Secure Messaging with in-app user defined schemes

M.Sc.(Tech.) Thesis
University of Turku
Department of Future Technologies
Networked Systems Security
May 2019
Mantena Mohit Raju

Supervisors:
Nanda Kumar Thanigaivelan, M.Sc.(Tech.)
Seppo Virtainen, D.Sc.(Tech.)

UNIVERSITY OF TURKU
Department of Future Technologies

MANTENA MOHIT RAJU: Secure Messaging with in-app user defined schemes

M.Sc.(Tech.) Thesis, 68 p., 31 app. p.
Networked Systems Security
May 2019

Cryptography has been the culmination of human trials and mistrials in an attempt to keep information safe from unintended access. We have learned from our mistakes in the past, and today with the help of both academician and software developers, we have robust cryptographic technologies. Cryptography however, is a race between increasing processing power of modern machines and the complexity of cryptographic systems. With quantum computing on the horizon, our present cryptographic systems seem to fall behind in this race. There is a need to catalyze research in the field.

Here, an application is proposed, which empowers users to write their own cryptographic schemes. It hopes to create a platform where people can share their cryptographic schemes and have an application that can help them share information securely. The author hopes, that an application which sources cryptographic schemes from users, would help catalyze research in the field. An application where the security implementation is dependent on the whim of the user could prove a hard target for attack. The thesis starts with a preliminary study of the Android platform. The thesis then analyzes implementations of a few secure messaging applications and then delves into details of NFC.

Using the background information accumulated during the course of this study, the authors attempt to formulate a sound implementation of a messaging application. The thesis is also accompanied with a proof-of-concept Android application that checks the viability of concepts discussed herein.

Keywords: Cryptography, NFC, Android, Security, Messaging

Contents

Abbreviations

ASS - Android System Service

API - Application Programming Interface

DAC - Discretionary Access Control

HAL - Hardware Abstraction Layer

JNI - Java Native Interfaces

IPC - Inter Process Communication

MLS - Multi Layer Security

OTR - Off-the-Record

SSH - Secure Shell

TLS - Transport Layer Security

ECC - Elliptic-curve cryptography

IM - Instant Messaging

ECDH - Elliptic-Curve Diffie-Hellman

PGP - Pretty Good Privacy

AKE - Authenticated Key Exchange

EFF - Electronic Frontier Foundation

UID - User Identification

GID - Group Identification

PID - Process Identification

MSB - Most Significant Bit

LSB	- Least Significant Bit
GCM	- Google Cloud Messaging
IV	- Initialization Vector

Chapter 1

Introduction

Any kind of change, that has ever brought about a major shift in the human way of living, has always been accompanied with a change in the way we communicate. Be it the great industrial revolution, the sociopolitical changes in the early 20th century, the Cold war or the advent of information age at the end of 20th century. Communication and its advancements have always played a vital role in human history.

The inherent power of information has motivated people to come up with ways to keep certain kinds of information hidden in transit. To keep secrecy, we have come up with a number of ingenious methods to scramble information for everyone except the intended receiver of the message. Depending on the importance of hidden information, there have also been people who have attempted to gain unauthorized knowledge of this information. This constant game of cat and mouse, between the information scramblers and information deciphers led to the advent of the field of cryptanalysis.

With the dawn of information age, we have faster communication channels. We have communication channels that use different types of transport medium to transmit data. Information itself is no more just text. Information can now be in the form of voice, video or text. The mode of access to information too has many varied end points of consumption. This complexity in information, introduced by the information age prompted formation of governing bodies that would advise people and organizations of ways to secure one's data.

However, the growing complexity of these systems makes it difficult to understand the underlying concepts of these systems. This thesis tries to bridge this gap by proposing an application which can be used by people with rudimentary programming skills to write their own simple cryptographic schemes. This would allow them to encrypt their SMS messages between their Android devices using their own encryption schemes. It hopes to create a platform where people can share their cryptographic schemes and ideas and possibly lead to better insight in the field of cryptanalysis.

As the work here is based on the Android platform, the thesis starts with an introduction to the Android security platform. It then cover the basic concepts of cryptography and follow this with an overview of some successful mobile messaging applications. The thesis then proposes a secure messaging application with a walk through of a proof of concept Android application.

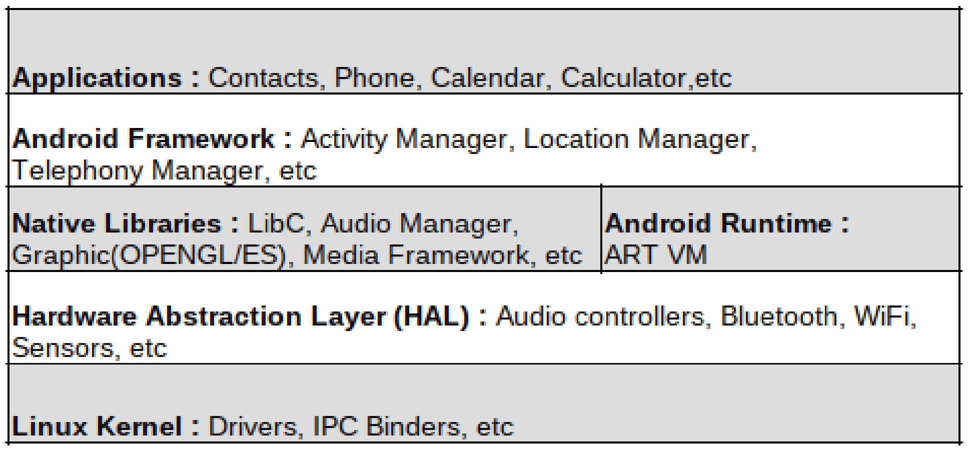
Chapter 2

Android Security Architecture

This section covers Android architecture from a security perspective. Android's security architecture builds on well known Linux security features. Keeping this in mind the section starts with details of Linux security implementation, which is then followed by new security features introduced in Android. Finally the section is concluded with a discussion on SEAndroid which is an extension of SELinux.

2.1 Android system architecture

Android platform was selected for studying the security architecture of modern systems because of its extensive reach in the market and for its support for NFC. An overview of the Android Architecture is depicted in Figure-2.1. The figure shows hierarchy of data access in an Android system. Applications request access to system resources through the Android framework. The Android framework, in turn requests accesss to the resources through Native Libraries and Android Runtime. Android Runtime and Native Libraries can access resources from the Linux Kernel through the Hardware Abstraction Layer (HAL). Android uses security features available in Linux operating system and adds new application layer security rules to complete its own security model. What is referred to as the Linux operating system here is the Linux kernel. To make use of the Linux kernel



Applications : Contacts, Phone, Calendar, Calculator,etc	
Android Framework : Activity Manager, Location Manager, Telephony Manager, etc	
Native Libraries : LibC, Audio Manager, Graphic(OPENGL/ES), Media Framework, etc	**Android Runtime :** ART VM
Hardware Abstraction Layer (HAL) : Audio controllers, Bluetooth, WiFi, Sensors, etc	
Linux Kernel : Drivers, IPC Binders, etc	

Figure 2.1: Android System Architecture.

in a low form factor device, Android made a few changes in the way linux handled secondary memory and kernel initialization. The following packages/features are different in Android [1] [2] [3].

- **Yet Another Flash File System (YAFFS2)**. Android uses YAFFS instead of ext4 file system which it commonly used in Linux. It makes sense as Android is almost always installed on portable devices with flash storage devices.

- '**The init**' module has been implemented differently in Android. Android does not follow either POSIX [4] or SystemV [5] standards for initialization of user space. The kernel initialization is however similar to other Linux distributions. Once the kernel is initiated, the user space initialization uses the **init** scripts defined in *init* and *initrc* files (other *init* scripts are included in the *initrc* folder). The *init* initializer is a compiled ELF executable which among other things, excutes scripts specified in *init.rc*. Executable and Linking Format (ELF) is the format to which binary executable files have to conform to in order to be executable in the Linux environment. Android uses it's own initialization language called the Android Init Language. This language contains 4 types of statements - Actions, Commands, Services and Options. Actions are statements that would follow a set of commands when a trigger in invoked. For example, the 'on init' action is followed by tab spaced operations in the following lines. These lines represent the operations that the systems must perform when it goes through normal initialization. Other actions are 'on early init', 'boot', 'charger', etc. Vendors of Android devices have the provision of adding extra actions in case they need to handle device specific peculiarities. Services are programs (mostly daemons) that would be launched on startup and are essential for the functioning of the system. Options are parameters that are passed to the service to tweak its behavior based on system requirements. The *init.rc* script contains the default initialization steps on an Android system. Following is what can be surmised from the file.

- When the system boots, the action with trigger 'on boot' is initiated. This initializes the network interface and sets up file system mount points. SELinux partition is loaded and the the ownership of the partition is set to system. This is to enable remote update of SELinux policy.
- Control groups are created for mount points and processes. Control groups called cgroups in short was introduced by Google in 2007, and was later inducted into the mainstream Linux kernel and used in many applications like Docker, Hadoop, etc. This allows Linux to limit resources to processes, prioritize resource share, accounting of resource usage and controlling processes. Through control groups, both user space and system applications can be limited to use a certain percentage of CPU time and certain bytes of primary and secondary memory. This prevents any single user application from consuming all compute resources and making the device unusable. The *init.rc* contains an extensive set of commands for allocating/limiting resources to processes.
- Services are started after completion of control group creation. This is done using the class_start command, which starts services belonging to the specified classes. All basic service belong to the class_core or main.

- One of the first Android services to be started is the zygote daemon (which is in-fact just the */system/bin/app_process* program), this initializes the ART VM. All the service and process initiated prior to this step pertain to the Linux system [6]. Once ART VM has started, the Android System Server (ASS) is initiated. The ASS first initializes the 'android_servers', which provides interfaces to the Linux system services and Hardware Abstraction Layer (HAL) services like Wi-Fi, system clock, battery, telecommunication network service, etc. These services are managed by Linux using the Control Groups, Discretionary Access Control (DAC) and SEAndroid. The interaction between Linux kernel and Android is accomplished with the help of interfaces defined by '*android_server*'. These interfaces are implemented using Java Native Interface (JNI). JNI makes communication between system native applications and ART native applications easier. In Android, JNI provides the Android system with binders/interfaces to communicate with Linux system services of Wi-Fi, system clock, etc. Using Binders, ART applications can access system services. The implementation of binders, replaces the standard Inter Process Communication (IPC) mechanism in Linux. Binders were implemented to offer secure IPC. Once these service are initialized, the applications specified in system partition (contacts, messaging application, phone applications and other default applications) are initiated followed by user installed applications.

2.2 Implementation of security

The following section discusses the components of Android system responsible for its security. This includes the discussion of Linux DAC, IPC, Segregation of System Applications, Application signing schemes and SELinux.

2.2.1 Linux DAC

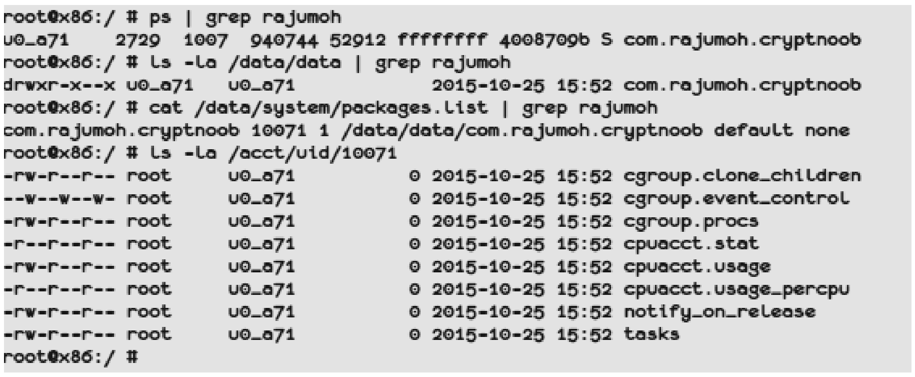
Android capitalizes on many well proven Linux security features. Linux is a multiuser system. It performs user and process management using Discretionary Access Control (DAC) (the study assumes that reader has some knowledge of DAC, though it does cover it briefly here) [7]. DAC allocates identities to users, processes and groups. User identities are designated to either individual users or programs. When a process starts

- The process is assigned a number,
- The process's parent's (program that spawns it) process ID is noted,
- The user or process that spawned the process is noted.

Groups are identifiers that can be associated with one or more users. User, group and process ID's would hence forth be referred to as UID, GID and PID. When an application attempts to run, the following checks are performed.

- Who is the owner of the file/executable/process? Does the user, trying to invoke the process, belong to the right group?
- Does the file, have read/write/execute permissions for that particular user (directly or through a group) ?

A process is executed only if the above rules are satisfied. DAC is called 'discretionary' as the users who have access to the file have some level of control with regard to sharing file access. Android uses DAC to assign each installed application with a unique UID and GID. The application is also assigned a data directory which cannot be accessed by other applications. Figure 2.2 shows the permissions of a newly installed application. UID of an application can be obtained from the list of running application by searching for the application package name that is included in the *AndroidManifest.xml* file of the application. *AndroidManifest.xml* specifies the list of resources the application needs access to. The file also specifies the location of application initialization files. The demo



```
root@x86:/ # ps | grep rajumoh
u0_a71    2729  1007  940744 52912 ffffffff 4008709b S com.rajumoh.cryptnoob
root@x86:/ # ls -la /data/data | grep rajumoh
drwxr-x--x u0_a71   u0_a71            2015-10-25 15:52 com.rajumoh.cryptnoob
root@x86:/ # cat /data/system/packages.list | grep rajumoh
com.rajumoh.cryptnoob 10071 1 /data/data/com.rajumoh.cryptnoob default none
root@x86:/ # ls -la /acct/uid/10071
-rw-r--r-- root     u0_a71          0 2015-10-25 15:52 cgroup.clone_children
--w--w--w- root     u0_a71          0 2015-10-25 15:52 cgroup.event_control
-rw-r--r-- root     u0_a71          0 2015-10-25 15:52 cgroup.procs
-r--r--r-- root     u0_a71          0 2015-10-25 15:52 cpuacct.stat
-rw-r--r-- root     u0_a71          0 2015-10-25 15:52 cpuacct.usage
-r--r--r-- root     u0_a71          0 2015-10-25 15:52 cpuacct.usage_percpu
-rw-r--r-- root     u0_a71          0 2015-10-25 15:52 notify_on_release
-rw-r--r-- root     u0_a71          0 2015-10-25 15:52 tasks
root@x86:/ #
```

Figure 2.2: System Permissions of a User Installed Android Application

application's package name is *com.mohitr.cryptnoob*. In Figure 2.2, the first command which lists processes running on android containing the string *rajumoh* shows that the application *com.mohitr.cryptnoob* has UID 'u0_a71'.

```
ps | grep rajumoh
```

The next command lists the application data directory. Installed applications have their data directories placed in '/data/data'. The output shows that the application's data directory exists and its UID and GID are 'u0_a71'. This along with the read-write-execute parameters shows that applications 'u0_a71' has all permissions on the directory. Processes and users belonging to the 'u0_a71' group have read and execute permissions and other users have execute permissions on the directory (i.e., they could open the directory).

```
ls -la /data/data | grep rajumoh
```

The third command extracts entries related to the application from the '*/data/system/packages.list*' file. *packages.list* and *package.xml* contain package name, UID, data directory location and other parameters related to every installed application.

```
cat /data/system/packages.list | grep rajumoh
```

The last command shows the files that are related to 'Control Groups' discussed earlier. Control groups of application resources resides directly with the root user and the application itself has limited control over it by virtue of group permissions. This prevents applications from changing their own control groups and gaining higher privileges

```
ls -la /acct/uid/10071
```

