API for Android is high priority for the next release of the S/MIME Client.

Chapter 8: Bouncy Castle

While the JavaMail API, and javamail-android packages provided MIME functionality for the Client, it still lacked the ability to create S/MIME emails. Bouncy Castle is a provider to the javax.crypto package that adds S/MIME functionality by using the JavaMail APIs [25].

PROS

Bouncy Castle is open sourced, and widely used by many programs wanting to use S/MIME for encryption and signing of messages. Documentation on Bouncy Castle is easy to find and within its code base provides many examples of its proper use. Its packages are readily available online, and is to some extent built into the Android code base.

CONS

Bouncy Castle is an example of a project that is perhaps too well documented. When searching its documentation online, it is easy to find documentation. Unfortunately, the documentation throughout all of its versions are spread throughout the internet. It is difficult to find out whether the examples are of the most current version. Additionally, it appears that although Android contains Bouncy Castle in its code base, it is also a cut down version of it. This caused issues when parsing S/MIME emails.

REASONING

Bouncy Castle is possibly the most widely used provider for providing S/MIME functionality for Java. However, for Android it is missing functionality. Android's behavior favoring internal libraries to external ones when the package name is the same

prompted a search for a provider that simply changes the package names of Bouncy Castle.

Chapter 9: Spongy Castle

Spongy Castle is a repackaging of Bouncy Castle for Android. Simply, it exists to solve the cons of the bouncy castle package [26].

PROS

Spongy Castle is open sourced, and like the javamail-android package, can use the Bouncy Castle documentation for development. Its main purpose is to cause Android to select the full version of Bouncy Castle instead of the gutted internal one.

CONS

Unfortunately, it appears that the maintainer of SpongyCastle ran into the very same dependency issues when porting in Android, that they dropped support for the mail package. This means that I had to clone the repository and add a branch which added in the javamail android packages to make it work. While documentation for Bouncy Castle gave a general idea of how to use Spongy Castle, real examples were only really achieved through searching through tests in the source code.

REASONING

As with javamail-android SpongyCastle was not perfect, however it was one of the few solutions that existed that did not require me to fully rewrite the mail subsystem for Android. While I intend to upload my branch into the Spongy Castle source code, I think testing the code changes with the new JavaMail API for android would be important for the near future.

Chapter 10: Gmail API

Gmail is one of the most used email services in the world [27]. Android, also developed by Google, is designed to work in the ecosystem started by Gmail. For easier sending and receiving of emails, Google has a set of Gmail APIs for developer use [28].

PROS

Gmail and the Gmail APIs are well documented and remove the barrier of authentication by allowing the application to use system account calls. Questions regarding the Gmail API are handled via stackoverflow, a website commonly used by developers to help each other through issues.

CONS

Using the Gmail APIs will limit the accounts used with the S/MIME Client to be restricted to be solely for Gmail. While very good documentation exists for the Gmail API [29] it is not truly open sourced.

REASONING

As the focus of this project is to add an Android application for mail, it does not seem like too much of a detractor to use Gmail accounts only for an initial release. This is due to the fact that on downloading items from the playstore on Android, typically requires a Google Gmail account already. It can be inferred that most people using this application will already have a Gmail account. Additional, accounts will be a topic for future enhancements to the S/MIME Client.

PROGRAM REQUIREMENTS

This section will describe the requirements that should be fulfilled for the S/MIME Client.

Chapter 11: Environment

Before evaluation of the functional requirements, we should focus on what the scope of the application should be, and what environment do we expect our users to be using.

HARDWARE

The S/MIME client should be run on mobile devices that have hardware capable of accessing the internet, and hardware that can act as a Bluetooth server for sending contacts, and as a Bluetooth Peripheral for receiving contacts.

SOFTWARE

The mobile device should be running on Android SDK version 21 (Lollipop) or above.

CONNECTIVITY

The mobile device should be internet capable with broadband speeds. Internet Connection and Bluetooth functionality should be enabled.

PREREQUISITES

Users should have their devices activated and install the S/MIME client on them. Users should have a Gmail account, and have it as the user account on their device. In addition, they should allow the application to have Bluetooth, email, and internet access permissions on the device.

Chapter 12: Functional Requirements

The following is a breakdown of the functional requirements of the application itself, they are separated into related groups.

EMAIL

- Application should be able to send an encrypted email.
- Application should be able to send an unencrypted email.
- Application should be able to read an encrypted email.
- Application should be able to read an unencrypted email.

ENCRYPTION

- Application should be able to encrypt an email using a shared key.
- Application should be able to decrypt an email using a shared key.
- Application should be able to encrypt a shared key using a recipient's public key.
- Application should be able to decrypt a shared key using the client's private key.
- Application should be able to sign an email using the client's private key.
- Application should be able to verify a signed email using the sender's public key.

CONTACTS

- Application should be able to add a contact's certificate.
- Application should be able to remove a contact's certificate.
- Application should be able to store a contact list.
- Application should be able to clear or remove a contact list.

KEY PASSING

- Application should be able to transmit the client's public key wirelessly.
- Application should be able to receive another client's public key wirelessly.

- Application should be able to create a client's public and private key.

USER INTERFACE

- Application should be able to display email headers in an inbox.
- Application should be able to compose an email to be encrypted and sent.
- Application should be able to compose an email to be sent unencrypted.
- Application should be able to display an icon indicating an email was received encrypted.
- Application should be able to display an icon indicating an email was received unencrypted.
- Application should allow the user to see contact's for whom they have certificates.
- Application should allow the user to select users with certificates to whom to send encrypted emails to.

Chapter 13: Additional Requirements

Additional requirements not directly tied into the coding will be described here. This includes considerations on the design of the code, and distribution of the code.

COUPLING

The logic for the base functionality of the S/MIME client should not be highly coupled to the logic of the User Interface. This will increase the reusability of the code in other applications and help the application's maintainability.

SOURCE CODE

The source code should be open source and be available publicly via an online source code repository. This will enable it to be used, and further scrutinized in order to improve quality.

TESTS

The application should have some form of testing framework and have test cases exercising the base functionality logic. The UI is not mandatory in this testing.

PROGRAM DESIGN

In order to make code reusable and less coupled, the S/MIME client consists of two modules. The App and the SMIME-CORE. The figure below describes how the modules are broken up. We shall go into these sections in more detail in this section.

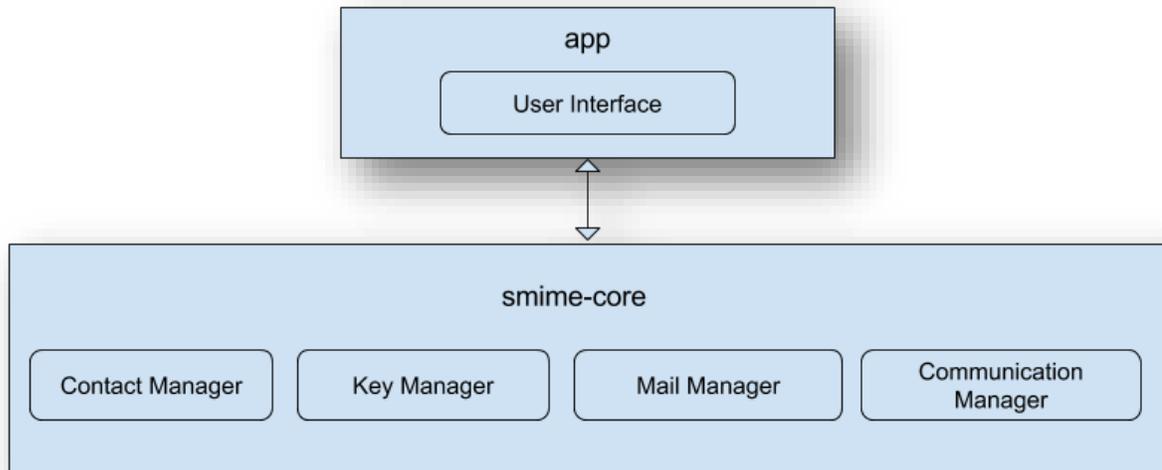


Figure 2: Module breakdown of the S/MIME Client. The app and smime-core are two separate submodules each with their own components. The design is intended to allow for the core to be used without the app.

