Android Security: Security using Encryption (AES)

Project report submitted in partial fulfilment of the requirement for the degree of

Bachelor of Technology

In

Computer Science and Engineering

Under the supervision of

Mr. Suman Saha

By

Abhishek Amola (121235)

to

Department of Computer Science & Engineering and Information Technology

Jaypee University of Information Technology Waknaghat, Solan-173234, Himachal Pradesh

Candidate's Declaration

I hereby declare that the work presented in this report entitled **"Android Security: Security using Encryption (AES)"** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** submitted in the department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from August 2015 to May 2016 under the supervision of **(Mr. Suman Saha)** (Assistant Professor, Computer Science & Engineering).

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Date: **Abhishek Amola (121235)**

Certificate

This is to certify that project report entitled **"Android Security: Security using Encryption (AES)"**, submitted by Abhishek Amola, 121235 in partial fulfillment for the award of degree of Bachelor of Technology in Information Technology to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date: **Mr. Suman Saha**

Assistant Professor

Acknowledgement

I gratefully acknowledge the Management and Administration of Jaypee University of Information Technology for providing me the opportunity and hence the environment to initiate and complete my project.

I would like to express my gratitude to Prof. Dr. RMK Sinha, Dean and Head of Department, Information Technology JUIT and Brig. (Retd) S.P.Ghrera Head of Department, Computer Science, JUIT Waknaghat for guiding, encouraging and inspiring by giving me valuable thoughts to carry out the project and also like to thank my project guide, Prof. Suman Saha, Assistant Professor, Dept. of Information Technology for his expert guidance, encouragement and valuable suggestions at every step.

Date: **Abhishek Amola**

Contents

List of Figures

Abstract

Security is an essential aspect of the software we develop. Data with which we deal is sometimes important and should be made secure. In case of android the database that we use is SQLite which is a very light database, which is very less secure, it has no type encryption or decryption available. Data stored in SQLite can be easily retrieved if our android device gets into wrong hands. Various tools are also available online which help in extracting data from SQLite database.

My project is related to security of data provided using encryption. The encryption method I am going to use is Advanced Encryption Standard (AES) which is the best known symmetrical encryption available.

In the project AES is implemented in java and is used to secure our data in database. Application vendors like facebook, twitter, etc. which are big companies have their own security and use servers to store and retrieve data, because the security provided by android is not enough. The code of AES implemented in the project can be used in any application as an API to provide security to the data being stored. So the project mainly focuses on the implementation of AES and demonstrating its usage in android database SQLite.

The idea of the project came from AOSP (Android Open Source Project) where this problem has been mentioned.

CHAPTER 1 – INTRODUCTION

1.1 Introduction

Android is an operating system developed by Google, primarily for mobile devices. It is based on Linux kernel and designed for touchscreen devices like smartphones and tablets. Google has further developed Android TV for televisions, Android Auto for cars and Android Wear for wrist watches.

My project is on one aspect related to android, which is its security. Android applications make use of advanced hardware and software. Data can provided locally or using severs. Android has multi-layered security that provides flexibility required for an open platform.

Android security can be divided into two parts-

- Kernel security
- Application security

Kernel security is the security which is at operating system level. The foundation of the Android platform is the Linux kernel. The Linux kernel itself has been in widespread use for years, and is used in millions of security-sensitive environments. Through its history of constantly being researched, attacked, and fixed by thousands of developers, Linux has become a stable and secure kernel trusted by many corporations and security professionals.

At kernel level android is provided several key features by Linux kernel. Some of them are-

- A user-based permissions model
- Process isolation
- Extensible mechanism for secure IPC
- The ability to remove unnecessary and potentially insecure parts of the kernel

One of the major security feature is Application Sandbox. The Android system assigns a unique user ID (UID) to each Android application and runs it as that user in a separate process. This approach is different from other operating systems (including the traditional Linux configuration), where multiple applications run with the same user permissions. This sets up a kernel-level Application Sandbox. The kernel enforces security between applications and the system at the process level through standard Linux facilities, such as user and group IDs that are assigned to applications. By default, applications cannot interact with each other and applications have limited access to the operating system. If application A tries to do something malicious like read application B's data or dial the phone without permission (which is a separate application), then the operating system protects against this because application A does not have the appropriate user privileges. The sandbox is simple, auditable, and based on decades-old UNIX-style user separation of processes and file permissions.