2.2.2 IPC

Using the above techniques, Android achieves a kind of application sandbox. Applications end up being enclosed in Linux user space with limited access to resources. The next step in Android security is definition of schemes for secure interactions between applications and Android system services. The interaction between subsystems is called Inter Process Communication (IPC). Android implements IPC using Binders. Figure 2.3 shows an example of 2 processes sharing information over binders. Binder implementa-

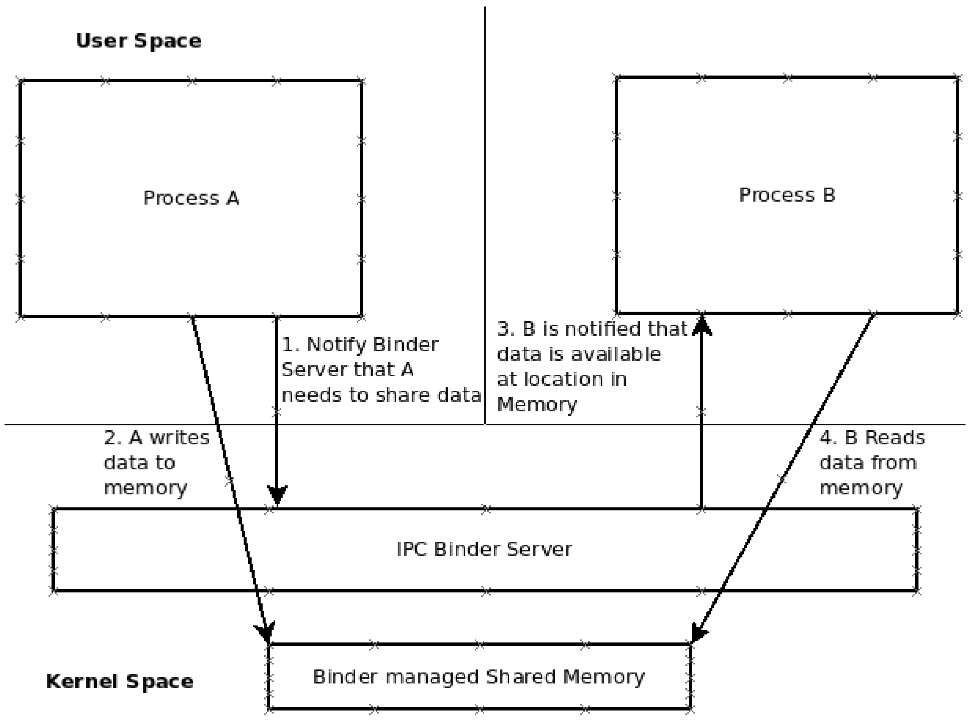

User Space
Process A
Process B
1. Notify Binder Server that A needs to share data
3. B is notified that data is available at location in Memory
2. A writes data to memory
4. B Reads data from memory
IPC Binder Server
Kernel Space
Binder managed Shared Memory

Figure 2.3: Android System Architecture.

tion used in Android is based on OpenBinder [8]. Memory space of a processes in Linux can be accessed by the process user or system admin (root). Binder manages memory space of processes by becoming owner of a part of memory space of each process, this memory space is read only to the process. The interface that provides handle to this binder memory space of a process is the /dev/binder file. In a typical inter-process communication, consider two processes, Process A and Process B. When process A wants to send information to process B, it would send a request for communication to the Binder driver. Process A can accomplish this using the IBinder interface. The Binder driver then contacts the Binder server, which writes the information to be exchanged onto the process B's binder managed memory space. The Binder server then notifies process B that there is some information/data available at a particular memory location in its binder space. As process B has read permission in its own binder memory space, it can access the information.

Android Intents (Allow communication between activities among other things), Content-Providers (database, file system management, etc) and Messengers are all implemented using binders and one can identify these binders in the Android source code by searching for the IBinder interface. The availability of this high level abstraction provides developers with tools to have secure access to application resources.

Binders form the basis of Capability Based security model of Android. When an application is installed, it prompts the user for a set of permissions that the application would use when installed. Application must declare its capabilities in its manifest file (*AndroidManifest.xml*) inside uses-permission tag. When an application does this, it is declaring its intentions of accessing specified Binders. Once an application has declared its intentions, it will have access to the IBinder interface which contain method declarations for available operation on the Binder. All objects that support IPC must implement IBinder interface and are then referred to as Binder object. Binder objects perform IPC operations in a transactional fashion. Each transaction contains reference to the target object, a unique global ID and address to the binder data buffer. PID and UID are added by the Binder driver to prevent privilege escalation. The target object can then check the PID, UID to perform operations on the Binder request. Also, as binders have global ID that is tracked by the kernel, it is not possible to duplicate binders. Though binder references can be passed between processes, the execution of the binder is performed only by the process which owns the binder's memory space (and hence can perform checks).

So far, the discussion on binders have been for the use cases where references to binders are explicitly obtained. However, there are services that are global. Services like Wi-Fi, Bluetooth, etc are not accessible to every application. Only applications that have requested for services are provided access to the binder of these global services. Binder providers (referred to as *contextmanagers*) can implement extra checks to validate service usage.

2.2.3 System Applications

Android distinguishes applications as system and user installed applications. Android devices have pre-installed default applications. Phone book, contacts, messaging and google-playstore are a few common system applications. These might also include vendor specific applications. System applications hold a special place in Android and are installed in the '*/system/priv-app/*' directory. The applications must be signed with the platform signing key and the applications protection level is set to '*signatureOrSystem*'. These applications have system level privileges and cannot be uninstalled by the mobile user (only disabled). The reason behind this is to provide vendors with the options to customize their products and provide system applications with some level of protection.

User installed applications are installed in the '/data' directory. These applications only have the privileges that have been requested by them.

Another set of applications that do not fall under the above two categories are rooted applications, these are applications that request for 'root' user permissions when executed. Normal Andriod devices cannot recognize these requests. Rooting is not supported by Google Android framework, as a result it is difficult to find official documentation on rooting. However there are a few organizations that provide tools for rooting [9] [10].

Though different rooting tools operate in different manner, they chiefly replace the existing kernel with ones that contain super user tools like 'su, chmod, chown, etc'. They also might add various utility scripts that applications can later capitalize on. It must be noted that these applications break the security architecture of Android.

2.2.4 Securing Android components with Signatures

Various components of Android are signed. From the time an Android device boots, upto the running of an application, there are a series of signature checks that take place to validate the integrity and authenticity of system. A few of these stages are discussed here.
Android uses verified boot [11] which provides the capability to detect software integrity, 'signal of integrity' for remote attestation, encryption and Trusted Execution Environment (TEE) [12]. The verified boot implementation uses dm-verity [13] [14] to map block devices (here mapped to disk partitions usually) on Linux into virtual block devices. Each virtual block device is assigned a hash value (4k - SHA 256). The block devices are mapped into a hierarchy with each leaf on the tree containing a hash value calculated for the block device. The nodes of the tree have a hash value, that is a hash function of its leaves. This forces any changes in hash values of the nodes or leaves of the tree to be reflected onto the root's hash value (referred to as the root-hash). This hash value is verified with the Original Equipment Manufacturer's (OEM) public RSA key (which has a modulus of 2048 or higher). If there is a mismatch of keys, Google defines another set of rules to notify the mobile user of a possible security breach on the device. The integrity check of devices is very important against root-kit malware that can hide from detection techniques, by embedding themselves into privileged locations on file system.
Code signing of applications is common across all mobile device platforms. Android needs every application released to the play store to be signed with developer's keys. This enables trusted application updates. System applications also need to be signed, and the keys are verified with the platform keys. Platform keys are maintained by the device man-

ufacturer.

System update is another operation that employs signatures. When OEM's send Over The Air (OTA) updates to mobile devices, the update is downloaded as a zip file, extracted and checked for signed keys from the OEM. The signature keys are present in '*/system/etc/security/otacerts.zip*'. If the zip file passes verification, then the system enters recovery mode and starts the update by placing files from the zip to correct locations.

2.2.5 SELinux

SELinux implements Mandatory Access Control (MAC) as opposed to DAC. However these two methods of access control are often used together to bring security robustness. DAC falls short when security designers need to dictate more fine grained access rules. For example manufacturers might want to change the behavior of certain root processes without going through the steps of creating a completely new GID for the process by changing permissions of all the files related to the process. This could be tedious and in worse cases leak security if configurations are missed. As a result Google started the SEAndroid project which ports SELinux to Android. SEAndroid was first introduced in Android 4.2 but it went mainstream with Android 4.4.

SEAndroid operates in three modes, 'permissive', where security violations are just logged, 'enforcing', where violations are denied and logged and the last mode being 'disabled'. SELinux consists of 4 elements.

- Object managers (OM) : When processes attempt to access a resource, they are termed as 'subjects' and the resource being accessed is called the 'object'. Object managers decide whether a 'subject' has permission to access an 'object'. The object information in maintained by the Object managers. OM is also responsible for making sure that the decision is carried out. This decision making is also referred to as 'action'.

- Access vector cache (AVC) : OM, refer to the AVC to check if the subjects have access to the object. AVC maintains a cache of recent requests to improve performance of decision making. In case AVC cannot find the request in its cache, it asks the Security server for the decision and caches the decision.

- Security server : Security server resides in the kernel and maintains all security policies.

- Security policy : Security policies are a set of rules specified by the system administrator that SELinux uses. It cannot be changed by subjects once SELinux has been enabled.

SEAndroid supports only type enforcement. This means that all process that fall under the purview of SEAndroid must be associated by a type. The type definition is done in security context or simply label. It is a colon separated string containing the 4 parameters

- user-name (always set to u)

- role (set to r or defaults to object_r)

- type

- Multi-Layer Security (MLS) security range (always set to s0)

The security context 'type' can be associated with a set of attributes. Policies can be differentiated based on their 'type' and its attributes. Roles can be assigned to 'subjects', where each role can be associated with security context 'type'. SELinux also allows transition rules for type and domain. A detailed discussion of the policy rules would be out of scope of this study.
SEAndroid cannot populate its 'Security Server' with too many rules as that would lead to performance issues. As a result, it does not implement MLS, does not include too many users or roles. SEAndroid secures Android Binders. Android Binders have security

hooks that SEAndroid can use to enforce security on Binders. Similarly many changes were made to other Android libraries to allow for SEAndroid to enforce policies. File system libraries are also modified to include security context on file system attributes (as SELinux reads file system attributes to obtain security context of a file). An extensive discussion on the changes to SELinux for Android have been well documented elsewhere [15] [16] [17].

Chapter 3

NFC in Android

Near field communication or NFC allows communication between two closely placed devices through electromagnetic induction and traces its origin back to Radio Frequency Identification (RFID). NFC does not refer to a single technology, but is a conglomerate of different radio frequency technologies like ISO/IEC 14443, FeliCa [18], GSMA NFC Standard and Single Wire Protocol [19]. Information of these technologies can be found on the NFC homepage [20]. Today NFC has come a long way with new innovative use cases emerging everyday. NFC operates in the 13.56 MHz radio frequency, with supported speeds of bit transfer being 106, 212 or 424 Kbit/s [21].
NFC can operate in three modes.

1. **Card Emulation mode**. NFC device can emulate NFC card and chips. When in emulation mode, other NFC devices can read the NFC data (also referred to as NFC tag)

2. **Reader/Writer mode**. This allows NFC device to read data embedded in NFC tags or from devices in Card Emulation mode

3. **Peer-to-peer mode**. To enable ad-hoc exchange of information, peer-to-peer mode is used

NFC devices have a loop antenna [22] which uses magnetic induction to communicate between devices. When a NFC device or a NFC tag comes in close proximity of another NFC device, an effective air-core transformer is formed. In case of NFC tags, this generates the current needed to make the tag functional. This is then used to gather information from the NFC tag/device. NFC tags are special and the data coding supported on NFC devices are Miller coding and Manchester coding. NFC devices are also capable of transmitting data at the same time (peer-to-peer). NFC devices support two modes of communication

1. **Passive communication**. When an NFC device exchanges information within the magnetic field of other NFC device without using its own magnetic field, it is referred as passive communication. NFC tags operate in passive mode. Fully capable device can also operate in passive communication mode. Passive communication makes use of NFC possible with passive entities like credit cards, game tags, keys and tokens. This feature, along with the ability of mobile devices to communicate on the Internet makes NFC devices suitable candidate for on-line identification and authentication.

2. **Active communication**. In active communication both NFC entities can generate their own NFC fields. When both the NFC entities can generate their own field, it raises the possibilities of ad-hoc communication. Peer-to-peer communication uses this ability.

3.1 NFC use cases

NFC devices have varied applications in real world.

1. **Mobile payment** - Mobile payment through NFC devices is already in use today. Use of mobile devices operating in active communication mode to make payments have been implemented as Google Wallet on the Android platform and Ap-

ple Pay[Apple Pay], to name a few. NFC technology is also being used with credit cards, where one can get NFC enabled cards.

2. **Data transfer** - Two NFC enabled devices can be used to transfer data between two devices. This, however, is not a popular use case due to availability of other, better modes.

3. **Identification and authentication** - NFC devices can substitute RFID's. All RFID use cases are valid here. As most popular mobile devices have started using NFC, in near future, it could replace RFID's for personal identification. It can also provide additional functionality, by using Internet connectivity of mobile devices to provide robust identification systems

4. **Gaming** - NFC and QR code readers have enabled the possibilities of real world gaming scenarios. A number of scavenger hunting games are available on Android Play Store which use NFC

3.2 NFC and Android

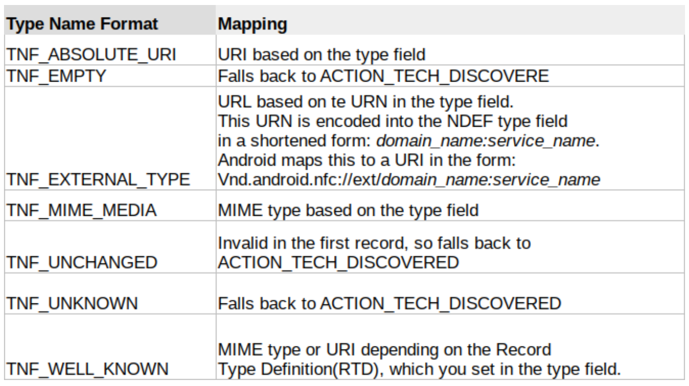
Android provides extensive support for NFC on its NFC capable devices. A list of supported devices can be found on the NFC website. Android provides developers with high level Application Programming Interfaces (API) to access all NFC features. These libraries abstract the underlying native libraries and provide the developer with easy to use extension to be able to operate with NFC. At the system level NFCService takes care of NFC operation. Application developers do not have direct access to this service. Instead they utilize libraries available under the package '*com.Android.nfc*' and invoke operation on NFCService through event driven model. For example, if the application wants to read NFC tags of a certain type, it would register callback method with the NFCService. When the device reads a tag, the NFCService would handle it. Based on the tag data, NFCService determines the application that has registered for the tag, and invokes the registered application.

All three modes of operation follow the above work flow. An event driven model was important as the designers wanted to minimize user input when operating with NFC operations. Another reason for the event driven model was security, as the NFCprotocol itself does not provide any security features. Security must be achieved by the developers using encryption and cryptographic hash functions.