Chapter 14: SMIME-CORE

The SMIME-CORE is a submodule of the S/MIME Client application, it is designed to be used as a library to enable third parties to use it without being tied to the User Interface. The SMIME-CORE is comprised of several managers that both mirror the requirements groupings and provide for separation of concerns.

KEY MANAGER

The Key Manager is responsible for the generation of symmetric keys, and asymmetric key pairs. Additionally, the key manager is responsible for the storing of keys in the key store. Not all keys are stored in the key store, only those belonging to the user.

The key store used in the application is the Bouncy Castle “BKS” key store. The original intention was to use the native Android key store for the client, however there was one caveat. The Android key store does not allow the modulus information about the keys within to be re-extracted. This was a barrier to our ability to encrypt and parse S/MIME emails using the Spongy Castle Libraries. The “BKS” key store provides all of the same functionality and security of the Android key store, but removes that restriction.

The key generators used in the Key Manager are initialized as 2048-bit keys for asymmetric RSA key pairs. The key size for the symmetric AES shared key is 256-bit. For AES, 256-bits is the largest supported key size on Android. For RSA, the 2048-bit key size attempts to balance performance and security. The largest RSA key size supported in Android is 4096-bits. The key generators are provided by the Spongy Castle libraries.

CONTACT MANAGER

The Contact Manager is responsible for almost all activities regarding X.509 Certificates. It can create self-signed certificates, as well as trusted certificates, and it can produce certificate chains for a user. However, where it gets its name from is its responsibility to manage the various certificates in the application's key store. Each certificate corresponds to an individual's public key, and thus is implicitly a list of contacts that are eligible for secure communication. Due to the nature of Certificate creation, it will almost always be necessary for a user to interact with the Key Manager first in order to generate the key pairs necessary for the certificate.

CRYPTO MANAGER

The Crypto Manager is responsible for all activities involving encryption. With the Crypto Manager a user can encrypt and decrypt data using either the RSA or the AES algorithms. The Crypto Manager also converts MIME Body Parts into S/MIME body parts, as well as providing the signature capability. Typically, either a certificate or a key must be obtained before you are able to use the functionality of the Crypto Manager.

The encryption algorithms are implemented using the javax.crypto library. They have been designed in such a way that you can use the encryption and decryption methods with any supported encryption algorithm. On the other hand, the S/MIME encryption and signing routines are currently restricted to being SHA-512 with RSA encryption for signing. For encrypted S/MIME, the symmetric key algorithm is AES-256 with Cipher Block Chaining. While the asymmetric key encryption is RSA.

The S/MIME body parts are being converted using the Spongy Castle libraries.

MAIL MANAGER

The Mail Manager is the largest manager in the SMIME-CORE, and is perhaps the most useful. The Mail Manager handles the managing the Gmail credentials and constructs, deconstructs, and submits MIME messages so that they are in a format compliant to the Gmail APIs. Additionally, the mail manager also handles retrieving attachments from the Gmail server.

The Mail Manager has many asynchronous methods, this allows the user to not have to implement threading on their own. In Android, it is not allowed for a user to run any commands interacting with the network on the main, or UI, thread. The reasoning for this is due to performance of the UI. When these asynchronous methods complete, they call callback methods to communicate their results.

COMMUNICATION MANAGER

The Communication Manager is responsible for device to device wireless communication. In our cause, it is responsible for the Bluetooth Low Energy (BLE) transmission and receipt of X509 Certificates. The Communication manager is designed to abstract out the protocol specific functionality so that it can be extendable in the future. However, for the meantime only BLE is supported.

The Communication Manager takes in a Server and a Client. The server has the functionality necessary to be able to transmit a contact to another device. In BLE, this means that the communication manager has to create a Gatt server. The Gatt server will then have a special service created that will transmit the desired X509 Certificate. This service will be assigned a unique identifier that will be advertised when the server is started. After the server is started, it waits for connection from the client. Once connection is established the client must query the server for the data. After the data has

been transmitted, the server is stopped. The flows for sending and receiving a contact can be found in the following figure.

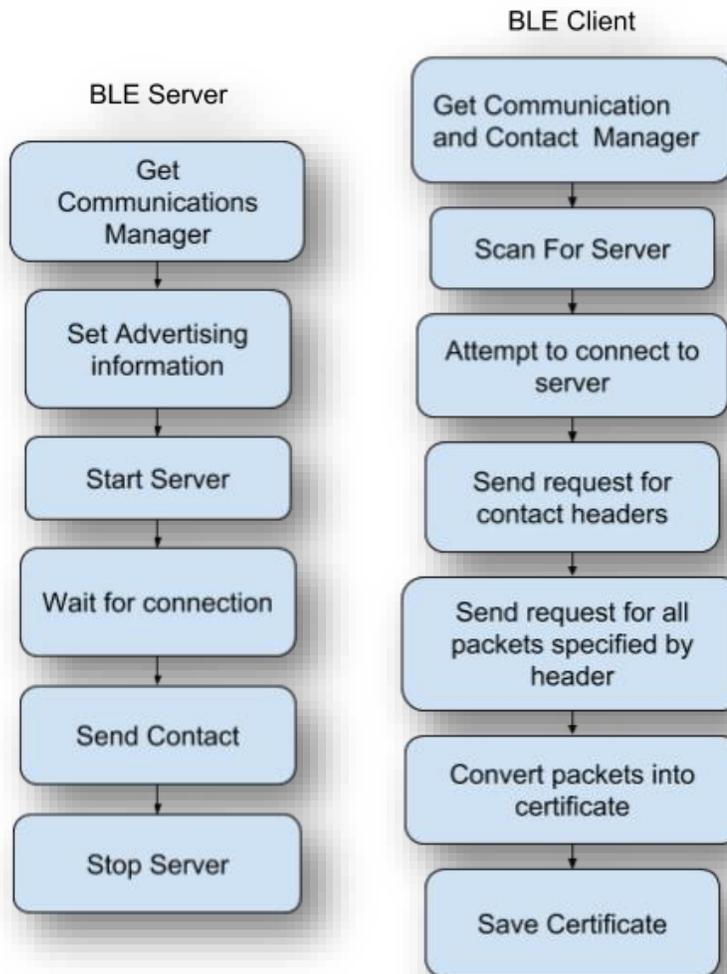


Figure 3: Left: The Flow for BLE Server Transmitting contact. Right: The Flow for BLE Client receiving contact.

There is a limitation on data transmission for the BLE protocol. There can only be one transmission sent per data query, and every transmission packet can only be up to 20 bytes long. Since X.509 Certificates are significantly larger than 20 bytes, the application has a protocol for the transmission and receipt of a certificate. When the BLE client connects to a BLE server, the client must send a query with a 2 byte short integer. This will be referred to as the index. The index corresponds to the packet number of the broken up certificate with the exception of two special values, 0xFFFF, and 0xFFFE. When the server receives an index of 0xFFFF, it will respond with a header packet which contains the number of packets, that the certificate has been broken into, and the total size of the certificate. When the server receives an index of 0xFFFE, then it will disconnect and stop the server. All indexes in between will respond with a data packet which contains a 2 byte index and 18 bytes of data. This protocol will allow for the client to query the server and obtain any lost packets. The figure below shows how the packets are broken up.