Application Security is the security of android at application level i.e. working of applications. Android applications are most often written in the Java programming language and run in the Dalvik virtual machine. However, applications can also be written in native code. Applications are installed from a single file with the .apk file extension. Android system itself provides with various security features for security of applications like Application Sandbox, Permissions, etc. Application Sandbox is explained earlier.

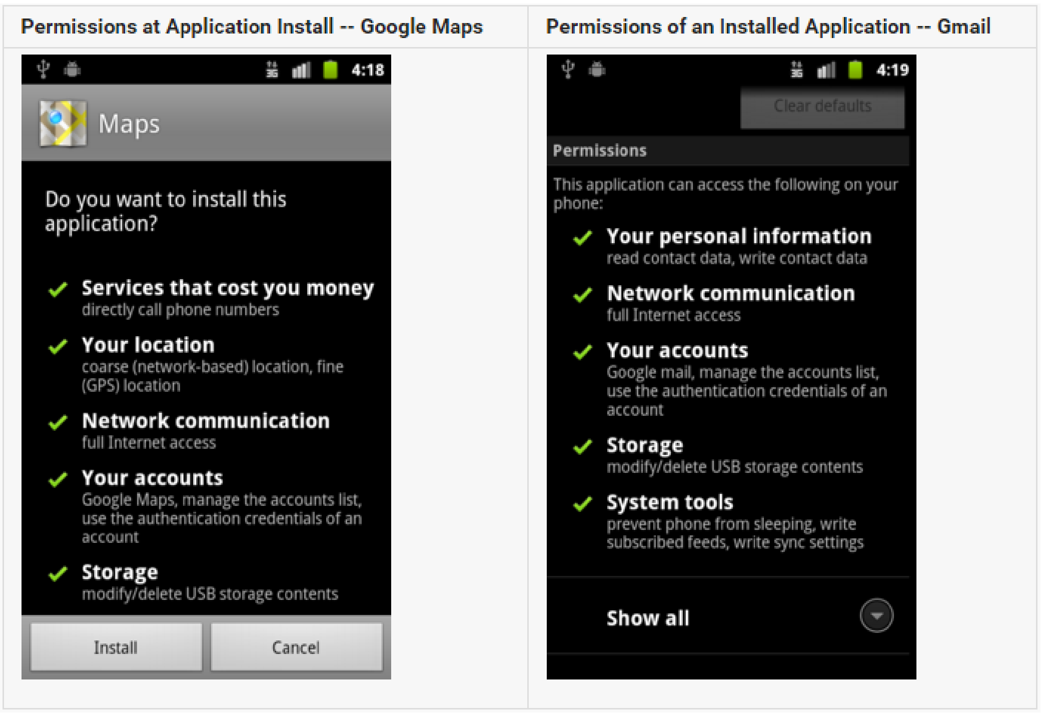
All applications on Android run in an Application Sandbox, described earlier. By default, an Android application can only access a limited range of system resources. The system manages Android application access to resources that, if used incorrectly or maliciously, could adversely impact the user experience, the network, or data on the device.

These restrictions are implemented in a variety of different forms. Some capabilities are restricted by an intentional lack of APIs to the sensitive functionality (e.g. there is no Android API for directly manipulating the SIM card). In some instances, separation of roles provides a security measure, as with the per-application isolation of storage. In other instances, the sensitive APIs are intended for use by trusted applications and protected through a security mechanism known as Permissions.

These protected APIs include:

- Camera functions
- Location data (GPS)
- Bluetooth functions
- Telephony functions
- SMS/MMS functions
- Network/data connections

These resources are only accessible through the operating system. To make use of the protected APIs on the device, an application must define the capabilities it needs in its manifest. When preparing to install an application, the system displays a dialog to the user that indicates the permissions requested and asks whether to continue the installation.