Android encapsulates NFC messages into NDEFMessage (NDEF stands for NFC Data Exchange Format) object. NDEFMessage can contain a number of NDEFRecords. The actual data is part of NDEFRecords. A NDEFRecord contains three fields : type, id and payload. The type field specifies the type of the payload. A table of available types on Android is shown in figure 3.1. Id field is not mandatory and the payload field is filled based on the type specified. An Android application must first register itself with NFC-Service stating the type of NFCMessage that the Android application would respond to. When NFCService receives a message, it opens the NDEFMessage array and picks the first NFCRecord. The type and the payload specified in the first record determines the target application that would be activated/chosen by NFCService to complete the request. For example, a Credit Card Wallet Android application would first register with the NFC-Service stating that it would respond to NDEFMessage from a NFC enabled Credit Card. When such a card is brought in contact with Android device's NFC field, NFCService would recognize the Credit Card tag and invoke the Credit Card Wallet application.

3.3 HCE Overview

Hardware-based card emulation (HCE) is used by Google Wallet application. Its operation sequences are worth noting as they are a bit different from normal NFC implemen-



Type Name Format	Mapping
TNF_ABSOLUTE_URI	URI based on the type field
TNF_EMPTY	Falls back to ACTION_TECH_DISCOVERE
TNF_EXTERNAL_TYPE	URL based on te URN in the type field. This URN is encoded into the NDEF type field in a shortened form: *domain_name:service_name*. Android maps this to a URI in the form: Vnd.android.nfc://ext/*domain_name:service_name*
TNF_MIME_MEDIA	MIME type based on the type field
TNF_UNCHANGED	Invalid in the first record, so falls back to ACTION_TECH_DISCOVERED
TNF_UNKNOWN	Falls back to ACTION_TECH_DISCOVERED
TNF_WELL_KNOWN	MIME type or URI depending on the Record Type Definition(RTD), which you set in the type field.

Figure 3.1: NFC Record Types

tations. HCE was developed by Doug Yeager for CyanogenMod and later merged into Android base. Unlike normal NFC operation where NFCService handles all operations and developers could only place callback requests (in an event drive model) in their code, in HCE the entities interact with each other. HCE runs continuously in the background and is coded to respond to external cards, thus performing two way communication.

Chapter 4

Basic Concepts of Cryptography

This chapter covers the basic techniques and concepts that are in use in the field of cryptography. The chapter starts with symmetric key cryptosystems, followed by cipher operational modes and public key cryptosystems. The chapter concludes with a discussion on Digital signatures and Hash functions.

4.1 Alice, Bob and Eve

Cryptology employs the use of fictional characters to present complex theoratical situations. The most common names in use are Alice, Bob, Eve and Mallory. The names Alice and Bob were first used in the papers presented by Rivest-Shamir-Adleman. The names Alice and Bob are used to represent two parties attempting to have secure communication. The name Eve is used for a third party trying to eavesdrop on private conversation. The name Mallory is used for a third party with malacious intent towards the conversation between Alice and Bob. This study would also employ the use of these fictional characters to enable easy understanding of cryptographic concepts.

4.2 Symmetric Key Cryptosystems

4.2.1 Horst Feistel's Lucifer cipher

Horst Feistel's work on encryption forms the basis for most modern encryption schemes. Feistel's scheme builds up on the older schemes of 'ideal block encryption'. His research into the ideal block encryption and his attempts to weed out its deficiencies lead to the proposal of the Lucifer cipher.

Ideal block encryption as Feistel referred to it, is the simplest form of block substitution cipher. It is a scheme where a n-bit input is transformed to a n-bit output. The function used to perform the operation is called the transform function. A one-to-one transform function can be devised that gives a unique result for each input block. The transform function must also have an inverse, so that scrambled data can be converted back to original. For example, let us consider a 3-bit to 3-bit block substitution (block size is 3). A 3-bit block can have $2^3 = 8$ values. One possible transform table could look like Table 4.1

Input Block Value	Output Block Value
0	1
1	7
2	0
3	3
4	4
5	6
6	2
7	5

Table 4.1: Simple Transform Table

A 3-bit sample like 110 with a decimal value of 6, gets transformed to 2 (from the table) which in binary is 010. A reverse transformation can be generated by reversing the above transform example. A preliminary cryptanalysis of this scheme brings about the following observations.

- This scheme is similar to substitution cipher for small block sizes and hence vulnerable to statistical analysis.
- Another concern is the key size. In the above example the table itself is a key. Consider that the same table can be used for both encryption and decryption (as they are reversible) then the key size in this particular case would be (6 bits) * 8 = 48 bits. Or in formulation for a n-bit to n-bit substitution $(2 * n) * 2^n$ bits.

In the same period of time, Shannon et al [23] had proposed the concepts diffusion and confusion to solve susceptibility to statitical cryptanalysis and large key sizes of robust encryption schemes.

- **Diffusion :** The mechanics by which plain text statistics is distorted/damaged by having every bit of plain text effect multiple cipher text bits.
- **Confusion :** The mechanics that would make is difficult to deduce the relation between the plain text statistics and the cipher text statistics.

With the scheme for block ciphers as the basis and also comments from Shannon about confusion and diffusion, Feistel introduced two new concepts.

- **Substitution**. The process by which plain text or plain text block is replaced by cipher text or block. Substitution is the logical equivalent of 'confusion' that Shannon et al [23] talked about. However, substitution alone does not add diffusion. Feistal, instead divides encryption process to rounds. Each round introduces substitution which is derived from the key and the round itself.
- **Permutation**. The process of permutation in rearrangement of bits. This further

confuses the relation between the plain text and cipher text. In Feistel's , the simplest permutation of interchanging the two halves of a block is done.

Describing Feistel's encryption becomes easier with the following description. The algorithm starts with taking the input plain text block. This block is then divided into two parts, conveniently called the left half (LH) and the right half (RH). The plain text then undergoes a series of repeated operations called rounds. In each round, the process of substitution and permutation are carried out to make the plain text jumbled.

- $LH_{p+1} = RH_p$

  In each p^{th} round, the right side of the text (RH_p) is set as the left side (LH_{p+1}) of the next round. This contributes to the permutation operation.

- $RH_{p+1} = LH_p \oplus RoundFunction(RH_p, K_p)$

  Each round has a derived round key K_p, which is derived from the encryption key. The derived key (K_p) along with the right half of the text (RH_p) are passed through a round function. This round function contributes for the substitution and is referred to as a S-box operation. As one can see, the formulas mentioned above are linear in nature, and given the nature of XOR operation, each round is reversible. Also removal of the round function, would still make the encryption scheme work, but it would just end up being a XOR encryption with only the benefit of permutations. So, the only operation that makes feistal cipher interesting is the S-boxes. It removes linearity from the system by introducing substitution. As solving linear equations is easier, cryptographers usually avoid their presence in the system. Forming this algorithm as the basis, DES was proposed as a encryption standard to the NIST encryption standard selection process.

4.2.2 Data Encryption Standard (DES)

DES is similar to Fiestel's encryption in many ways except for its implementation of S-Boxes. Similar to Feistel's, DES divides the plain text block into two halves. The right half becomes the left half in the next step and the left half is XORed with the right half and passed through a round function. The implementation of the round function is more concrete in the DES. The round function, which is also referred to as the S-box, takes care of the substitution process.
DES implements 8 S-boxes, one for each round. These are implemented as lookup tables which maps a 6-bit data to 4-bit data. The 2 most significant bits of the 6-bit data, form the column of the s-box and the last four bits form the row. Mapping to 4-bit is obtained at the intersection of the 2-bit column and the 4-bit row.
As the S-boxes are static, they have been an area of concern for cryptanalyst and have been heavily researched for back doors. Through out my research, I have found that the cryptanalysis of S-boxes serve valuable information to algorithm developers.
DES was broken with demonstrable products presented by EFF in 1998 [24] and COPACOBANA in 2006 [25]. Theoretical breakthroughs against DES started with the seminal work of Eli Biham and Adi Shamir published under the name of Differential cryptanalysis and could discover keys by using 2^{47} chosen plain texts. Other works by Mitsuru Matsui published as Linear cryptanalysis could discover keys by using 2^{42}know plain texts.

4.2.3 Advanced Encryption Standard (AES)

Advanced Encryption standard is the successor of DES and replaced it under the 2001 proposal by National Institute of Standards and Technology (NIST). AES is roughly similar to DES. It has a round function that incorporates permutation using sub-keys derived from the symmetric key. It has a substitution function which is implemented using S-boxes in a manner similar to DES. Each round function is repeated a predefined number of times to evenly scramble plain text input. In fact, the above set of similarities, are marked property of set of cryptosystems, collectively referred to as ”Iterated Cryptosystems”, of which AES and DES are part.

Stages of AES Encryption.

1. AES encryption starts by dividing the plain text into 128 bit block sizes (16, 8-bit bytes). Each 128 bit block is represented as a columnar 4x4 matrix which is referred to as the Input State.

2. The Input State is then bit-wise XORed with the sub key that is generated for this stage. Key management in AES is done by a key scheduling algorithm that generates 44 32-bit words, with each round getting a 4 word sub key.

3. The State then enters the first round function. Each round function is a sequence of following operations.

   - **Substitute byte :** An S box is used to make substitutions on the State matrix. In AES, S-box is a 16x16 matrix. Each byte in the State matrix is substituted with a byte in the S-box. For a byte S in State matrix, the 4 most significant bits of S mark the row in the S-box and the least significant 4-bits mark the column in the S-box. Unlike DES, where the S-box has fixed values, in AES, S-box values are precomputed as follows.
     In AES, the operations are all performed in a Galois Field of 2^8 represented as $GF(2^8)$ with the reducing polynomial for multiplication as $x^8+x^4+x^3+x+1$. A brief description of Galois Fields is warranted here. Both modular arithmetic and Finite Field theory (of which Galois Field is a part) are used in cryptography to bound mathematical operations to certain limits. In choosing $GF(2^8)$, for arithmetic operations, AES ensures that any operation on two 8-bit bytes would always yield an 8-bit byte as output. The 2 in $GF(2^8)$ is referred to as the characteristic of the field and 8 is the order. The characteristic signifies that all operations are to be performed modulo 2 and the maximum

degree of the polynomial is 8. In $GF(2^8)$, each byte is represented as a polynomial of form below.

$\Sigma a_i x^i$ where $0 \leq i \leq 8$

A byte 11001100 is represented as $x^8 + x^7 + x^4 + x^3$. The behavior of the 4 operations can be simplified. Addition and subtraction are same as XOR (as the operations are modulo 2).

$(x^8 + x^7 + x^4 + x^3)(+/-)(x^8 + x^5 + x^4 + x^3) = (x^5 + x^7)$

Division is similar to polynomial division with the additions carried out as XOR. Multiplication of two polynomials can cause resulting polynomials of degree as large as 16. This would not fit into a 8-bit byte. So, a reduction is done on polynomial using an irreducible polynomial in the $GF(2^8)$ field. AES uses $x^8 + x^4 + x^3 + x + 1$ (other irreducible polynomials exist in this field), which is used to divide the resultant of multiplication and the remainder is chosen as the result. Irreducibility is when the said polynomial is relatively prime to all other members of the field. The irreducible polynomial acts as modulo. So the multiplication of two polynomials a(x) and b(x) becomes

$a(x) * b(x)\ (mod\ (x^8 + x^4 + x^3 + x + 1))$.

- **Multiplicative Inverse :** Irreducible polynomials are also of interest when the need arises to determine a multiplicative inverse. Inverses are handy in reversing the operational direction, that is when trying to decrypt an encrypted byte. Two polynomials e(x) and d(y) are multiplicative inverse of each other over modulo m(x) when there exists two other polynomials v(x) and u(x) such that a(x).v(x) + b(x).u(x) = 1 (mod m(x)).

  Using the above as the base theory and considering that all following operations will be done in $GF(2^8)$, following is the AES S-box generation method. AES generates an S-box for a byte YX where Y is the row value (4-bit MSB) in the S-box and X is the column. AES then determines the inverse of YX

in $GF(2^8)$ with the chosen irreducible polynomial. Let the inverse be labeled b with each bit being $(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7)$. AES then applies an affine transformation over the byte b. Byte b is first multiplied with a 8x8 matrix (multiplication matrix), then the result is XORed with a 8x1 matrix representation of decimal number 99 to give the resultant *b'*. Both the matrices are fixed and have been chosen carefully to minimize cases where both S-box and inverse S-box map the same values. The output *b'* takes its place in the S-box. The inverse S-box is generated in a similar fashion. AES generates an IS-box for a byte YX where Y is the row value (4-bit MSB) in the S-box and X is the column. YX is multiplied with the inverse of the 8x8 multiplicative matrix and then XORed with the 8x1 matrix of decimal number 5.
S-box is used during encryption and IS-box is used during decryption.

- **Shift Rows :** Rows are cyclically left shifted, with the first row being untouched, the second shifted by one byte, the third by 2 bytes and the last row by 3 bytes. During decryption, rows are cyclically right shifted. Row shifting tries to remove the columnar relation between bytes.
- **Mixing Columns :** The state is further multiplied with a fixed transformation matrix. The transformation matrix's first row has values 2,3,1,1 and the rest of the rows are cyclically right shifted. The inverse mix is done using an inverse of the transformation matrix used in the forward encryption scenario. The inverse transformation matrix is multiplied with state. Again, all operations in this stage are performed under $GF(2^8)$.
- **Adding Round Key :** The state is XORed with the 128 bit round key, both in the forward and reverse case. The above set of 4 operations forms a round function in AES.

4. AES-128 uses 128 bit key with 10 rounds and 16 byte sub key, AES-192 uses 192 bit key with 12 rounds and 24 byte sub-key and AES-256 uses 256 bit key with 14

rounds and 32 byte sub key.

5. The final round is different from other round functions and does only substitution, shift rows and adding round key.

6. Decryption starts in the similar manner of performing the initial transformation of adding round key. This is followed with the round function. The order of operations in the round function are also different here. Each round performs row shift, followed by substitution, followed by adding round key and finally mixing of columns. Final round is also different in decryption with the order being shift rows, substitution and add rows key in that order. Key expansion is also reversed during decryption.

7. Key expansion for AES-128 would need 11 sub-keys of size 4 bytes, that is 44 bytes, from the main key. The first 16 bytes are derived directly from the key. For the next set of keys, expansion is done per word basis. For each new word is derived from the immediate previous word and the forth last word. If the word's position in the key is not a multiple of 4, the new word is a XOR of the last word and the 4th last word. If not, then the following is done. The last word is transformed in a left cyclic shift. The word is then substituted using a S-box. The resultant word is then XORed with a 4 byte counter (with the count value updated to the most significant word). After this operation, the counter (referred to as Rcon) is incremented.

4.3 Cipher Operational modes

One would have noticed by now that, if AES alone was used to encrypt a message, it would accomplish this by encrypting one block at a time. In cases where the plain text has a different size than multiples of the AES block size, padding techniques are applied. With such a set up, seeing a cipher text, an attacker can guess the type of encryption algorithm being used. Also, given a plain text block, AES always returns the same cipher text block. This information can be used by an attacker in crypt analysis. Information on computer systems are almost always structured to assist in easy translation of messages. This is done in form of Extensible Markup Language (XML), JavaScript Object Notation (JSON), Network packets, etc. Because of the structured nature of these formats, there are repetitions withing the message or in its descriptors. A cipher which produces the same cipher text for a given plain text, would leak information about the structure of the data. This is not desirable. Also, the knowledge of the structure of a cipher text for a given plain text, enables cryptanalysis to improve key discovery methods using techniques like differential analysis (which was very successful in defeating DES).

To overcome the above problems, several modes of operation were defined for symmetric key encryption algorithms. These modes of operations are not tightly coupled with any give symmetric cipher and any cipher technique can use these operational modes to prevent the above set of information leaks. The idea behind these modes is to make each encryption operation dependent on the previous operation, so that cipher text repetitions can be avoided. Following are the well know techniques in use today. All of these modes are supported on the Android mobile platform.