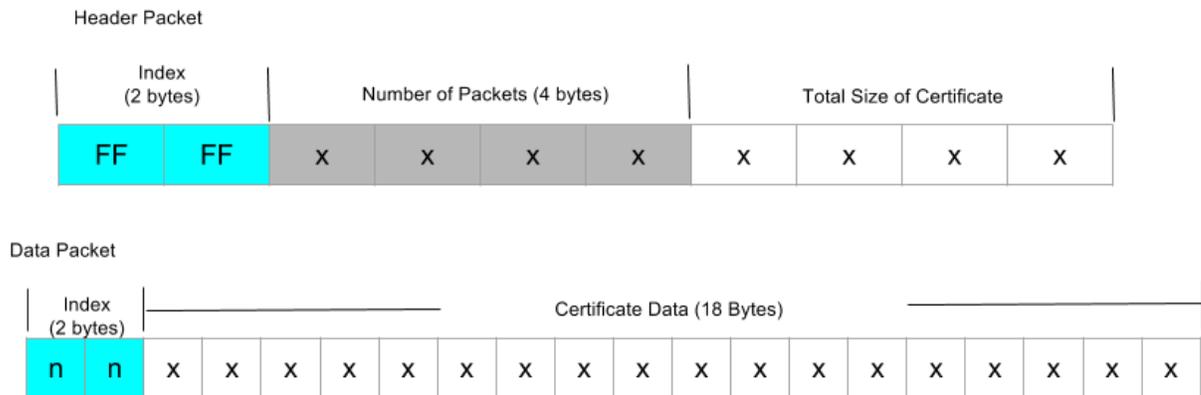


Figure 4: Top: Certificate Header Packet. Bottom: Certificate Data Packet. The header packet will be received first and shall describe the number and size of the data packets to follow.

Chapter 15: App

The App module for the S/MIME client, primarily consists of the User Interface. When implementing the User Interface, simplicity was the chief concern.

USER INTERFACE

In Android, User Interfaces (UI) consist of main components called Activities. Activities typically translate to visible screens and serve to break up the logic in an application. When creating the UI for the S/MIME client, we intentionally tried to separate the logic using different modules from the SMIME-CORE. A high level view of the application is shown in the following figure.

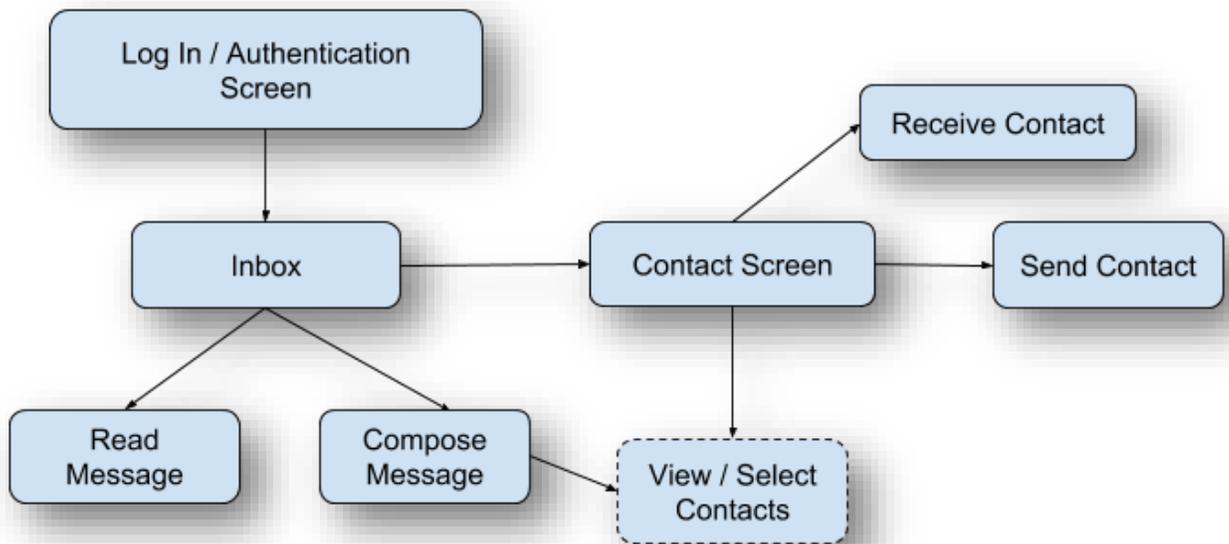


Figure 5: High Level view of the Application User Interface. Each box corresponds to a separate activity. The dashed box corresponds to a shared modal.

As you can see the Inbox screen serves as the base for commands dealing with messages. Whereas the Contact screen serves as the base for commands dealing with contacts/certificates. The View/Select contacts dialog, seen in dashes, is the one location where this separation breaks down. However, it is necessary since you can only send encrypted messages to contacts in your contact list.

The Inbox Screen primarily uses the Mail Manager and its related classes in order to properly display and compose emails. It uses the Contact Manager in order to convert MIME messages to S/MIME messages.

The Contact Screen primarily uses the Contact Manager, and the Communication Manager. The View/Select contacts dialog uses the Contact Manager.

PROGRAM OUTCOME

The results of the development of the S/MIME Client for Android are described in more detail in this section. The presentation of the program itself shall be described in more detail with visuals of the data. The difficulties faced will be outlined as well as their respective outcomes. Finally, as software development is an ongoing process, future plans will be discussed, chronicling future enhancements to the code, releases to the code base, and other planned contributions to the coding community in regards to S/MIME.

Chapter 16: The S/MIME Client

The object of this project is of course the application itself. This chapter will cover and display the outcome of the development of the S/MIME client. Images of the app screen will also be displayed.

INBOX

The Inbox is the main point of contact when dealing with emails. In the inbox you expect to see the headers of the emails in your mailbox and be able to select if you want to read, or compose a new email. Additionally for this project you want to be able to visually identify encrypted emails.

Luckily, using the Gmail APIs, the headers were easy to retrieve in a way where you did not have to retrieve the entire message. I created the inbox using a recycler view, which will populate the screen with the emails as cards that display the sender, the subject and display an icon indicating the type of email. A view of the inbox is in the following figure. Notice that encrypted emails are noted using the dark lock symbol, while the unencrypted emails have a unlocked symbol. Signed messages are indicated via a scribble. The email type is identified based on the MIME type of the message. All messages that are of type “application/pkcs7-mime” are encrypted, those of type “multipart/signed” are signed, and other are unencrypted.