Permissions at Application Install -- Google Maps

4:18

Maps

Do you want to install this application?

✓ **Services that cost you money**
directly call phone numbers

✓ **Your location**
coarse (network-based) location, fine (GPS) location

✓ **Network communication**
full Internet access

✓ **Your accounts**
Google Maps, manage the accounts list, use the authentication credentials of an account

✓ **Storage**
modify/delete USB storage contents

Install | Cancel

Permissions of an Installed Application -- Gmail

4:19

Clear defaults

Permissions

This application can access the following on your phone:

✓ **Your personal information**
read contact data, write contact data

✓ **Network communication**
full Internet access

✓ **Your accounts**
Google mail, manage the accounts list, use the authentication credentials of an account

✓ **Storage**
modify/delete USB storage contents

✓ **System tools**
prevent phone from sleeping, write subscribed feeds, write sync settings

Show all

Figure 1: Permissions for Applications

All these features and many more are provided by android for security. At application level though android provides security features managed by kernel but these are not enough. At application level developer has to provide some type his/her own security to make application more secure. This is a major area where we can do productive work. So our project will mostly be focusing on Application Security of Android.

1.2 Problem Statement

Features provided by Android Operating System are not enough for security of applications. Developer needs to provide his/her own type of security for the application.

Applications made by big vendors like Facebook, Twitter, etc. are having their own security features like encryption of data which is coming from sever, security for username and password etc. They spend huge amount of money for security of data of their customers. If an application is made by some person with that not that much funding and is using some sensitive data it will not secure and can be attacked.

When we install an application we allow the application to use some features of our device like Wi-Fi and, this is called giving permission as explained earlier. When permissions are given we might be sharing our sensitive data with the application. If application is not secured than our data will not be secure. Small scale developers are not able to secure their application completely. As market of android is growing there new applications on the play store every day. Security issue is also rising. It’s difficult to handle security when you are making an application yourself and at a small scale but using sensitive data.

I am focusing on the problem of securing data which an application uses and saves somewhere. Data can be stored in local database or in severs. Some applications also send and receive data. There can be various forms of data. Data can be in the form of text, audio, video, image, etc. Security should be provided for these forms of data.

1.3 Objectives

Main objective of the project is to provide security for the android applications by securing the data they are using. There can be various forms of data. Data can be in the form of text, audio, video, image, etc. Security should be provided for these forms of data. We have to secure the data.

Data can be secured if encrypt the data. Type of encryption to be used depends on whether it's being sent and received or whether it's being saved. There are two types of encryption-

- Symmetrical Encryption
- Asymmetrical Encryption

Symmetrical encryption is the encryption done with only one key. Using this key we do various shifting and jumbling up of data and form an encrypted text called the cipher. Best known and most widely used symmetric encryption algorithm is AES (Advanced Encryption Standard).

Asymmetrical encryption is the encryption done with only two keys – a pubic and a private key. The public one is available for everyone, but the private one is known only by the owner. When the message is encrypted with the public key, only the corresponding private key can decrypt it. Moreover, the private key can't be learned from the public one. Best known and most widely used symmetric encryption algorithm is RSA algorithm.

We will be focusing on securing the data an android application saves in the SQLite database of android. Let's say there is an application that is using database to save some information of user. If data being saved is not secured it can be hacked someone can get sensitive data. This can be stopped if we encrypt the data while storing and decrypt the data while retrieving it. Our aim is make a system where we can encrypt and decrypt the data and make applications more secure. First we will be securing the data that we save in the SQLite database and then we can extend it to various other forms through which data is stored and communicated.

1.4 Methodology

We want to secure the data which we save in SQLite database of android. We are not communicating and sending some data, we are storing it. Symmetric Encryption would be appropriate for this scenario.