4.3.1 Cipher Block Chain (CBC) Mode

CBC was formulated to eliminate the problem of same plain texts generating same cipher text. This was achieved by performing XOR operation on the current plain text block with the previous cipher text block. Encryption and decryption in this mode can be represented using the formulas

$C_j = E(K, (P_j \oplus C_{j-1}))$

$P_j = D(K, C_j) \oplus C_{j-1}$

The implementation, however, is a bit different. This is because when deciphering, one

would not know what the previous plain text is. To solve this Initialization Vector (IV) was introduced, which is a randomly chosen number, which both the parties using the cipher system must know before hand.
Encryption : $C_1 = E(K, P_1 \oplus IV) and C_j = E(K, (P_j \oplus C_{j-1}))$
Decryption : $P_1 = D(K, C_1) \oplus IV and P_j = D(K, C_j) \oplus C_{j-1}$
One apparent problem with this system is that each plain text (not the plain text block) with always produce the same cipher text as long as the same IV is used. A way of solving this is using nonce (which are essentially counters which are either directly the result of message exchange or more universal counters like the clock). Another problem with CBC is that it leaks information about the type of cipher in use. The output is always multiples of the block sizes of the block chain.

4.3.2 Cipher Feedback Chain (CFC) Mode

Unlike the CBC, CFB is a stream cipher. In CFB, cryptographic operations are performed on each bit of data. Data is not divided into block of bytes to perform encryption. Because there is no padding, the output cipher text is the same size as the input plain text. In CFB following steps are followed for encryption.

1. Encryption starts with the selection of an IV. This IV is then encrypted with the key K to produce a set of bits, which are adjusted to the size of the plain text choosing Least Significant Bit (LSB)
2. The plain text is then XORed with the bits selected to produce the cipher text
3. This cipher text, then takes the role of IV in step (1) and process (1) through (2) are repeated till the plain text is exhausted

Decryption is performed as follows:

1. Decryption start with the IV. This IV is then encrypted with the key K to produce a set of bits, which are adjusted to the size of the plain text (choosing LSB)
2. Cipher text is XORed with the bits selected in step (1)
3. Cipher text then takes the role of IV in step (1) and the steps (1) through (2) and repeated till the cipher text is exhausted

4.3.3 Output Feedback (OFB) Mode

OFB is closer to a full key substitution cipher. In OFB, a key as large as the plain text is created, and then XOR this key to the plain text to produce a cipher text. In operation, this very large key is created using the following method.

1. OFB uses nonce instead of an IV. It encrypt the Nonce with the cipher key to produce an intermediate output.
2. The output of the previous stage is fed back to be encrypted with the key, to produce another set of output. This continues till the size limit of the plain text is reached.
3. All the outputs generated in the previous steps are appended, in the same order of their execution to generate a really large key with is XORed with the plain text.

Decryption occurs in a similar manner, where the generated key is XORed with the cipher text to produce the plain text. OFB uses a nonce here, because otherwise the same IV would produce the same cipher text for the same plain text, block by block. As this is one of the main goals that lead to the creation of these modes of operations, the nonce solution is the way to go. One advantage that OFB has is, error in one byte of plain text would not effect the entire cipher text. The error will be confined to the said block. However the same property also leads to bit substitution attacks of the cipher stream. For example if it is know that the first n bits of the message carry a message checksum (for integrity), an attack can modify these bits to fail the checksum and thus forcing the receiver to discard the message as invalid causing a denial of service attack.

4.3.4 Counter (CTR) Mode

CTR is similar to OFB, but instead of taking feedback from the previous output, it uses counters to generate a intermediate outputs. The encryption works as follows.

1. The counter is initialized to a random value.
2. This counter is encrypted with the encryption key to produce an intermediate key.
3. The counter is then incremented and the steps (1) to (2) are repeated. All the outputs generated in the previous steps are appended, in the same order of their execution to generate a really large key with is XORed with the plain text.

Decryption occurs in a similar manner, where the generated key is XORed with the cipher text to produce the plain text. CTR is often chosen for is easy implementation, fast operation and parallel generation of intermediate sub keys (as there is no dependence of previous sub key generation).

4.3.5 Galois Counter Mode (GCM)

This is a fairly new mode that is well supported on Android and as its name suggests is an extension of CTR mode with operations in Galois field. It improves upon CTR mode by adding authentication tags.

1. After the CTR mode produces the first cipher text block, a Galois Hash of an IV is XORed with the first cipher text block. Let us call this hash as GF_1
2. GF_1 is then XORed with cipher text 2 of the CTR output.
3. This feed back operation is repeated till all plain text blocks are exhausted.

Note that the above steps generate just the authentication tags. The actual cipher text is still generated using CTR mode with the CTR counter always initiated at 0. The output that is transmitted to the recipient contains the IV (used for generating authentication tag), the cipher text and the authentication tag. This takes care of message integrity.

4.4 Public Key Cryptography

Public key cryptography are radically different from symmetric key cryptography. Public key cryptography works with pairs of keys instead of a single secret key. This eliminates the need of key sharing which was necessary in symmetric cryptosystems. However, public key cryptosystems come with computational overhead which make them less likely candidates for encryption of large block of messages. As a result, public key cryptosystems are limited to key exchange and digital signatures. All crypto systems today, use key negotiation protocols that are handled by these public key systems and the message encryption carried out by AES-256 or other symmetric key cipher.

The basic structure of a public key crypto systems is as defined below.

1. Each party who wishes to participate in a public key crypto system must generate a pair of public and private keys.

2. The party then shares their public key with other participants or publishes it to a lookup server for public access.

There are two ways now, in which Bob and Alice (the aforementioned fictional characters) can have a secure conversation. In the first method public key is used to encrypt the message. It works in the following manner.

1. Let us say that Alice and Bob have both generated their key pairs and published their public keys.

2. If Bob wants to send a message to Alice, he would encrypt the message with Alice's public key and send it to Alice.
3. Alice can read the message by using her private key to decipher the message sent by Bob

In the second method private key is used to encrypt the message.

1. Bob sends message to Alice, by encrypting the message with his private key.
2. Alice can read the message by deciphering it with Bob's public key.

In the first method, it is impossible for Alice to figure out who sent the message. Anyone can pose as Bob and Alice would have no way of finding out. In the second method, Alice can determine with certainty that the message was indeed sent by Bob, as the message could have only been encrypted by Bob. The following sections discuss some well known public key crypto systems.

4.4.1 Rivest-Shamir-Adleman (RSA)

Unlike the older cypher systems, the basis of public key crypto systems is based on strong mathematical proofs in the field of modular arithmetic, field theory and prime numbers. RSA is based on following proofs. For integers X,E,D and N the following equation
$X^{ED}\,(mod\,N) = X$
holds when E and D are multiplicative inverse modulo $\varphi(N)$ (the Euler totient function). Given any two primes p and q, $\varphi(pq) = (p-1)(q-1)$, would create a group in modulo $\varphi(pq)$ where each element will have a multiplicative inverse. So one can formulate an equation where, if C is the cipher text, M is the message, e is the encryption key, d is the decryption key and n is $\varphi(pq)$ of two primes p and q, the following equation emerges
$M^{ed}\,(mod\,n) = (M^e)^d\,(mod\,n) = C^d\,(mod\,n) = M$

The above equations, hence, shows that encryption key is a multiplicative inverse of the decryption key and p and q are two very large prime numbers. Either one of e or d becomes a private key and the other the public key. Both the public keys and the number n are known publicly. The security of the RSA protocol hinges on the fact that it is very difficult to factorize n and determine the values of p and q, which would reveal the decryption key given the knowledge of encryption key.

4.4.2 Diffie Hellman (DH) Key exchange

DH key exchange depends on similar mathematical principles used in RSA algorithm. Following steps illustrate its operation.

- DH start with publicly know elements, a prime number q and a number αwhich is primitive root of q.
- Alice chooses a private key X_a and generates a public key Y_a using $Y_a = \alpha^{X_a}\ (mod\ q)$
- Bob uses the same operation to generate X_b and Y_b.
- Both exchange their public keys and can determine K using
  $K = (Y_a)^{X_b}\ (mod\ q) = \alpha^{X_a.X_b}\ (mod\ q) = (\alpha^{X_b})^{X_a}\ (mod\ q) = (Y_b)^{X_a}.\ (mod\ q)$

Unlike RSA, the security of DH depends on DLP (Discrete Logarithmic problems). In the above case, if an attacker wanted to compute X_a given that he knows Y_a, Y_b, α and q, then he would have to calculate
$Y_a = \alpha^{X_a}\ (mod\ q)$
$==> log_\alpha Y_a = X_a$ (under modulo arithmetic)
This is a difficult task and hence adds security to the system.

4.4.3 El Gamal Cryptosystem

El Gamal system also uses the same theory as DH and its security is based on difficulty of solving discrete logarithmic problem. This system, however is a complete message encryption systems and not just a key exchange system.

- El Gamal system start with publicly know elements, a prime number q and a number α which is primitive root of q.
- Alice selects a private key X_a less than $q-1$. She then calculates $Y_a = \alpha^{X_a}\ (mod\, q)$. The public key is a tuple of q, α, Y_a. This is then shared with Bob.
- Bob now needs to send the message M. He selects a random integer $k < q$. He calculates the key K using Alice's public key Y_a using
  $K = (Y_a)^k\ (mod\, q)$. He also calculates
  $C_1 = \alpha^k\ (mod\, q)$ and
  $C_2 = KM\ (mod\, q)$. He then transmits the pair C_1, C_2 to Alice.
- Alice gets
  $K = (C_1)^{X_a}\ (mod\, q) = (\alpha^k)^{X_a}\ (mod\, q) = (\alpha^{X_a})^k\ (mod\, q) = (Y_a)^k\ (mod\, q) = K$
  She then uses K to get the Message
  $M = (C_2K^{-1})\ (mod\, q) = KMK^{-1}\ (mod\, q)$.

4.4.4 Elliptic-Curve Diffie-Hellman (ECDH)

A newer member to the public crypto systems. It uses a mixture of Elliptic curve theory, Galois Field theory and modular arithmetic to perform key exchanges between parties. The difficulty of solving DLP is again used here in ECDH to provide security. Going in depth with the mathematics involved in this algorithm would take sizable portion of this document and is hence avoided.

4.5 Cryptographic Hash Functions

4.5.1 Message Digest

Message digests are one way hash functions that take large number of bits as input and generate fixed length outputs. Message digests are implemented in the same manner as encryption operation modes like CBC or CTR. However instead of returning outputs of the same size as input plain/cipher text, they return fixed length outputs. It can be of use in many situations.

- Can be used to determine message integrity. For finding another message that has the exact digest output as the original message is through brute force.
- Identification and versioning of files. File versioning systems like Git make use of message digests to track different version of files.
- Passwords are usually not stored in systems, instead their message digests are stored.

4.5.2 Digital Signatures

Digital Signatures are used to verify the source of information. If Alice sent a message, and Bob wanted to verify that the sender was indeed Alice, this could be achieved using digital signatures. Alice would create a message digest of the pain text, and encrypt the message digest with her private key. She would then send both the encrypted message digest and the encrypted message to Bob. Bob can then decipher the message using Alice's public key to validate both the authenticity and integrity of the message.

4.5.3 Message Authentication Code (MAC)

A common use of message digests is to authenticate messages. MAC functions are provided with a private key (to validate the source) and the message. The message is passed through a message digest and the output hash value of the digest is encrypted with the private key. This enables users to check the authenticity and integrity of the message at the same time.

Chapter 5

Secure Messaging applications

A fairly common form of communication is messaging through portable devices. Messaging could refer to SMS/MMS over telephony or Internet messaging. Information carried in messaging could be text or multimedia. It could carry the most benign form of conversation or could be a heated political debate in a messaging group. Exchange of banking details and Internet passwords using messaging is not uncommon. Given the value of information that passes through messaging applications, they are prime targets for attackers. Lately messaging applications have attracted interests from governments. Though the intentions behind governments tapping into messaging streams might be noble, it does violate user's privacy. With the resources which governments have at their disposal, along with the fact that many cryptographic schemes used today have been pioneered and promoted by them, many security experts are worried. As a result, lately most messaging applications are putting effort in securing their applications. It also provides us with an opportunity to promote awareness in secure information exchange.

It has been difficult to implement encrypted messaging (or any form of encrypted information exchange) as encryption schemes often conflict with user experience. To help people choose good messaging applications, the Electronic Frontier Foundation maintains a list of usable messaging applications and rates them based on the following 7 questions [26].

- Is the message encrypted in transit?
- Encrypted so the provider can read it?
- Can you verify contacts' identities?
- Are past communications secure if your keys are stolen?
- Is the code open to independent review?
- Is security design properly documented?
- Has there been any recent code audit?

With the help of the above list and considering the availability of source code and documentation, the following two applications were used to understand how modern cryptographic application function.

5.1 TextSecure

TextSecure is an open and federated messaging application managed by Open Whisper Systems [27]. A similar applications is available in Apple App Store by the name of Signal. TextSecure provides asynchronous end to end encryption with secure transfer of text, group messages and media files. The application passes all the set parameters by EFF and has been recommended by many security experts. The following section covers in detail the technologies used in TextSecure.

5.1.1 Operation

TextSecure initially provided support for SMS/MMS and IM. The support for SMS/MMS was dropped, however a fork of the project called SMSSecure was created to support SMS/MMS [28]. TextSecure now uses Google Cloud Messaging as communication medium. The application provides the following key features.

- Communication is possible with Signal users on Apple Appstore users.
- Allows local encryption of user messages.
- Forgets pass phrase after certain interval.
- Prevents screen shots.
- Automatically deletes old messages and conversation threads.
- Works seamlessly even when receivers do not use TextSecure.

When user uses it for the first time, the application provides the user with a public key which is 33 bytes P-256 Elliptic curve cryptography (ECC) key. Users can manually read and exchange the key or they can use QR code scanners to exchange each others keys. Once the keys are exchanged the applications can securely transmit information between them. If the receiver of the message is not an TextSecure or Signal user, the messages will be transmitted without encryption, hence providing a seamless data exchange.

The cryptographic system used in this application is a derivative of Off-the-Record (OTR) Messaging Protocol. OTR is used to provide an encryption scheme for Internet messaging. It assumes that Instant Messaging (IM) messages are in-order. It assumes that some messages might never be delivered and that attackers could perform denial of service attacks, but the secrecy of the message could never be compromised. It also provides forward secrecy by using ephemeral keys (more on this later).

Forward secrecy is said to have been implemented if compromised global keys does not compromise past session keys. Forward secrecy is used in IPSec as an optional feature, SSH, OTR, OpenSSL and Diffie-Hellman RSA. Forward secrecy in OTR is achieved using Ephemeral key exchanges. In Ephemeral key exchange, intermediate messages are encrypted using fresh set of keys (which are derivative of the main public key). Once the conversation is over, there temporary keys are discarded. As these messages have been encrypted using ephemeral keys, exposure of public keys would not compromise past

messages. This in contrast with Pretty Good Privacy (PGP) which is a popular encryption scheme provided by email service providers. PGP does not implement forward secrecy. All messages in PGP are encrypted with one key and compromise of global key can compromise every message ever encrypted using it.
While OTR is a novel scheme, it's failure to provide asynchronous message transfer prompted TextSecure to borrow features from another cryptographic scheme called the SCIMP Ratchet. The following sections discuss OTR and SCIMP ratchet and how TextSecure reaches a middle ground to meet the application's requirement of asynchronous messaging with forward secrecy.