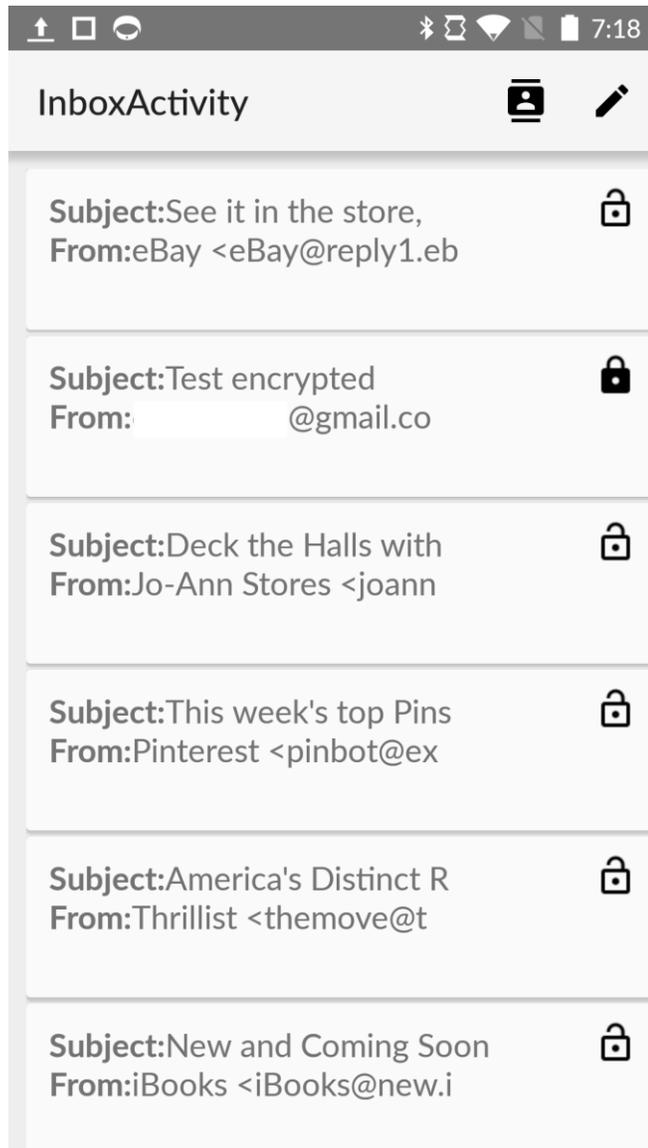


Figure 6: Inbox view of SMIME Client. Every card is a separate email, the icons on the right describe the message type.

READ MAIL

Of course an email client is only good if you can read emails. Once you click on a card of an email in the inbox, it will automatically take you to the read email screen. The read email screen begins with the headers that are a larger version of what is seen on the

inbox card. All content is shown in sections below the card. Multipart emails are supported. Additionally, images and html parts are also automatically displayed. The figures below show the read screen for an unencrypted email.

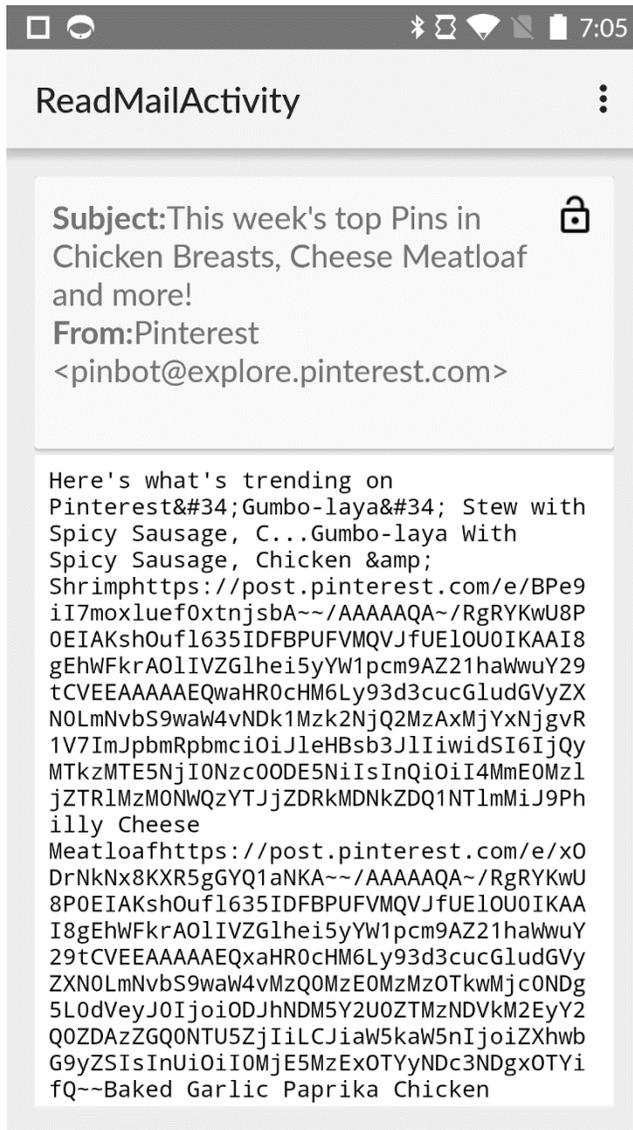


Figure 7: Read Screen of unencrypted email with headers and text part. With exception of the header, each card in this screen describes a different part of a multipart message.

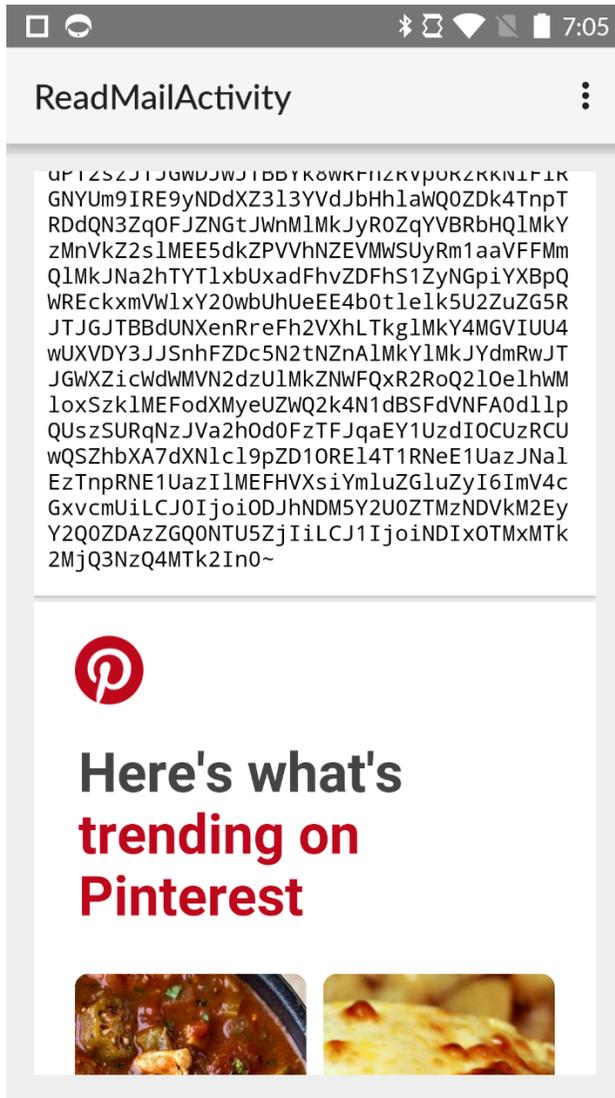


Figure 8: Read Screen of unencrypted email with text part and HTML.

The entire purpose of this project is to be able to read encrypted emails, so the images for those are displayed below. Notice, that for the email without a valid certificate, the contents are simply not displayed. It is of developer preference, but it can easily be altered to display an error message instead of nothing for these cases.

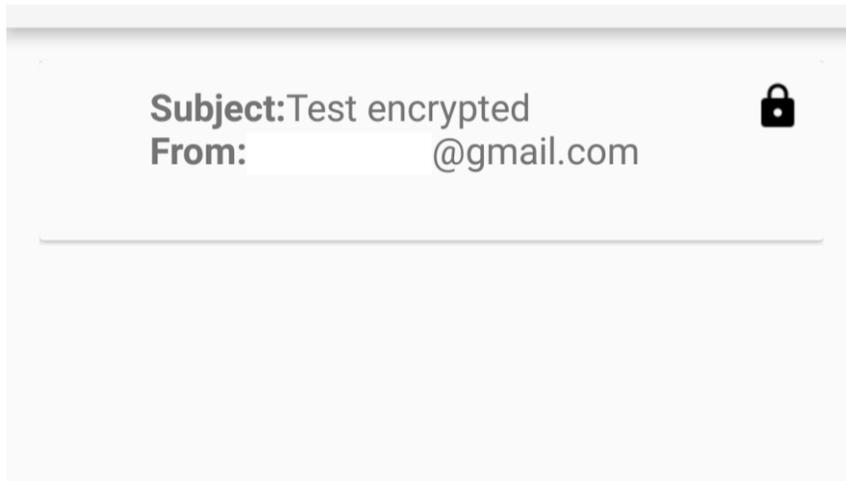


Figure 9: Read Screen of encrypted email without proper certificate. Encrypted emails without a proper decryption key will simply not be displayed.