5.1.2 OTR

The OTR [29] technology version 3.4 ensures secrecy in IM by assuming in-order messages. It uses SIGMA [30] protocol as the Authenticated Key Exchange (AKE). AKE is the mechanism by which parties involved in cryptograpic communication can securely exchange keys and authenticate identity of each other. SIGMA uses Diffie-Hellman (DH) as the basis and improves upon it using AES and hash functions. Before going into AKE scheme, an introduction to nomenclature is provided. From here on, it is considered that DH mean Diffie-Hellman, g as the generator of a 1536-bit prime number and all modulo operations are done on that prime. pub_A and pub_B are public keys of Alice and Bob.
The following are the AKE steps

1. Bob picks a 128-bit random number *r* and at least 320-bits of another random number *x*. Unlike the usual DH where Bob would just pick a random number and raise it to the power of g and sends it's modulo over the network, in this case Bob sends $AES_r(g^x)$ and $HASH(g^x)$. Both the sender and receiver use the same HASH function. This enables g^x to be sent securely and also be verifiable by Alice.

2. Alice then picks up her own random number *y* (at least 320 bits) and sends Bob g^y.

3. Bob calculates $s = (g^y)^x$. This is his DH secret key for the messages. Here is where OTR departs away from the traditional DH in a more radical manner. Bob then generates two keys using AES, lets say *c* and *c'*. He also calculates 4 Message Authentication Code (MAC) keys *m1, m1', m2, m2'*. Picks a serial number $keyid_B$ for his public key g^x. He then computes
   $M_B = MAC_{m1}(g^x, g^y, pub_B, keyid_B)$ and $X_B = pub_B, keyid_B, sig_B(M_B)$ and sends $r, AES_c(X_B), MAC_{m2}(AES_c(X_B))$ to Alice.
   The M_B value is a function of Bob's public key (which is global), has a serial ID for the session.

4. Alice uses *r* to decrypt the $AES_r(g^x)$ sent in step 1 and obtain g^x. Then she runs the HASH function on g^x to determine its integrity using the hash value from step 1. She then computes her $s = (g^x)^y$. Using the exact same algorithm as Bob, she computes two keys using AES, *c* and *c'* and 4 MAC keys *m1, m1', m2, m2'*. Using the values *c* and *m2* she can decipher the AES and match hashes to extract and verify the data sent by Bob in step 3. This would in the end provide Alice with the $pub_B, keyid_B, sig_B(M_B)$. Now Alice has the secret key, the public key, the serial id and uses the signature proof of a successful key exchange by verifying $sig_B(M_B)$ with pub_B.
   Alice must then generate $AES_{c'}(X_A), MAC'_{m2}(AES_{c'}(X_A))$ to be sent to Bob so that he can verify the keys provided by Alice.

5. Bob uses *m2'* to verify $MAC_{m2'}(AES_{c'}(X_A))$ and *c'* to decrypt $AES_{c'}(X_A)$ to obtain $X_A = pub_A, keyid_A, sig_A(M_A)$ and can verify $sig_A(M_A)$ with pub_A.

Both Alice and Bob used the $keyId_A$ and $keyId_B$ parameters. These parameters indeed separate OTR from other key exchange implementations. When Alice is calculating secret keys using bob's recent key and her own key, she also chooses her next key and passes it with the encrypted message that Alice sends to Bob. The message also contains a new

serial id for the new key. Bob deciphers the received message, finds the new key within and sends an acknowledgement to Alice. In the next message Alice can use her new key to encrypt the message and the older keys can be discarded. Hence, the system has constantly changing cryptographic keys which are short lived and are termed as ephemeral keys. Serial id's also help in distinguishing different clients used by same users.
The dependence of OTR on synchronous messaging systems poses problems with SMS/MMS systems where conversations last longer, unlike IM. Even in case of IM, Alice's keys are updated only when Bob acknowledges message reception. If Bob fails to reply to messages by Alice, her key would be valid till he does reply. It also provides for a very poor user experience in systems like iOS, where messaging applications have access to messaging system only once the user has read the message. This prevents systems from leveraging the ephemeral key exchange of OTR with some messeging applications falling back to PGP and S/MIME to handle asynchronous messaging [31] and all together abandoning forward secrecy.

5.1.3 SCIMP Ratchet

Silent Circle Instant Messaging Protocol (SCIMP) [32] was developed by Simple Circle group for their Silent Text IM application. It is a synchronous IM protocol where the keys are changed with every message based on predefined hash function. This provides perfect forward secrecy. A high level explanation of the steps involved is provided below.

1. Alice and Bob set of a secure channel using the first set of DH keys.

2. Alice sends the first message by encrypting it using her first DH-1 key. Bob decrypts the message and responds with a message and an acknowledgement.

3. Alice and Bob generate next set of DH keys by hashing their previous secret key. Both of them use the same hashing scheme.

4. Alice or Bob can now send an encrypted message using the DH-2 key.

According to Textsecure developers, this scheme has following drawbacks.

1. When messages are out of sequence (which can be common in SMS/MMS and rare in IM), users cannot discard their older hash keys. Schemes of determining life time of unused old hash keys would be problematic.

2. SCIMP provides optimal forward secrecy, however compromise of anyone of the older hash keys could compromise the system as the hash function generating next key is known. Hence it fails to provide future secrecy.

5.1.4 The TextSecure Way

TextSecure finds a middle ground to the above cryptographic schemes. It tries to achieve optimal forward secrecy of SCIMP and future secrecy of OTR with as little negatives as possible [33]. The innovative new method used in TextSecure is credited to Trevor Perrin. Problems and proposed solutions from both the above techniques are illustrated below.

1. The parameters used in OTR prevents man in the middle attacks as each message contains next key id's, thus allowing for traceability and detection of tampered messages. This however increase book keeping and also adds additional steps to key exchange which the authors of TextSecure wanted to minimize. They wanted to reduce this by eliminating the "advertise" step (where the message also contains the next key) of OTR. To reduce this step, TextSecure derives a RootKey in the initial handshake. This is then re-derived from every subsequent DH handshake. This would eliminate the "advertise" phase. Alice can then send a message to Bob by generating a key derived from root key without waiting for Bob to acknowledge the key.

2. The immediate forward secrecy of SCIMP can be mixed into the above method by providing hash iteration within each DH ratchet click. What this means is that for a given DH key pair, any messages sent before either Alice or Bob moves to the

next Rootkey, the messages would be encrypted with a key that is a derivative of the RootKey and SCIMP hash iteration.

3. The application also uses Elliptic-Curve Diffie-Hellman (ECDH) (where the generator class is an elliptic curve on a Cartesian system).

4. Figure 5.1, better explains the ephemeral key generation in the new TextSecure messaging model. From Alice's perspective, the key pairs that she processes for each ECDH exchange is represented as ECDH (An, Bn) where An is Alice's RootKey on a particular ECDH ratchet (instance) and Bn is Bob's key. n represents the nth RootKey generated by either Alice or Bob. The branches with node of the form CK-An-Bn represent the sub ratchet and it comes up in cases where a party has sent more than one message before response from the other. For example under the node CK-A1-B0, Alice sent 3 messages and the keys used for these message are shown as MK-0,1,2. These sub-chain keys are hash iterated with the first key MK-0 derived from the RootKey ECDH (A1-B0). Thus, the new implementation ensures that Alice can generate ephemeral keys and that these keys can still be traced back to the RootKey (prevent MITM attacks) while providing the freedom to send future secure messages.

5.2 ChatSecure and Orbot

Chat Secure is an open source OTR encryption application which enables messaging over XMPP protocol.

- It supports XMPP with TLS certificate pinning.
- It operates on OTR.
- It makes use of The Onion Router (TOR) network.

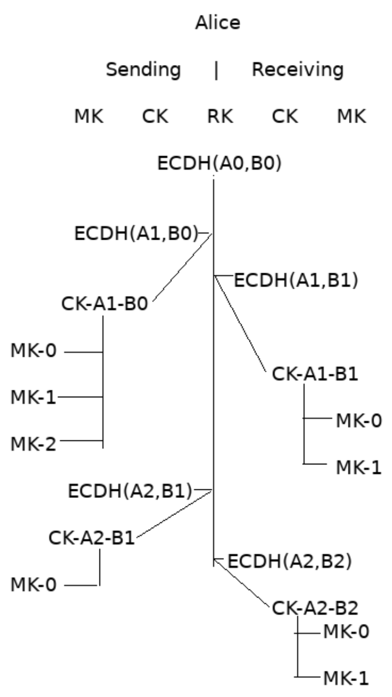


Figure 5.1: Key Generation from Alice's perspective

- It provides encryption of application data through SqlCipher [34].

The most interesting feature in the application is the use of The Onion Router (TOR) to route traffic anonymously. Access to the Tor network is available through its own browser (a modified version of Firefox that integrates well with the tor client). On Android, the Tor project has been ported as Orbot. The Tor project consists of volunteer across the Internet that maintain a network of systems called tor-relays which communicate with each other using Virtual Private Network (VPN) tunneling. When a user uses Orbot or any other Tor client, they get access to this virtual network. As the information is passed between the devices and the Tor relays, data is encrypted in the application layer along with the source and destination. The Tor relays, further randomly select peers to create a virtual circuit with each relay adding its own layer of encryption. This provides for a secure and anonymous network which can be used to gain privacy on the Internet.

The use of Orbot, along with the OTR encryption technology provides for a very secure application. According to the EFF secure messaging scorecard, chatsecure with orbot passes all the important security questions. ChatSecure however does not support SM-S/MMS. In fact many Messaging application have either dropped or moved away from SMS/MMS security. The prime reason for this could be that Telephony channels are no more considered to be secure. With devices readily available [35] in the market which could act as false telephony base stations and siphon user calls and data, it makes the telecommunication channels highly unreliable for secrecy. Also with news of Telecom providers readily siphoning their data to government agencies is disconcerting [36].

Chapter 6

A New Messaging Solution

Trusted applications in cryptography lately, have moved towards open frameworks. Applications that can be easily audited, whose underlying implementation is in the public domain are common now. These applications are all supported by EFF and the open source community. Such applications are open to scrutiny by security experts all over the world and accept new proposals of improvements accompanied with quick peer reviewed patches. This is in sharp contrast to close sourced applications where peer reviews on large scale is not possible and vulnerabilities can go unnoticed for long periods of time.

6.1 Goals

Cryptographic applications however have not yet reached mainstream acceptance. The most common problem being that cryptographic schemes interfere with user experience. However, problems with user experience have been demonstrably solved by applications like TextSecure. The more important problem with cryptographic applications, is the lack of understanding of their concepts among less technical users. As a result most users shun away from these applications.

Another factor, which most cryptographers would ignore, as it does not go well with Kerckhoffs's principle [37]. Kerckhoffs principle states that ”A cryptosystem should be secure even if everything about the system, except the key, is public knowledge". The concept of allowing the system internals be know to public has many advantages. However, it also has disadvantage of enabling cryptanalyst, with malicious intent, to be able to reverse engineer solutions.

My proposal is to design an application that meets the following design goals.

1. 1. Provide a messaging application, that allows novice users with the ability to write their own cryptographic schemes, in an easy, learn-able scripting language.

2. Provide advanced programmers with tools to write complex cryptographic schemes. The idea being that, as more developers get to play with cryptographic schemes on messaging applications, it would generate a collection of schemes which could be open or close sourced based on the discretion of the user. It would creates a platform with multiple encryption schemes, with the possibility of overwhelming cryptanalysts trying to break encryption systems.

6.2 Design

Design specification for secure messaging solution can be better explained by dividing it into smaller components. The following section starts the discussion by explaining the choice of the application platform. It then presents a discussion on the medium of key exchange. Thirdly, it discuss the medium of message exchange. This is followed with a discussion on the choice of a scripting language that the users would need to know to write their encryption schemes.

6.2.1 Platform

The platform of choice for the application is Android. This choice is dependent on the following factors.

1. The Android platform provides better support for messaging application than iOS. The security implementation on iOS does not give access to custom SMS applications unless the user has acknowledged the reception of message from the home screen notification. This would prove a hassle in case of cryptographic schemes which perform key exchange handshakes. A key exchange in cryptography usually involves sending at least a couple of messages to and fro, which would hamper user experience.

2. Android provides great support for NFC technology that has only recently been introduced on iOS.

3. AndroidSE and its application sandbox are well understood security technologies. This could assist future development/enhancement if additional security features were felt necessary.

4. Rooting on Android allows applications to exploit system level features of Linux which could prove useful for creating custom hash functions.

5. ROM modification communities in Android provide a good platform when testing experimental applications and can serve as a good community feedback platform.

6.2.2 Medium of Key Exchange

For the application, the default channel for exchanging cryptographic schemes and keys is NFC. NFC allows information to be exchanged in very close proximity. The device has to be in constant contact for data transfer. This eliminates any kind of Man in the middle (MIMT) attack. One could argue that QR codes would be a better (and more common, similar to TextSecure), but possibilities presented by NFC seemed interesting. Also, QR codes can be copied by taking a picture of the QR code from another phone, where as the same thing cannot be done to NFC tags.

6.2.3 Medium of Data Exchange

Initially, the idea was to develop a solution for SMS messaging, and in fact the proof-of-concept application uses SMS as the Medium of data exchange. But during the case study of Messaging applications, it was concluded that the final application must support Instant Messaging (IM). The user would be provided with the option to send message over IM or SMS. In case of IM, it was decided that Google Cloud Messaging would be the first supported platform and later expansion to other messaging platforms would be done based on user demands.

When sending data over certain mediums, one must keep in mind the character encoding supported on the medium. For example, SMS on GSM networks follows the GSM 03.38, where the default encoding uses 7 bit alphabets, which limits SMS messages to 160 characters. When using an encoding of 8-bit, the maximum character limit per message falls down to 140 characters. The encoding also effects the recognizable characters. Wrongly encoded messages would not be decoded correctly and could break user defined cryptographic schemes.

Similar, design considerations must be taken when sending messages over IM where the encoding could be either UTF-8, UTF-16 or UTF-32. It would be better to provide users with the option to select the encoding, but only in cases of advanced users who have better understanding of encoding schemes. With novice users, the application must take care of the encoding by itself.

6.2.4 Scripting Language

For the application, I researched the following scripting languages and weighed on their pros and cons to come up with a suitable choice. Android does not directly support any scripting languages that can be run from inside the application. The need for this feature has however been felt by some developers prompting them to try incorporating this feature into Android.

1. Scripting Layer for Android (SL4A) : This project is hosted on Git-hub and active development has stopped on the project. The project's goals was to add support for scripting languages Python, Perl, JRuby, Lua, BeanShell, JavaScript, Tcl, and shell. However as the project was discontinued, I chose to not use this project.

2. Qpyton : It provides a platform to run python code inside Android application. I however did not choose this, as I was looking for an scripting language closer to Java.

3. Groovy : I finally chose groovy as the language to support scripting of cryptographic schemes. Groovy seemed interesting, as it can benefit directly from the cryptographic libraries available on Android platform. Groovy was originally developed to provide better scripting on Java and today has branched into its own language that can be compiled to run on Oralce JVM.

Choosing groovy, however posed a problem. Groovy code does not compile to ART VM. Compiled groovy scripts generate byte code that is compatible with Oracel JVM or OpenJDK VM. This meant that the groovy code the users would write had to be ported to ART VM compatible byte code. Searching solutions for the problem, led to a solution by Cedric Champeau [38], who has been working on a project which would enable Android application development in groovy. His project grooidshell-example on Git-hub contained the key features which I needed to transform the groovy scripts written by users into ART VM compatible code. The procedure is documented in the code added in this thesis's appendix.

6.2.5 Contact Management

In the proposed application, the user can either write their own cryptographic scheme, or they could be utilizing a scheme that has been shared with them by a contact. This creates a situation where a user would have different schemes available for different contacts.

This called for better management of schemes.

Having a scheme stored for every contact on the user's phone is not ideal. Instead, the application would maintain a separate database table for schemes to handle redundancy. Each scheme is identified by an id. The application would have a second table of contacts, with a column for scheme id. When a new scheme is available, either through user's input in "*Algorithm Code*" or from being shared through another user, the scheme would be checked with available schemes in the database. If the new scheme matches an already existing scheme, no new entries would be made in the scheme table, instead the scheme id of the existing scheme would be added to the contact table for the corresponding contact. This would save space and eliminate data redundancy.

The check for scheme match could be done either by doing a direct string compare, or by maintaining message digest hash of the available schemes with the hash of the newly available scheme.

6.2.6 Data Encryption

The tables in the database would be encrypted. One can use readily available libraries like SqlCrypt. Further encryption could be provided by prompting user to enable device encryption on the Android device. This would add to the application's and user's data security.

6.2.7 Orbot

Applications that want to use Tor on Android must also install Orbot. The Netcipher project available under '*guardianproject/NetCipher*' on Git-hub, provides libraries that can be included into an Android project. Android applications can then access the Orbot services hosted by the Orbot applications using the proxies libraries available in Netcipher project and connecting over *HTTP Proxy localhost 8118 SOCKS 45 Proxy localhost 9050*. NetCipher also provides libraries that can be used to prompt the user to install

Orbot if the Orbot service is missing on the Android system.

Chapter 7

Proof Of Concept

The proof of concept application, is named as CryptoNoob, tries to check the viability of the following solutions.

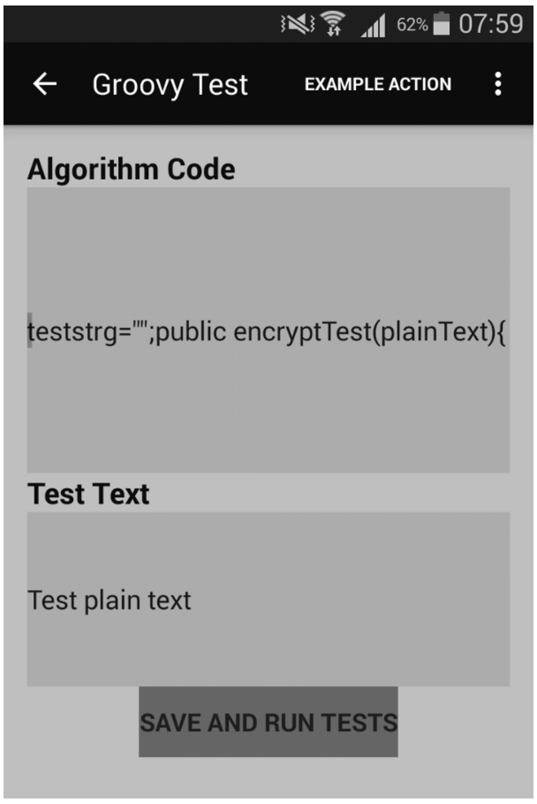
1. Viability of NFC in cryptographic scheme and key exchange.

2. Viability of allowing user defined cryptographic schemes.

7.1 Walkthrough

The application follows the recommended Drawer Pattern designed by Google for Android. If you are familiar with the Gmail Application on Android KitKat, it is the pop out drawer on the left with options that the users can choose, with each option providing the user with a new view of the application. CryptoNoob application implements a similar drawer with the following four options.

1. Groovy Scripts : The Groovy Script navigation drawer is the part of the application where a user can enter their custom cryptographic schemes and test them. It contains a text box titled "*Algorithm Code*" where a user can enter the algorithm code. Algorithm code must be written in Groovy language. It must implement two functions/methods called *encryptTest* and *decryptTest*. Both the methods take one String parameter as input. By default the application comes with a simple Ceaser Shift Cipher, that the users can use an example. The second text area names "*Test String*" is where a user can enter some test string and check the validity of their cryptographic scheme by pressing the "*Save and Run Test*" button. When this button is pressed, the application takes the text from the "*Test String*" text area and passes it as an input parameter to the *encryptTest* method defined in cryptographic scheme. The output of *encryptTest* method is then passed a s a parameter to the *decryptTest* method. If the text in "*Test String*" is same as the output of the *decryptTest* method, the application assumes that the messages are encrypting and decrypting fine. The cipher text of the message/string in "Test String" message box is copied to clipboard and the encryption scheme (the groovy script in "*Algorithm Code*" is saved to the database.). The user can now open the SMS application of his choice on the device, paste the cipher text from Android clipboard and be able to send an encrypted message.



```
teststrg="";public encryptTest(plainText){
```

Test plain text

Figure 7.1: Application View : Groovy Scripts

2. NFC Initiate : NFC Initiate contains a "*NFC Initiate*" button. On pressing the button, the application creates a NFC tag containing the cryptographic algorithm stored and tested in *Groovy Scripts→ Algorithm Code*. Once the NFC Initiate button is pressed, the device can be brought in contact with any NFC enabled Android device with CryptoNoob installed on it and the message stored in NFC Initiator device will be transferred to the NFC Receiver device.

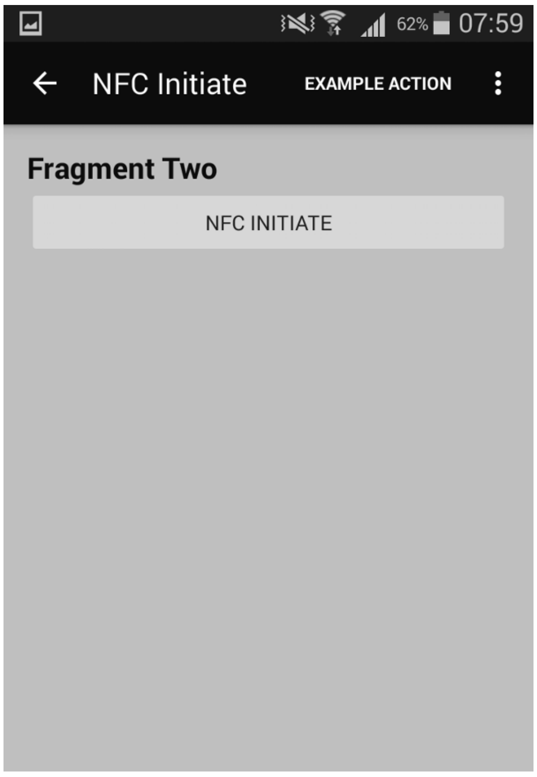
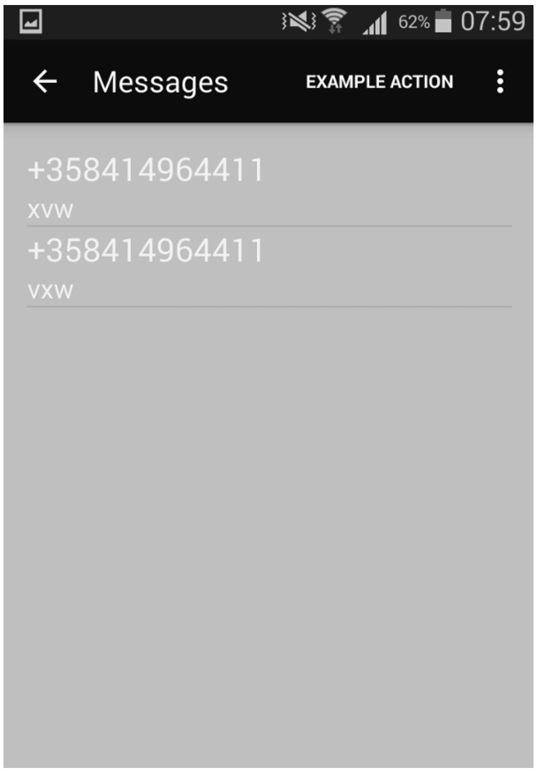


Figure 7.2: Application View : NFC Initiate

3. NFC Respond : A drawer, designed to confirm that the application has in-fact received a new cryptographic scheme from the NFC initiator. Once the NFC receiver device detects the NFC tags sent from a CryptoNoob application, it prompts the user to accept or decline the NFC tag. If the NFC tag is accepted, the new scheme is saved into the receiver devices' database.

4. Messages : When Android devices with CrytoNoob installed on them, receive an encrypted message, CryptoNoob recognizes the message and displays a notification in the Android notification bar saying that a new encrypted message has been received. The user can view all their past messages decrypted in this navigation drawer by either navigating to it from the CryptoNoob application or by taping the notification item in the notification bar.



+358414964411
xvw
+358414964411
vxw

Figure 7.3: Application View : NFC Respond

7.2 Observations

With the application, I was able to prove that scripting language **can** be embedded into the Android application, and that they are capable of performing encryption and decryption of data. The application also proved the ability of NFC to act as a good medium for exchange of keys and cryptographic schemes.

There is one concern with the application. The performance of the application is not very impressive. The process of recompiling Groovy script to JVM code, then to ART VM compatible code causes noticeable lag in the application. Right now the application compiles the Groovy script on every incoming message. This is not ideal. One immediate solution would be cache the compiled dex files into the applications data directory and associate each dex file with its Groovy code. This would help eliminate the problem of recompiling scripts for every request. For improvements on the compilation process itself would require a deeper study of the ART VM byte code.

Another, more radical method of improving performance could be through the use of rooted Android ROM. Where one could install scripting languages onto the Linux kernel (probably python). Then all user scripts can be run natively, improving performance. This would effect the usability of the application which is counter objective to making a user friendly applications for novice users.

Chapter 8

Conclusion

The bulk of the work in this thesis was directed towards attaining implementation level knowledge of the initial problem and it's proposed solution. As a result, the thesis covered the Android architecture and NFC in great lengths. To develop a messaging application, it was very important to understand existing secure messaging applications. Considerable time was hence spent on TextSecure and ChatSecure. The thesis examined the basics of Android Security architecture and in the process SELinux. It discussed some practical cryptographic systems which are being used today by numerous applications. Availability of resources regarding these topics in public domain proved very helpful in understanding the various cryptographic systems.

The goal of the thesis was to provide a solution that has the possibility to provide a platform for people to share and understand different cryptographic schemes. To achieve this, the thesis proposed an application that was easy to use and provided basic security for messaging through cryptographic schemes implemented by users.

The thesis, was able to prove key concepts that would be helpful in refining the solution further in the future. It was able to prove with a working example that, one can have user scripts run successfully on Android platform. That these user scripts have the possibility of using both the Android libraries and Linux system libraries to enable users in creating cryptographic schemes. It was able to successfully demonstrate that one can share cryptographic schemes over the NFC medium.

The thesis recognizes that there are certain areas that would need work in the future, specifically the performance overhead that comes with running scripts that do not have any kind of support on the Android platform.

Future goals of this project would be following.

1. Set up a web portal where user's can share/discuss new cryptographic schemes.

2. Set up a web application where user's can test new schemes before moving them to their devices.

3. Find better methods to improve the present performance issues that have been discussed under section 7.2.

References

[1] H. T. Al-Rayes, "Studying main differences between android and linux operating systems," (accessed 2019-05-22). [Online]. Available: http://www.ijens.org/vol_12_i_05/128005-4747-ijecs-ijens.pdf

[2] E. Linux. Wiki, "Android booting," (accessed 2019-05-22). [Online]. Available: http://elinux.org/Android_Booting

[3] Google. Inc, "Android Init language," (accessed 2019-05-22). [Online]. Available: https://android.googlesource.com/platform/system/core/+/master/init/README.md

[4] Open. Group. Base. Specification. IEEE, "Opengroup," (accessed 2019-05-22). [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/

[5] Santa. Cruz. Operations. Inc. AT&T, "System v, application binary interface," (accessed 2019-05-22). [Online]. Available: http://www.sco.com/developers/devspecs/gabi41.pdf

[6] E. Linux. Wiki, "Android zygote startup," (accessed 2019-05-22). [Online]. Available: http://elinux.org/Android_Zygote_Startup

[7] Department. Of. Defense. STANDARD, "Department of defense trusted computer system evaluation criteria," (accessed 2019-05-22). [Online]. Available: https://web.archive.org/web/20060527214348/http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html

[8] Palm. Source, "Open binders," (accessed 2019-05-22). [Online]. Available: http://www.angryredplanet.com/ hackbod/openbinder/docs/html/

[9] ClockworkMod, "Clockworkmod recovery software," (accessed 2019-05-22). [Online]. Available: https://www.clockworkmod.com/

[10] P. Singh, "Source page for agni rom," (accessed 2019-05-22). [Online]. Available: https://github.com/psndna88/AGNI-pureSTOCK

[11] Google. Inc, "Verified boot," (accessed 2019-05-22). [Online]. Available: https://source.android.com/devices/tech/security/verifiedboot/verified-boot.html

[12] GlobalPlatform, "Trusted execution environment," (accessed 2019-05-22). [Online]. Available: https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf

[13] Google. Inc, "Verified boot 2," (accessed 2019-05-22). [Online]. Available: http://source.android.com/devices/tech/security/verifiedboot/

[14] Google. Inc, "Crypt-DM," (accessed 2019-05-22). [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt

[15] S. Smalley and R. Craig, "Security Enhanced (SE) android: Bringing flexible mac to android," (accessed 2019-05-22).

[16] R. Haines, *The SELinux Notebook*. http://www.gnu.org/: GNUFree Documentation, 2014.

[17] S. Smalley, "Security Enhancements (SE) for android," (accessed 2019-05-22). [Online]. Available: https://events.linuxfoundation.org/sites/events/files/slides/abs2014_seforandroid_smalley.pdf

[18] S. Corporation, “Felica,” (accessed 2019-05-22). [Online]. Available: http://www.sony.net/Products/felica/

[19] ETSI, “Smart cards; UICC - contactless front-end (clf) interface; part 1: Physical and data link layer characteristics,” (accessed 2019-05-22). [Online]. Available: http://www.etsi.org/deliver/etsi_ts/102600_102699/102613/07.03.00_60/ts_102613v070300p.pdf

[20] NFC.ORG, “Near field communication home page,” (accessed 2019-05-22). [Online]. Available: http://www.nearfieldcommunication.org/

[21] ISO.ORG, “ISL/IEC 18000-3:2010 : Information technology – radio frequency identification for item management – part 3: Parameters for air interface communications at 13,56 mhz,” (accessed 2019-05-22). [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53424

[22] D. M. Gebhart, “Loop antenna,” (Course Material for University of Applied Sciences Upper Austria, RF Basics and Components, Course 3, Unit 3rd. accessed 2019-05-22). [Online]. Available: http://rfid-systems.at/03_Loop_Antennas.pdf

[23] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. CRC PRESS, Boca Raton, Florida: CRC PRESS, 2007.

[24] Electronic Frontier Foundation, “Cracking DES: Secrets of encryption research, wiretap politics, and chip design.” [Online]. Available: http://cryptome.org/jya/cracking-des/cracking-des.htm

[25] J. Pelzi. G. Pfeiffer. Sandeep Kumar, Christof Paar and M. Schimmler, “Breaking ciphers with copacobana,” *ACM DL*, October 2006.