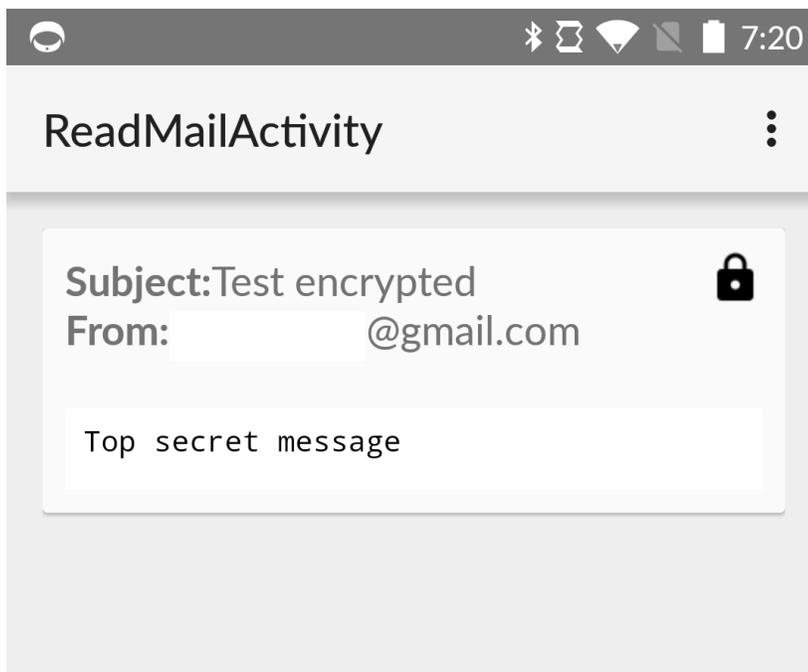


Figure 10: Read Screen of encrypted email with proper certificate.

COMPOSE MAIL

Composing an email is another necessary component of any email client. The compose email activity is available in the S/MIME Client by clicking on the pencil icon in the Inbox. Once in there, you can click on the lock icon to toggle between unencrypted, signed, and encrypted emails. The icon will change depending on the mode. Additionally, you can cancel the email by pressing the 'X' icon on the action bar. The paperclip icon enables you to add attachments, while the arrow icon submits the message to be sent. The figure below shows the compose screen.

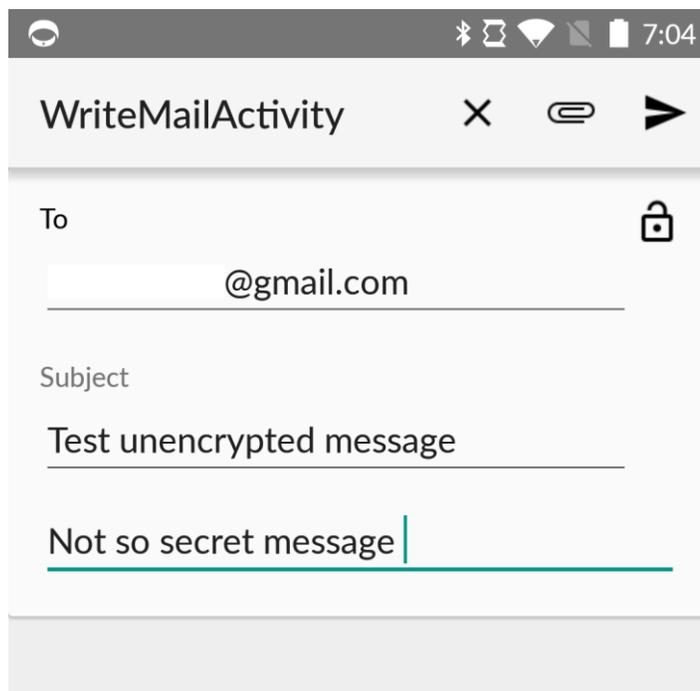


Figure 11: Compose Screen in unencrypted mode.

When the encrypted email mode is selected, the recipient text box becomes active and the To label becomes a link. Once you click on the To label you will get a

contact selector, which will display all of the contacts who have certificates in your key store. After selecting the desired recipients, their names will appear in the text box. The figures below will display both those situations.

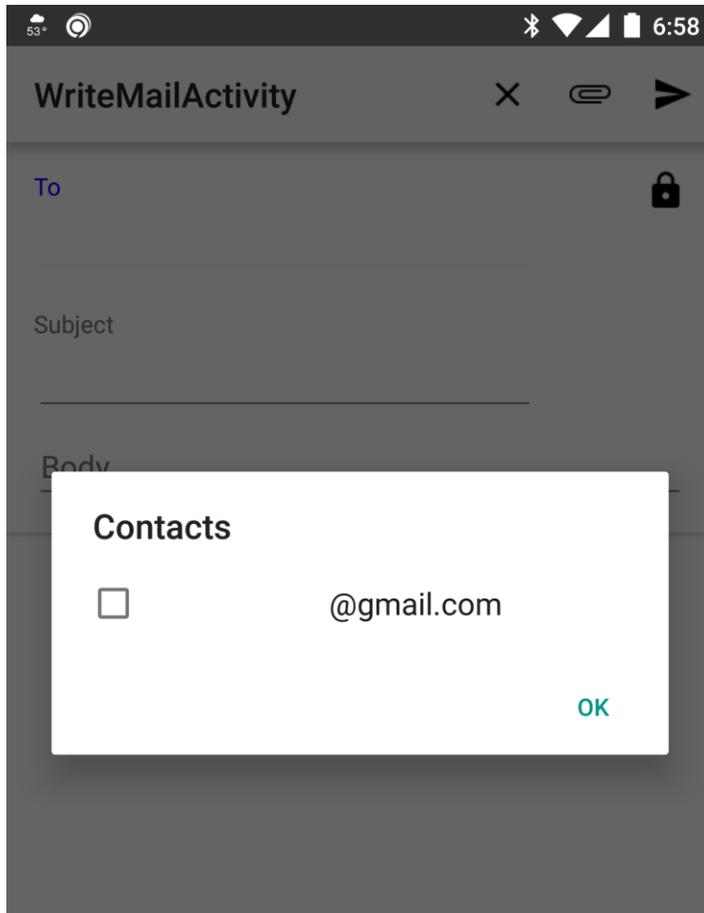


Figure 12: Secure contact selector in Compose screen. From this screen you can select multiple email addresses.

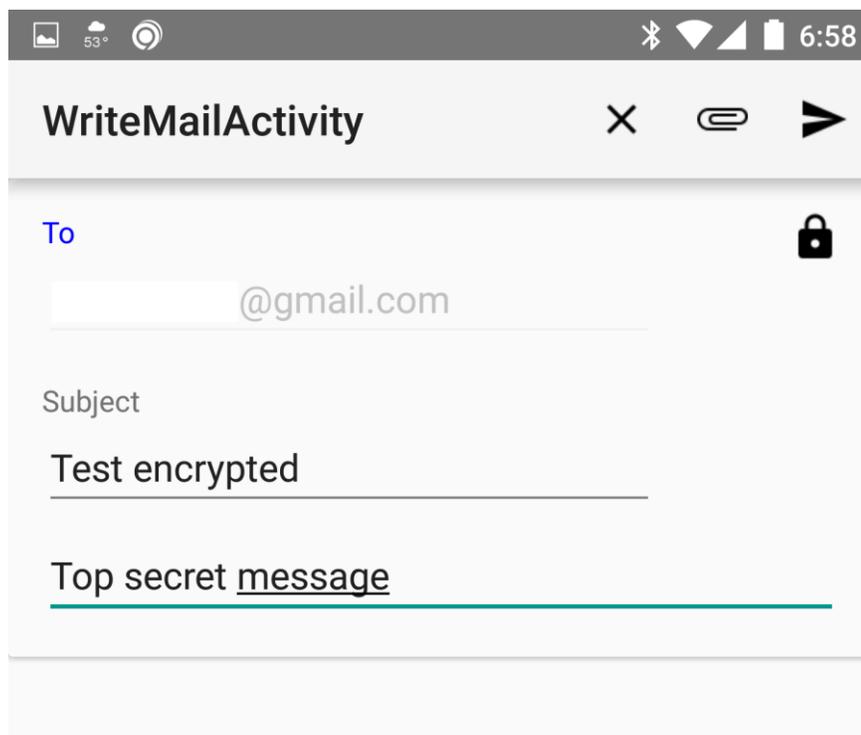


Figure 13: Encrypted mode Compose screen with contact selected.

CONTACTS

Accessing the contact screen is achievable by clicking the contacts icon in the inbox. You will be presented with a screen with 3 buttons, view contacts, receive contact info and send contact. Clicking view contacts will pop up the contacts dialog which will list out all of the contacts contained in the keystore. Clicking Receive contact info will take you to the device scan page where you can scan for nearby devices acting as a BLE peripheral. Once connected, you will receive the contact. Clicking send contact info will pop up a contact selector. The selected contact will then be set as the contact to transmit while the device acts like a peripheral. Figures for those screens are presented below.

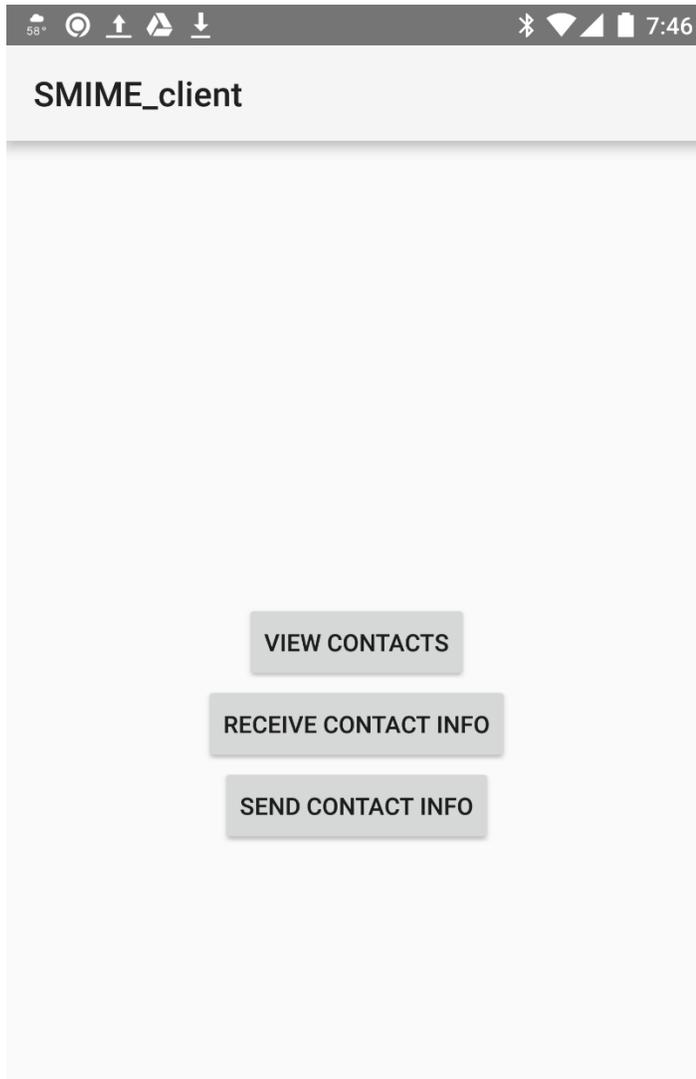


Figure 14: Contacts screen. From this screen you can select which contact related action to perform.

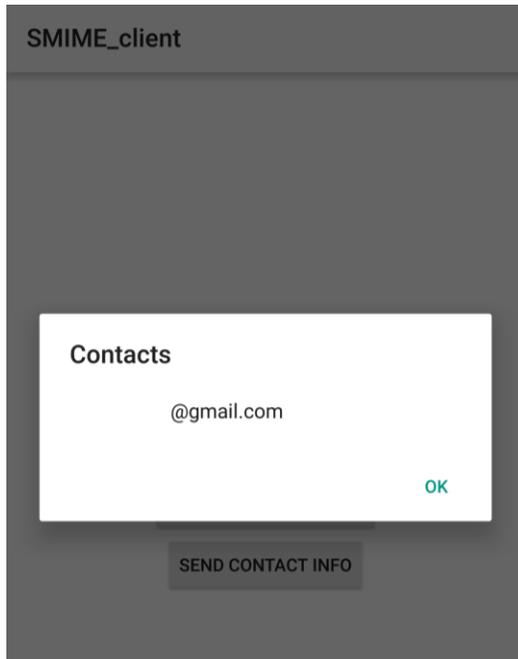


Figure 15: View contacts popup. This popup simply lists contacts with certificates.

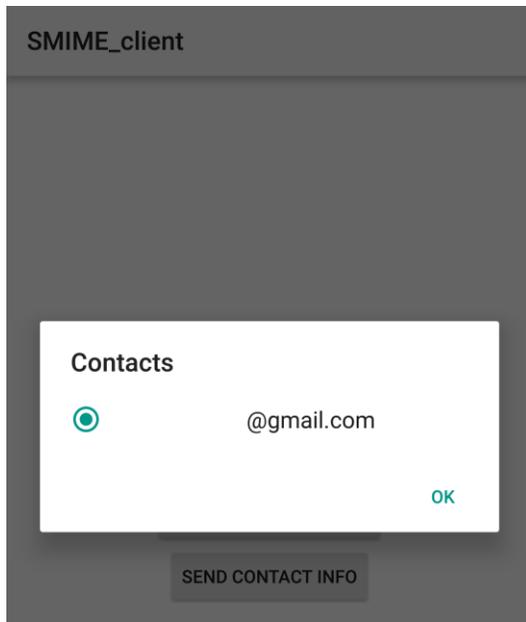


Figure 16: Select contact to send popup.

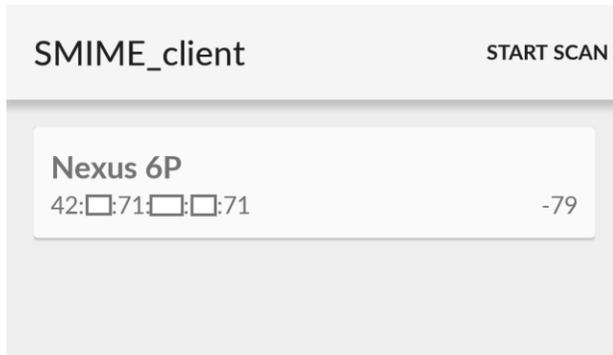


Figure 17: Select Contact server peripheral. You can see the device name, address and signal strength.