[26] Electronic Frontier Foundation, “Secure messaging scorecard,” (accessed 2019-05-22). [Online]. Available: https://www.eff.org/secure-messaging-scorecard

[27] Open. Whisper. Systems, “Open whisper systems,” (accessed 2019-05-22). [Online]. Available: https://whispersystems.org

[28] B. L. Querrec and C. Metcalfe, “Sms secure and fork of textsecure,” (accessed 2019-05-22). [Online]. Available: https://github.com/SMSSecure/SMSSecure

[29] “Off-the-record messaging protocol version 3,” (accessed 2019-05-22). [Online]. Available: https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html

[30] H. Krawczykt, “Sigma: the sign-and-mac approach to authenticated diffie-hellman and its use in the ike protocols,” (accessed 2019-05-22). [Online]. Available: http://webee.technion.ac.il/ hugo/sigma-pdf.pdf

[31] M. Marlinspike, “Forward secrecy for asynchronous messages,” (accessed 2019-05-22). [Online]. Available: https://whispersystems.org/blog/asynchronous-security/

[32] P. Zimmermann. Vinnie Moscaritolo, Gary Belvin, “Silent circle instant message protocol,” (accessed 2019-05-22). [Online]. Available: https://cdn.netzpolitik.org/wp-upload/SCIMP-paper.pdf

[33] M. Marlinspike, “Advanced cryptographic ratcheting,” (accessed 2019-05-22). [Online]. Available: /https://whispersystems.org/blog/advanced-ratcheting/

[34] “Sql cipher,” (accessed 2019-05-22). [Online]. Available: https://www.zetetic.net/sqlcipher/documentation/

[35] R. Gallagher, “Meet the machines that steal your phone’s data,” (accessed 2019-05-22). [Online]. Available: http://arstechnica.com/tech-policy/2013/09/meet-the-machines-that-steal-your-phones-data/

[36] J. Sanchez, “Tech giants named in prism want to see an nsa transparency report,” (accessed 2019-05-22). [Online]. Avail-

able: http://arstechnica.com/tech-policy/2013/07/tech-giants-named-in-prism-want-to-see-an-nsa-transparency-report/

[37] A. Kerckhoffs, “Kerckhoffs principle,” (accessed 2019-05-22). [Online]. Available: https://www.petitcolas.net/kerckhoffs/index.html

[38] C. Champeau, “Groovy shell in android,” (accessed 2019-05-22). [Online]. Available: https://github.com/melix/grooidshell-example

Appendix A

Building and Installing the Application

The source of the application can be found at the Git-hub location

```
https://github.com/flyingarg/CryptoNoob
```

The code available on GitHub contains Android Java files and build file. The target apk can be built using the following steps. 1. Install git, Oracle JDK 7 and gradle version 2.4 or above.
2. Clone the project using the following command.

```
git clone https://github.com/flyingarg/CryptoNoob.git
```

3. Enter the cloned directory and build it using ”gradle assemble”

```
cd CryptoNoob
gradle assemble
```

4. Once the build completes, the application can be found in the location.

```
cd CryptoNoob/app/build/outputs/apk
```

5. The ”app-debug.apk” can be copied to a Android device with minimum version 4.2.2 and installed by clicking app-debug.apk through a file manager.