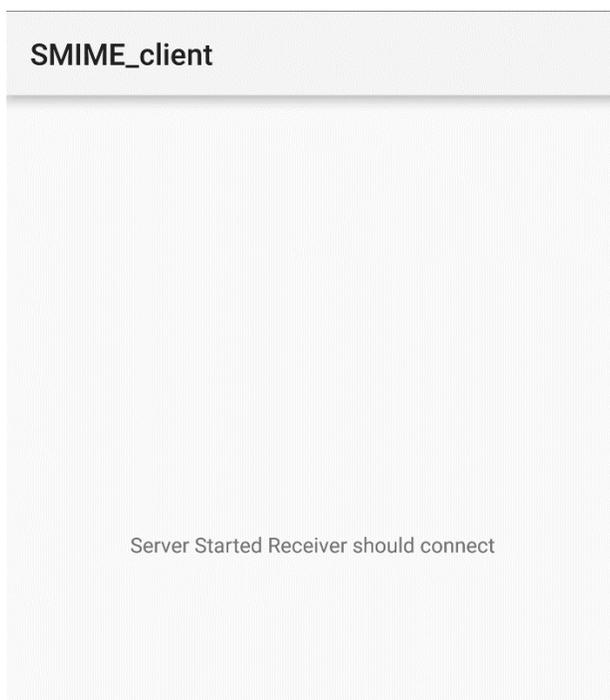


Figure 18: Server status page. The text in this page will update with progress of transfer.

Chapter 17: Difficulties Faced

While developing the S/MIME client, there were several difficulties faced along the way. This section will discuss those more in depth and describe the solutions, work arounds, or concessions that were decided for them.

DEPENDENCY ISSUES

As mentioned briefly in previous sections, Android had issues with internal libraries. While trying to convert a MIME message into an S/MIME body part, a runtime error due occurred causing the application to crash. This runtime error was due to the `javax.awt.DataFlavor` class missing from within the `javax.mail` libraries defined within Android. It happened that while Android had the `javax.mail` package integrated into the source code, it was a cut down version that was missing several classes. As I was using *Spongy Castle*, the S/MIME conversion hit upon this error when trying to deal with attachments. After searching the web, I found that the error was due to this version and found a replacement library that could fix this issue. The library was the *Javamail-android* library [24].

The *Javamail-android* library, was intended to port the missing `javax.mail` libraries into Android to solve the situation I encountered. Looking through the source code, I found that the port simply had the classes in `java.awt` in a new package called `myjava.awt` and modified an additional class within `javax.mail` to refer to that version of the `DataFlavor` class instead of the original `javax.mail` class. To my surprise this still did not solve the issue.

After further research, I realized that the additional class that was modified to refer to the `myjava.awt` library was itself within a `javax.mail` package. The naming itself is an important consideration due to the Android default behavior of favoring its own

packages over third party ones in conflicts. It so happened that although the Javamail-Android libraries would fix this issue, they were being ignored because the class that referred to DataFlavor was still listed in the javax.mail package and was thus being ignored.

Resolving this issue proved to have a few more steps than expected, because Javamail-android is now an archived project, I had to export the source code into my own github repository in order to rebuild it. I did so, but found that the original code base, was built on Eclipse in 2009. This meant that the gradle build environment used standard in the Android Studio IDE, was not existent in the code base. I was able to create a new project and import the source code in order to use the gradle build system and build these libraries. The issue was solved after that.

SPONGY CASTLE

Spongy Castle is a port of Bouncy Castle for the Android platform [26]. For the most part it was functional and performed well, with one major exception, mail. S/MIME conversion was not wanting to generate a message at all, looking into the error, I found that the administrator of the Spongy Castle repository ceased support of the mail modules in Spongy Castle due to “dependency issues”. Those issues happened to be the exact issue I mentioned above which was solved with Javamail-android. In order to fix this, I had to open a branch on the Spongy Castle repository and make several edits.

First, as with Javamail-android, I had to import the repository to use the gradle build system I was already using. I followed similar methods as I did with the Javamail-android libraries and created a new project with the gradle build environment. This allowed me to be able to attempt to build the project, and lead me into the next issue.

Since the administrator had stopped support of the mail sections of the Spongy Castle libraries, the mail module was out of date with regards to the packaging renamed that the administrator had done with the rest of the repository. In addition, the entire module itself was not even being built. To remedy this, I had to manually reassign the packages for the mail module code to match those of the rest of the module and re-enable the mail module to be recognized by the build system.

Once I had the build system running and the mail module repackaged, I indeed ran into the dependency issue that caused the mail module to not be supported. To my delight, the error said that the `javax.awt.DataFlavor` class could not be found. I imported the Javamail-android libraries into the Spongy Castle repository, and rebuilt it. This solved the issues with S/MIME.

MISSING HARDWARE

As part of the Bluetooth Low Energy (BLE) model. Devices can either be clients or peripherals. Peripherals are the devices that broadcast information, whereas the clients are the devices that consume that information. Support for Android devices for Android to become a peripheral is only implemented in OS releases starting from SDK 21, or Android Lollipop. This means that the majority of phones, those created to work with Android Kit Kat, did not have the capability to become an Android peripheral. As such, many of those phone developers, did not add the BLE hardware allowing the device to become a peripheral either. The issue here is that although a phone can be upgraded to Android Lollipop or above, without the hardware to broadcast advertisement data then a phone can still not act like a peripheral. This severely hindered testing efforts on BLE until I received a new device with the latest SDK and hardware.

ADDITIONAL 3RD PARTY ISSUES

Having fixed the dependency issues, and having found a workaround for the hardware issues, I ran into one more problem with S/MIME. Although S/MIME was tested to be working to convert MIME body parts into S/MIME body parts. There is an issue embedded within the Spongy Castle libraries that encounters errors with multipart messages. The issue is only seen in the S/MIME parser and has difficulty reading a multipart message due to it not recognizing the multipart as being an ASN1 object. This issue requires a significantly deep dive into third party code that I do not fully comprehend. Given the time allotment left in development, this problem has yet to be resolved. I planned to open a ticket for the Spongy Castle administrator to resolve this issue, however, this is within the mail module of Spongy Castle which they stopped supporting. This means that I shall have to merge in my port of the mail module into Spongy Castle and merge the code into the repo. Additionally, I would have to work with the Spongy Castle admins to be able to discover the error and fix the issue. However, since non-multipart messages could be encrypted and decrypted just fine (including messages with files as the body), I decided that the project could still fulfill the requirements with this error still resident.

Chapter 18: Future Plans

At the closing end of this project, there are a few loose ends to handle in terms of maintenance of the S/MIME Application.

OPEN SOURCE CODE

In order to allow this project to be expanded and evaluated with the open source community, it has been shared on a public repository online [30]. This project will be under continuous development and can be branched and forked by the public. Further steps on this will require testing the access to the repository where it will require administrative approval to push changes into major branches. As the sole administrator, it will require additional attention on my behalf to conduct code reviews and accept and merge code branches.

The repository will also require more documentation to describe its purpose and known limitations. Such work will be done using markdown in order for it to be read directly from the repository site.

JAVADOC

The APIs of this project require additional in code documentation. Javadoc is ideal because it will automatically convert the in code comments into html documents accessible via web browser. Javadocs are incredibly useful for traversing documented classes in source code.

USER INTERFACE

A very noticeable aspect of the S/MIME client that is deficient are the aesthetics of the user interface. While originally developed for functionality in mind, the user interface is simple but not very visually appealing. Updating visual aspects using material

design, as well as integrating more options menus and navigation options will serve for making the client more appealing to the general public.

ADDITIONAL ACCOUNT TYPES

The current client functions reading and sending emails from Gmail accounts. In order to improve the client, it would be beneficial to make it work with all email accounts that support S/MIME. This would involve altering and developing authentication modules to support the different account types.

ADDITIONAL ENCRYPTION ALGORITHMS

Currently, the S/MIME client works with 2048-bit RSA and 256-bit AES. While the encryption and decryption methods in the Crypto Manager are algorithm agnostic, the S/MIME conversion methods should be modified to be able to accept any algorithm supported by the Spongy Castle libraries. This will allow users to take more control of their security or default to the current settings.

AUTOMATED TESTING

While there is currently a test suite that helps stop future regressions in the project, these test cases need to be run manually by the developer. Tying the codebase to a continuous integration will allow the test suite to be run every time a commit is submitted for review on the code repository. This allows the system to automatically alert when a regression has been submitted into the code base.

BUG FIXES

Even in the best of code projects, bug fixes are going to be a given. Code will always have to be maintained and kept up to adapt for emerging technologies, timing issues, or hidden code paths. It should always be assumed that bug fixes will be needed.

THIRD PARTY CODE

Finally, since in this project I had to make significant edits to the Spongy Castle and Javamail-android libraries. A future task is to submit my changes to the proper repository branches and seek approval from the administrators. For Spongy Castle, this entails creating a pull request on their repository and submitting it for review. In the case of Javamail-android, however, I may need to maintain my fork of the exported repository on my own code repositories. This is a task that can be done soon.

CONCLUSION

The S/MIME Application was certainly possible to create. However, it was not an easy task due to the many dependency issues that were encountered in Android. Based on my experience, I suspect that one reason the Ciphemail app, or any other app for that matter, does not provide an inline S/MIME client is due to these very same issues. Additionally, wireless key passing is now possible, however, it has only been possible since the Android Lollipop release earlier this year. This explains why wireless key passing had not been developed as of yet. It was further hindered by the fact that many phones do not contain the hardware necessary for the device to be a BLE server. With the further development of the Android libraries and the growing demands for privacy, I fully expect that S/MIME can definitely be put into widespread use. I shall continue to do my part, to maintain the S/MIME client as open sourced software and will communicate my findings to the appropriate library projects such as Spongy Castle, javamail-android, and the JavaMail APIs. As for Ciphemail, I intend to open lines of communication with them to see if they will open source their Android distribution, in order to help create a client using already familiar technologies.

Although cyber security threats are constantly evolving and employ powerful techniques, I believe that with the proliferation of secure messaging can make us all a bit more secure about our data.

References

- [1] S. Fitz-Gerald. (2014, Dec. 22). Everything That's Happened in The Sony Leaks Scandal [Online]. Available: <http://www.vulture.com/2014/12/everything-sony-leaks-scandal.html>
- [2] G. Greenwald. (2013, June 6). NSA Collecting Phone Records of Millions of Verizon Users Daily [Online]. Available: <http://www.theguardian.com/world/2013/jun/06/nsa-phone-records-verizon-court-order>
- [3] G. Greenwald. (2013, June 7). NSA Prism Program Taps in to User Data of Apple, Google, and Others [Online]. Available: <http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data>
- [4] S. Radicati. (2013, April). Email Statistics Report, 2013-2017 [Online]. Available: <http://www.radicati.com/wp/wp-content/uploads/2013/04/Email-Statistics-Report-2013-2017-Executive-Summary.pdf>
- [5] B. Esslinger. (N/A). Current Trends in IT-Security [Online]. Available: http://www.e-mentor.edu.pl/_xml/wydania/5/62.pdf
- [6] J. O'Toole. (2014, Feb. 28). Mobile Apps Overtake PC Internet Usage in U.S. [Online] Available: <http://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-internet/>
- [7] W. Stallings. *Cryptography and Network Security Principles and Practice*, 6th Ed. Upper Saddle River, NJ: Pearson Ed. Inc. 2014
- [8] B. Schneier. *Applied Cryptography*, 2nd Ed. New York, NY: J. Wiley & Sons Inc. 1996
- [9] A. Geek. (2015, Aug. 27) RSA Encryption/Decryption for Longer Strings [Online]. Available: <http://crazytechstuffz.blogspot.com/2015/08/rsa-encryptiondecryption-for-longer.html>
- [10] D. R. Stinson. *Cryptography Theory and Practice*, 3rd Ed. Boca Raton, FL: Chapman & Hall. 2006
- [11] J. Lee, et al. (2014, Mar. 20) Graphics Processing Unit-Accelerated Double Random Phase Encoding For Fast Image Encryption [Online]. Available: <http://opticalengineering.spiedigitallibrary.org/article.aspx?articleid=1863832>
- [12] whatismyipaddress.com (N/A). The Mailman Inside Our Computers. Or: What is Simple Mail Transfer Protocol? [Online]. Available: <http://whatismyipaddress.com/smtp>
- [13] Aldie, et al. (2015, Oct. 2015). Simple Mail Transfer Protocol [Online]. Available: https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol

- [14] J. Brodtkin. (2011, Feb. 1). The MIME Guys: How Two Internet Gurus Changed Email Forever [Online]. Available: <http://www.networkworld.com/article/2199390/uc-voip/the-mime-guys--how-two-internet-gurus-changed-e-mail-forever.html?page=3>
- [15] P. Resnick (2008, Oct) Internet Message Format [Online]. Available: <https://tools.ietf.org/html/rfc5322>
- [16] Microsoft (2006, Aug. 16). Understanding S/MIME [Online]. Available: <https://technet.microsoft.com/en-us/library/aa995740.aspx>
- [17] Ciphermail (2014, June 19). Ciphermail for Android Reference Guide [Online]. Available: <https://www.ciphermail.com/documents/ciphermail-android-reference-guide.pdf>
- [18] Ciphermail (2015). Distribution Packages [Online]. Available: <https://www.ciphermail.com/downloads-gateway-distributions.html>
- [19] Oracle (2014). Java Cryptography Architecture Standard Algorithm Name Documentation [Online]. Available: <http://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html>
- [20] Oracle (2014). Package javax.crypto [Online]. Available: <http://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html>
- [21] B. Shannon (2015, Sept. 2). JavaMail API [Online]. Available: <https://java.net/projects/javamail/pages/Home>
- [22] TutorialsPoint (2015). JavaMail API – Quick Guide [Online]. Available: http://www.tutorialspoint.com/javamail_api/javamail_api_quick_guide.htm
- [23] B. Shannon (2015, Sept. 2). JavaMail for Android [Online]. Available: <https://java.net/projects/javamail/pages/Android>
- [24] A. Thiel (2009, Sept). javamail-android [Online]. Available: <https://code.google.com/p/javamail-android/>
- [25] T. Winters (2015, Oct. 15). The Legion of the Bouncy Castle [Online]. Available: <https://www.bouncycastle.org/>
- [26] R. Tyley (2015, Oct.). Spongy Castle – a repackaged version of Bouncy Castle for Android (which ships with a crippled version of BC) [Online]. Available: <https://github.com/rtyley/spongycastle>
- [27] G. McMillan (2013, Nov. 5). Hotmail No More: Gmail is Now the World’s Most Used Email Service [Online]. Available: <http://www.digitaltrends.com/web/hotmail-no-more-gmail-is-now-the-worlds-most-used-email-service/>

- [28] Google Developers (2015, October 7). Gmail API [Online]. Available: <https://developers.google.com/gmail/api/>
- [29] Google (2015). Gmail API v1 (Revision 34) [Online]. Available: <https://developers.google.com/resources/api-libraries/documentation/gmail/v1/java/latest/>
- [30] R. Diaz (2015, Dec.). SMIME_client [Online]. Available: https://github.com/ramirod/SMIME_client