Appendix B

Source Code

```
public class MainActivity extends BaseActivity{

  private String tempStoreAlgo = "";

  @Override
    protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setTitle(R.string.app_name);
      setContentView(R.layout.activity_main);
      mTitle = getTitle();
      navigationDrawerFragment = new NavigationDrawerFragment();
      Bundle argsNDF = new Bundle();
      argsNDF.putInt("position", 0);
      navigationDrawerFragment.setArguments(getIntent().getExtras());
      navigationDrawerFragment.setArguments(argsNDF);
      getSupportFragmentManager().beginTransaction().add(R.id.fragment_co
         navigationDrawerFragment).addToBackStack(null).commit();
    }
```

```
public void onNavigationDrawerItemSelected(int position) {
  Log.i("rajumoh", "MainActivity position : " + position);
  super.onNavigationDrawerItemSelected(position);
}

@Override
  public View createRootView(LayoutInflater inflater,
     ViewGroup container, Bundle savedInstanceState) {
    final View rootView =
       inflater.inflate(R.layout.fragment_1, container,
       false);
    EditText algoSave =
       (EditText)rootView.findViewById(R.id.algo_save);
    EditText testTextSave =
       (EditText)rootView.findViewById(R.id.test_text_save);
    tempStoreAlgo = DatabaseUtils.getAlgoFromDb(null,
       getApplicationContext());
    algoSave.setText(tempStoreAlgo);
    testTextSave.setText("Test plain text");
    Button runTest =
       (Button)rootView.findViewById(R.id.run_test);
    runTest.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
        EditText algoSave = (EditText)
           rootView.findViewById(R.id.algo_save);
        EditText testTextSave = (EditText)
           rootView.findViewById(R.id.test_text_save);
        String encryptTest = "encryptTest(testString);\n";
        String encDecTest =
           "decryptTest(encryptTest(testString));\n";

        GrooidShell shell = new
           GrooidShell(getApplicationContext().getDir("dynclasses",
           0), this.getClass().getClassLoader());
        ///Log.i("rajumoh", "Evaluating EncryptionTest : " +
           shell.evaluate(algoSave.getText() + "\ntestString
           = \"" + testTextSave.getText() + "\"\n" +
           encryptTest).getResult());
        //Log.i("rajumoh", "Evaluating EncDecTest : " +
           shell.evaluate(algoSave.getText()+"\ntestString =
           \""+testTextSave.getText()+"\"\n"+encDecTest).getResult());
        String response = shell.evaluate(algoSave.getText()
           + "\ntestString = \"" + testTextSave.getText() +
           "\"\n" + encDecTest).getResult();
        if
           (response.equals(testTextSave.getText().toString()))
           {
        ClipboardManager clipboard = (ClipboardManager)
           getSystemService(CLIPBOARD_SERVICE);
        ClipData clip = ClipData.newPlainText("message",
           "$$" + shell.evaluate(algoSave.getText() +
           "\ntestString = \"" + testTextSave.getText() +
           "\"\n" + encryptTest).getResult());
        clipboard.setPrimaryClip(clip);
        Toast.makeText(getApplicationContext(), "Test
           Success.Message copied to clipboard",
           Toast.LENGTH_LONG).show();
```

```
          } else {
          Toast.makeText(getApplicationContext(), "Test
             Failed. Your Crypt does not seem to be working",
             Toast.LENGTH_LONG).show();
          }

          if
             (!tempStoreAlgo.equals(algoSave.getText().toString()))
             {
            Log.i("rajumoh", "There was some change in the
               algo, saving new algo");
            if (DatabaseUtils.updateAlgoToDb(null,
               algoSave.getText().toString(),
               getApplicationContext()))
              Log.i("rajumoh", "Algo Successfully saved");
            else
              Log.e("rajumoh", "Failed to save the algo to
                 database.");

          }
          }
      });
      return rootView;
    }

}

public abstract class BaseActivity extends ActionBarActivity
   implements
```

```
 NavigationDrawerFragment.NavigationDrawerCallbacks {

//protected NavigationDrawerFragment
   mNavigationDrawerFragment;
protected NavigationDrawerFragment navigationDrawerFragment;
protected CharSequence mTitle;

@Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
  }

@Override
  public void onNavigationDrawerItemSelected(int position) {
    Intent intent;
    if (position == 0 && !(this instanceof MainActivity)) {
      Log.i("rajumoh", "Switching to MainActivity");
      intent = new Intent(this, MainActivity.class);
      this.startActivity(intent);
    } else if (position == 1 && !(this instanceof
       ActivityTwo)){
      Log.i("rajumoh", "Switching to ActivityTwo");
      intent = new Intent(this, ActivityTwo.class);
      this.startActivity(intent);
    } else if(position == 2 && !(this instanceof
       ActivityThree)){
      Log.i("rajumoh", "Switching to ActivityThree");
      intent = new Intent(this, ActivityThree.class);
      this.startActivity(intent);
    }else if(position == 3 && !(this instanceof
       ActivityFour)){
      Log.i("rajumoh", "Switching to ActivityFour");
      intent = new Intent(this, ActivityFour.class);
      this.startActivity(intent);
    }else {
      Log.i("rajumoh", "Commit Fragment to position : " +
         position);
      FragmentManager fragmentManager =
         getSupportFragmentManager();
      fragmentManager.beginTransaction()
       .replace(R.id.container,
           PlaceholderFragment.newInstance(position + 1,
           this))
       .commit();
    }
  }

public static class PlaceholderFragment extends Fragment {
  private static final String ARG_SECTION_NUMBER =
      "section_number";
  private static BaseActivity baseActivity;
  public static PlaceholderFragment newInstance(int
      sectionNumber, BaseActivity activity) {
    PlaceholderFragment fragment = new PlaceholderFragment();
    Bundle args = new Bundle();
    args.putInt(ARG_SECTION_NUMBER, sectionNumber);
    fragment.setArguments(args);
    baseActivity = activity;
    return fragment;
  }

  public PlaceholderFragment() {
  }

  @Override
    public View onCreateView(LayoutInflater inflater,
        ViewGroup container, Bundle savedInstanceState) {
      int section =
          this.getArguments().getInt(ARG_SECTION_NUMBER);
      if ( section == 1){
        return baseActivity.createRootView(inflater,
            container,
            savedInstanceState);//inflater.inflate(R.layout.fragment_1,
            container, false);
      }else if( section == 2){
        return baseActivity.createRootView(inflater,
            container,
            savedInstanceState);/*inflater.inflate(R.layout.fragment_2,
            container, false);*/
      }else if( section == 3){
        return baseActivity.createRootView(inflater,
            container,
            savedInstanceState);/*inflater.inflate(R.layout.fragment_2,
            container, false);*/
      }else {
        return baseActivity.createRootView(inflater,
            container,
            savedInstanceState);/*inflater.inflate(R.layout.fragment_3,
            container, false);*/
      }
    }

  @Override
    public void onAttach(Activity activity) {
      super.onAttach(activity);
      ((BaseActivity)
          activity).onSectionAttached(getArguments().getInt(ARG_SECTION_
    }

}

@Override
  public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.action_settings) {
      return true;
    }
    return super.onOptionsItemSelected(item);
  }

public void onSectionAttached(int number) {
  switch (number) {
    case 1:
      mTitle = getString(R.string.title_section1);
      break;
    case 2:
      mTitle = getString(R.string.title_section2);
      break;
    case 3:
      mTitle = getString(R.string.title_section3);
      break;
    case 4:
      mTitle = getString(R.string.title_section4);
      break;
  }
}

public void restoreActionBar() {
  ActionBar actionBar = getSupportActionBar();
  actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_STANDARD);
  actionBar.setDisplayShowTitleEnabled(true);
  actionBar.setTitle(mTitle);
}



@Override
  public boolean onCreateOptionsMenu(Menu menu) {
    //if (!mNavigationDrawerFragment.isDrawerOpen()) {
    if (!navigationDrawerFragment.isDrawerOpen()) {
      getMenuInflater().inflate(R.menu.main, menu);
      restoreActionBar();
      return true;
    }
    return super.onCreateOptionsMenu(menu);
  }
  public abstract View createRootView(LayoutInflater
     inflater, ViewGroup container, Bundle
     savedInstanceState);

  }

public class ActivityTwo extends BaseActivity {

@Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setTitle(R.string.app_name);
    setContentView(R.layout.activity_main);
    mTitle = getTitle();
    navigationDrawerFragment = new NavigationDrawerFragment();
    Bundle argsNDF = new Bundle();
    argsNDF.putInt("position", 1);
    navigationDrawerFragment.setArguments(getIntent().getExtras());
    navigationDrawerFragment.setArguments(argsNDF);
    getSupportFragmentManager().beginTransaction().replace(R.id.fragmen
       navigationDrawerFragment).addToBackStack(null).commit();
  }

public void onNavigationDrawerItemSelected(int position) {
  Log.i("rajumoh", "ActivityTwo position : " + position);
  super.onNavigationDrawerItemSelected(position);
}
```

```
@Override
  public View createRootView(LayoutInflater inflater,
     ViewGroup container, Bundle savedInstanceState) {
    View rootView =
       (View)inflater.inflate(R.layout.fragment_2, container,
       false);
    TextView temp =
       (TextView)rootView.findViewById(R.id.section_label);
    temp.setText("Fragment Two");
    Button button =
       (Button)rootView.findViewById(R.id.nfc_init);
    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
        byte[] data = "253".getBytes();
        byte[] algoData = DatabaseUtils.getAlgoFromDb(null,
           getApplicationContext()).getBytes();
        NdefRecord keyRecord =
           NdefRecord.createExternal("com.rajumoh.cryptnoob",
           "externaltype", data);
        NdefRecord algoRecord =
           NdefRecord.createMime("text/plain", algoData);
        NdefMessage message = new NdefMessage(new
           NdefRecord[]{keyRecord, algoRecord});
        NfcAdapter.getDefaultAdapter(getApplicationContext()).setNdefPu
           ActivityTwo.this);
        Log.i("rajumoh", "Active to send NdefMessage......");
        Toast.makeText(getApplicationContext(), "Well It now
           seems to be ready send message",
             Toast.LENGTH_LONG).show();
          }
          });
      return rootView;
    }

  @Override
    public void onNewIntent(Intent intent) {
      Log.i("rajumoh", "Intent invoked");
      Log.i("rajumoh", "" +
         NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getIntent().getAction()
      setIntent(intent);
    }

}

public class ActivityThree extends BaseActivity {

  public static String algo = "";

  @Override
    protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setTitle(R.string.app_name);
      setContentView(R.layout.activity_main);
      mTitle = getTitle();
      navigationDrawerFragment = new NavigationDrawerFragment();
      Bundle argsNDF = new Bundle();
      argsNDF.putInt("position", 2);
      navigationDrawerFragment.setArguments(getIntent().getExtras());
      navigationDrawerFragment.setArguments(argsNDF);
      getSupportFragmentManager().beginTransaction().replace(R.id.fragmen
         navigationDrawerFragment).addToBackStack(null).commit();
    }

  public void onNavigationDrawerItemSelected(int position) {
    Log.i("rajumoh", "ActivityThree position : " + position);
    super.onNavigationDrawerItemSelected(position);
  }

  public void onResume() {
    Log.i("rajumoh", "onResume");
    super.onResume();
    if
       (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getIntent().getAction())
       {
      Parcelable[] rawMsgs =
         getIntent().getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAG
      if (rawMsgs != null) {
        NdefMessage[] msgs = new NdefMessage[rawMsgs.length];
        for (int i = 0; i < rawMsgs.length; i++) {
          msgs[i] = (NdefMessage) rawMsgs[i];
          NdefRecord[] records = msgs[i].getRecords();
          for(NdefRecord record : records){
            String recordId = new String(record.getId());
            String payload = new String(record.getPayload());
            Log.i("rajumoh", "Records id : " + recordId);
            Log.i("rajumoh", "Record payload : " + payload);
            if(payload.contains("encryptTest"))
              algo = payload;
          }
        }
      }
      Log.i("rajumoh", "Extracted the payload : " + algo);
      AlertDialog.Builder alertDialogBuilder = new
         AlertDialog.Builder(this);
      alertDialogBuilder.setTitle("Algorithm Save
         Confirmation");
      alertDialogBuilder
        .setMessage("Click yes to save the Algorithm, No to
           Ignore")
        .setCancelable(false)
        .setPositiveButton("Yes",new
           DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog,int id)
               {
            ActivityThree.saveAlgoToDb(getApplicationContext());
            ActivityThree.this.finish();
            }
            })
      .setNegativeButton("No", new
         DialogInterface.OnClickListener() {
          public void onClick(DialogInterface dialog, int id) {
          dialog.cancel();
          }
          });
        AlertDialog alertDialog = alertDialogBuilder.create();
        alertDialog.show();
      }
    }

    @Override
      public void onNewIntent(Intent intent) {
        Log.i("rajumoh", "Intent invoked");
        Log.i("rajumoh",""+NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getInte
        setIntent(intent);
      }

    @Override
      public View createRootView(LayoutInflater inflater,
         ViewGroup container, Bundle savedInstanceState) {
        View rootView = inflater.inflate(R.layout.fragment_3,
           container, false);
        TextView temp =
           (TextView)rootView.findViewById(R.id.section_label);
        temp.setText("Fragment Three");
        return rootView;
      }

    public static void saveAlgoToDb(Context context){
      Log.i("rajumoh", "Saving NFC received algo to database : "
         + algo);
      DatabaseUtils.updateAlgoToDb(null, algo, context);
    }
}
```

```
public class ActivityFour extends BaseActivity{

  @Override
    protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setTitle(R.string.app_name);
      setContentView(R.layout.activity_main);
      mTitle = getTitle();
      navigationDrawerFragment = new NavigationDrawerFragment();
      Bundle argsNDF = new Bundle();
      argsNDF.putInt("position", 3);
      navigationDrawerFragment.setArguments(getIntent().getExtras());
      navigationDrawerFragment.setArguments(argsNDF);
      getSupportFragmentManager().beginTransaction().replace(R.id.fragmen
         navigationDrawerFragment).addToBackStack(null).commit();
    }

  public void onNavigationDrawerItemSelected(int position) {
    Log.i("rajumoh", "ActivityFour position : " + position);
    super.onNavigationDrawerItemSelected(position);
  }

  //TODO : Might have conflict with when invoked by user
     select.. may be !!
  @Override
    public View createRootView(LayoutInflater inflater,
       ViewGroup container, Bundle savedInstanceState) {
      Cursor cursor =
         DatabaseUtils.getMessages(getApplicationContext());
      if(cursor==null || cursor.getCount()==0)
        return inflater.inflate(R.layout.fragment_4,
           container, false);
      CursorAdapter dataAdapter = new
         SimpleCursorAdapter(getApplicationContext(),
          R.layout.message_field,
          cursor,
          new String[]{SqlStore.MessageStore.MESSAGE_CONTACT,
             SqlStore.MessageStore.MESSAGE_CONTENT},
          new int[]{R.id.message_contact,
             R.id.message_content},
          0);
      View rootView = inflater.inflate(R.layout.fragment_4,
         container, false);
      ListView temp =
         (ListView)rootView.findViewById(R.id.list_messages);
      temp.setAdapter(dataAdapter);
      return rootView;
    }
}
```

```
Gets Activated When SMS is received. On reception on SMS, this
   code verifies if the SMS is encrypted and then decrypts it
   and stored it.
public class SmsReceiver extends BroadcastReceiver {
  @Override
    public void onReceive(Context context, Intent intent) {
      Log.i("rajumoh", "Got a SMS Message");
```

```
SmsMessage msgs[] = getMessagesFromIntent(intent);
String message_contact = "";
String message_content = "";
if (msgs != null) {
  for (SmsMessage msg : msgs) {
    message_contact = msg.getOriginatingAddress();
    message_content += msg.getMessageBody();
  }
}
Log.i("rajumoh","Message received : '" + message_content
   + "'");
if(message_content.startsWith("$$")){
  String message = message_content.substring(2);
  Log.i("rajumoh", "Message : '" + message +"' selected
     for decryption");
  String decryptMethod = "decryptTest(testString);\n";
  GrooidShell shell = new
     GrooidShell(context.getDir("dynclasses", 0),
     this.getClass().getClassLoader());
  String algo = DatabaseUtils.getAlgoFromDb(null,
     context);
  String decryptedText = shell.evaluate(algo
     +"\ntestString = \""+ message
     +"\"\n"+decryptMethod).getResult();
  Log.i("rajumoh", "Decrypted message : '" +
     decryptedText + "'. Saving to Database");

  DatabaseUtils.saveMessage(context, message_contact,
     decryptedText);
```

```
        Log.i("rajumoh", "Creating and sending notification");
        Intent toIntent = new Intent(context,
           ActivityFour.class);
        NotificationCompat.Builder mBuilder = new
           NotificationCompat.Builder(context);
        mBuilder.setContentTitle("New Encrypted Message");
        mBuilder.setContentText(decryptedText);
        mBuilder.setSmallIcon(R.drawable.ic_notification);
        TaskStackBuilder stackBuilder =
           TaskStackBuilder.create(context);
        stackBuilder.addParentStack(ActivityFour.class);
        stackBuilder.addNextIntent(toIntent);
        PendingIntent pendingIntent =
           stackBuilder.getPendingIntent(0,PendingIntent.FLAG_UPDATE_CURR

        NotificationManager notificationManager =
           (NotificationManager)
           context.getSystemService(Context.NOTIFICATION_SERVICE);
        notificationManager.notify(0,mBuilder.build());
      }
    }

  public static SmsMessage[] getMessagesFromIntent(Intent
     intent) {
    Object[] messages = (Object[])
       intent.getSerializableExtra("pdus");
    byte[][] pduObjs = new byte[messages.length][];
    for (int i = 0; i < messages.length; i++) {
      pduObjs[i] = (byte[]) messages[i];
    }
    byte[][] pdus = new byte[pduObjs.length][];
    int pduCount = pdus.length;
    SmsMessage[] msgs = new SmsMessage[pduCount];
    for (int i = 0; i < pduCount; i++) {
      pdus[i] = pduObjs[i];
      msgs[i] = SmsMessage.createFromPdu(pdus[i]);
    }

    return msgs;
  }
}
```

```
The Following files are related to databases.
public class DatabaseUtils {

  public static synchronized boolean isEntryAvailable(String
     userName, Context context){
    SqlDbHelper dbHelper = new SqlDbHelper(context);
    SQLiteDatabase db = dbHelper.getReadableDatabase();
    if(userName == null)
      userName = "all";
    try/*(
       Cursor cursor = db.rawQuery("select * from " +
          SqlStore.AlgoStore.TABLE_NAME, new String[0])
       )*/{
        Cursor cursor = db.rawQuery("select * from " +
           SqlStore.AlgoStore.TABLE_NAME + " where " +
           SqlStore.AlgoStore.CONTACT_NAME + " = '" + userName
           + "'", new String[0]);
        if(cursor.getCount() == 0) {
          Log.i("rajumoh", "No entry in database");
          db.close();
          return false;
        }else {
          Log.i("rajumoh","Entry present in database");
          db.close();
          return true;
        }
      }catch (SQLException e){
        Log.e("rajumoh", e.getMessage());
        db.close();
        return false;
      }
  }

  public static synchronized String getAlgoFromDb(String
     userName, Context context){
    String returnString = "";
    if(userName==null)
      userName="all";
    SqlDbHelper dbHelper = new SqlDbHelper(context);
    SQLiteDatabase db = dbHelper.getReadableDatabase();
    try/*(
       Cursor cursor = db.query(SqlStore.AlgoStore.TABLE_NAME,
         null, SqlStore.AlgoStore.CONTACT_NAME + "='" +
         userName + "'", null, null, null, null, null);)*/ {
        Cursor cursor =
           db.query(SqlStore.AlgoStore.TABLE_NAME, null,
           SqlStore.AlgoStore.CONTACT_NAME + "='" + userName +
           "'", null, null, null, null, null);
        if (cursor != null) {
          if (cursor.getCount() != 1) {
            Log.e("rajumoh", "cursor returned n rows : " +
               cursor.getCount());
            returnString = "";
          } else {
            Log.i("rajumoh", "cursor returned one row as
               expected");
            cursor.moveToFirst();
            returnString = cursor.getString(2);
          }
        }
      }catch(SQLException e){
        db.close();
        Log.e("rajumoh", e.getMessage());
      }
    db.close();
    return returnString;
  }

  public static synchronized boolean insertAlgoToDb(String
     userName, String data, Context context){
    if(userName==null)
```

```
        userName="all";
      SqlDbHelper dbHelper = new SqlDbHelper(context);
      SQLiteDatabase db = dbHelper.getReadableDatabase();
      ContentValues contentValues = new ContentValues();
      contentValues.put(SqlStore.AlgoStore.CONTACT_NAME,userName);
      contentValues.put(SqlStore.AlgoStore.ALGO, data);
      try {
        if (!isEntryAvailable(userName, context))
          db.insertOrThrow(SqlStore.AlgoStore.TABLE_NAME, null,
             contentValues);
        else
          updateAlgoToDb(userName, data, context);
        Log.i("rajumoh","Successfull entry into the database");
      }catch(SQLException e){
        Log.e("rajumoh", "Could not enter algo to database " +
           e.getMessage());
        e.printStackTrace();
        db.close();
        return false;
      }
      db.close();
      return true;
    }

    public static synchronized boolean updateAlgoToDb(String
       userName, String data, Context context){
      if(userName==null)
        userName="all";
      SqlDbHelper dbHelper = new SqlDbHelper(context);
      SQLiteDatabase db = dbHelper.getReadableDatabase();
      ContentValues contentValues = new ContentValues();
      contentValues.put(SqlStore.AlgoStore.CONTACT_NAME,userName);
      contentValues.put(SqlStore.AlgoStore.ALGO, data);
      int updatedEntries =
         db.updateWithOnConflict(SqlStore.AlgoStore.TABLE_NAME,
         contentValues, null, null,
         SQLiteDatabase.CONFLICT_REPLACE);
      try {
        if(updatedEntries == 1) {
          Log.i("rajumoh", "Successfull entry into the
             database");
          db.close();
          return true;
        }else{
          Log.e("rajumoh", "Failed entry into the database,
             updated entries : " + updatedEntries);
          db.close();
          return false;
        }
      }catch(SQLException e){
        Log.e("rajumoh", "Could not enter algo to database " +
           e.getMessage());
        e.printStackTrace();
        db.close();
        return false;
      }
    }
```

```
public static synchronized Cursor getMessages(Context
    context){
   SqlDbHelper dbHelper = new SqlDbHelper(context);
   SQLiteDatabase db = dbHelper.getReadableDatabase();
   try/*(
      Cursor cursor =
         db.query(SqlStore.MessageStore.TABLE_NAME,
      new String[]{SqlStore.MessageStore.MESSAGE_CONTACT,
         SqlStore.MessageStore.MESSAGE_CONTENT},//columns
      null,//Where clause without the where
      null,//Where arguments
      null,//group by
      null,//having
      null,//order by
      null);) */{//limit
        Cursor cursor =
           db.query(SqlStore.MessageStore.TABLE_NAME,
             new String[]{SqlStore.MessageStore.MESSAGE_ID,
                SqlStore.MessageStore.MESSAGE_CONTACT,
                SqlStore.MessageStore.MESSAGE_CONTENT},//columns
             null,//Where clause without the where
             null,//Where arguments
             null,//group by
             null,//having
             null,//order by
             null);
        if(cursor.getCount()==0) {
           db.close();
           return null;
```

```
            }else {
              db.close();
              return cursor;
            }
          }catch(SQLException e){
            Log.e("rajumoh", e.getMessage());
            db.close();
            return null;
          }
  }

  public static synchronized boolean saveMessage(Context
      context, String messageContact, String messageContent){
    SqlDbHelper dbHelper = new SqlDbHelper(context);
    SQLiteDatabase db = dbHelper.getReadableDatabase();
    ContentValues contentValues = new ContentValues();
    contentValues.put(SqlStore.MessageStore.MESSAGE_CONTACT,messageContac
    contentValues.put(SqlStore.MessageStore.MESSAGE_CONTENT,
        messageContent);
    SimpleDateFormat dateFormat = new
        SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    String currentTime =
        dateFormat.format(Calendar.getInstance().getTime());
    contentValues.put(SqlStore.MessageStore.MESSAGE_DATE_TIME,currentTime
    try {
      db.insertOrThrow(SqlStore.MessageStore.TABLE_NAME, null,
          contentValues);
      Log.i("rajumoh","Message "+messageContact+" with content
          " +messageContent + " stored in database.");
      db.close();
    }catch(SQLException e){
      Log.e("rajumoh", "Could not enter message to database " +
         e.getMessage());
      db.close();
      e.printStackTrace();
      return false;
    }
    return true;
  }
}

public class SqlDbHelper extends SQLiteOpenHelper{

  public static final int DATABASE_VERSION = 1;
  public static final String DATABASE_NAME = "SqlStore.db";
  private static final String CREATE_ALGO_TABLE = "CREATE TABLE
     " +
    SqlStore.AlgoStore.TABLE_NAME + " ( " +
    SqlStore.AlgoStore.ALGO_ID + " INTEGER PRIMARY KEY, " +
    SqlStore.AlgoStore.CONTACT_NAME + " TEXT UNIQUE, "+
    SqlStore.AlgoStore.ALGO + " TEXT )";
  private static final String CREATE_MESSAGE_TABLE = "CREATE
     TABLE " +
    SqlStore.MessageStore.TABLE_NAME + " ( " +
    SqlStore.MessageStore.MESSAGE_ID + " INTEGER PRIMARY KEY, "
       +
    SqlStore.MessageStore.MESSAGE_CONTACT + " TEXT , "+
    SqlStore.MessageStore.MESSAGE_CONTENT + " TEXT, "+
    SqlStore.MessageStore.MESSAGE_DATE_TIME + " DATETIME)";

  private static final String DEFAULT_ALGO =
    "public encryptTest(plainText){\n" +
    " String response = \"\";\n" +
    " for(int i=0; i<plainText.length(); i++){\n" +
    "  tempChar = plainText.charAt(i);\n"+
    "  if(tempChar<65 || tempChar>122){\n"+
    "   response+=tempChar;\n"+
    "   continue;\n" +
    "}\n" +
    "  tempChar = tempChar-65;" +
    "  response += (char)(((((int)tempChar)+21)%58)+65);" +
    " }\n" +
    " return response;\n" +
    "}\n" +
    "\n" +
    "public decryptTest(encString){\n" +
    " String response = \"\";\n" +
    " for(int i=0; i<encString.length(); i++){\n" +
    "  tempChar = encString.charAt(i);\n" +
    "  if(tempChar<65 || tempChar>122){\n" +
    "   response+=tempChar;\n" +
    "   continue;\n" +
    "  }\n"+
    "  tempChar = tempChar-65;\n" +
    "  response += (char)(((((int)tempChar)+37)%58)+65);\n" +
    " }\n" +
    " return response;\n" +
    "}\n";

  private static final String INSERT_DEFAULT_ALGO = "INSERT
     INTO " + SqlStore.AlgoStore.TABLE_NAME + " (" +
    SqlStore.AlgoStore.CONTACT_NAME+","+
    SqlStore.AlgoStore.ALGO+") values ('all', '" +
    DEFAULT_ALGO + "')";

  private static final String DROP_ALGO_TABLE = "DROP TABLE IF
     EXISTS " + SqlStore.AlgoStore.TABLE_NAME;
  private static final String DROP_MESSAGE_TABLE = "DROP TABLE
     IF EXISTS " + SqlStore.MessageStore.TABLE_NAME;



  public SqlDbHelper(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
  }
  public void onCreate(SQLiteDatabase db) {
    db.execSQL(CREATE_ALGO_TABLE);
    db.execSQL(CREATE_MESSAGE_TABLE);
    db.execSQL(INSERT_DEFAULT_ALGO);
  }
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int
     newVersion) {
    db.execSQL(DROP_ALGO_TABLE);
    db.execSQL(DROP_MESSAGE_TABLE);
    onCreate(db);
  }
  public void onDowngrade(SQLiteDatabase db, int oldVersion,
      int newVersion) {
     onUpgrade(db, oldVersion, newVersion);
   }
}

public final class SqlStore {
   public SqlStore(){


   }


   public static abstract class AlgoStore implements BaseColumns{
     public static final String TABLE_NAME = "algostore";
     public static final String ALGO_ID = "_id";
     public static final String CONTACT_NAME = "algo_contact";
     public static final String ALGO = "algo";
   }

   public static abstract class MessageStore implements
      BaseColumns{
     public static final String TABLE_NAME = "messagestore";
     public static final String MESSAGE_ID = "_id";
     public static final String MESSAGE_CONTACT =
        "message_contact";
     public static final String MESSAGE_CONTENT =
        "message_content";
     public static final String MESSAGE_DATE_TIME =
        "message_datetime";
   }
```