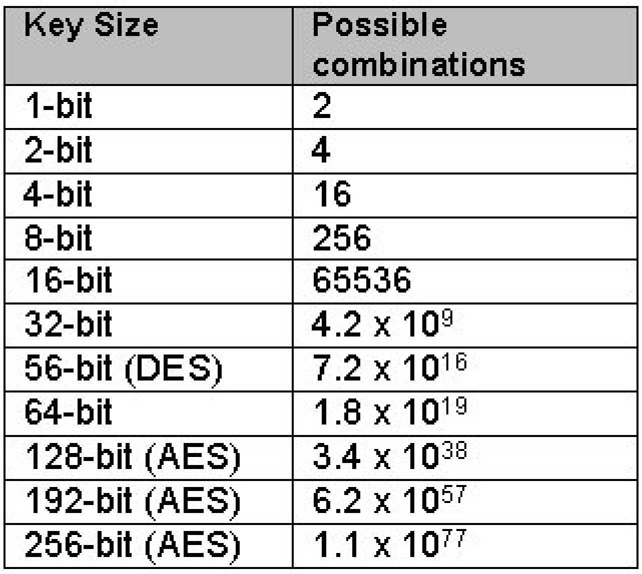
Symmetric encryption as explained earlier works on one key. It is the oldest and best-known technique. A secret key, which can be a number, a word, or just a string of random letters, is applied to the text of a message to change the content in a particular way. This might be as simple as shifting each letter by a number of places in the alphabet. As long as both sender and recipient know the secret key, they can encrypt and decrypt all messages that use this key. AES (Advanced Encryption Standard) is the best symmetric encryption algorithm available. Symmetric encryption algorithm are less complex and stronger than asymmetric encryption algorithm. AES is the most secure algorithm available in the world right now. So we will be using AES encryption algorithm.

AES comprises three block ciphers, AES-128, AES-192 and AES-256. Each cipher encrypts and decrypts data in blocks of 128 bits using cryptographic keys of 128-, 192- and 256-bits, respectively. (Rijndael was designed to handle additional block sizes and key lengths, but the functionality was not adopted in AES.) Symmetric or secret-key ciphers use the same key for encrypting and decrypting, so both the sender and the receiver must know and use the same secret key. All key lengths are deemed sufficient to protect classified information up to the "Secret" level with "Top Secret" information requiring either 192- or 256-bit key lengths. There are 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys -- a round consists of several processing steps that include substitution, transposition and mixing of the input plaintext and transform it into the final output of cipher text.

As a cipher, AES has proven reliable. The only successful attacks against it have been side-channel attacks on weaknesses found in the implementation or key management of certain AES-based encryption products. (Side-channel attacks don't use brute force or theoretical weaknesses to break a cipher, but rather exploit flaws in the way it has been implemented.) The BEAST browser exploit against the TLS v1.0 protocol is a good example; TLS can use AES to encrypt data, but due to the information that TLS exposes, attackers managed to predict the initialization vector block used at the start of the encryption process.

Various researchers have published attacks against reduced-round versions of the Advanced Encryption Standard, and a research paper published in 2011 demonstrated that using a technique called a bi clique attack could recover AES keys faster than a brute-force attack by a factor of between three and five, depending on the cipher version. Even this attack, though, does not threaten the practical use of AES due to its high computational complexity.

Following table shows possible number of key combinations with respect to key size:



Key Size	Possible combinations
1-bit	2
2-bit	4
4-bit	16
8-bit	256
16-bit	65536
32-bit	4.2×10^{9}
56-bit (DES)	7.2×10^{16}
64-bit	1.8×10^{19}
128-bit (AES)	3.4×10^{38}
192-bit (AES)	6.2×10^{57}
256-bit (AES)	1.1×10^{77}

Figure 2: Key combinations versus key size

We are going to use AES-128. AES-128 has 128 bit long key. For AES-128, the key can be recovered with a computational complexity of $2^{126.1}$ using the attack. There is a physical argument that a 128-bit symmetric key is computationally secure against brute-force attack. Just consider the following:

Faster supercomputer (as per Wikipedia): 10.51 Penta flops = 10.51 x 10^{15} Flops [Flops = Floating point operations per second]

No. of Flops required per combination check: 1000 (very optimistic but just assume for now)

No. of combination checks per second = (10.51 x 10^{15}) / 1000 = 10.51 x 10^{12}

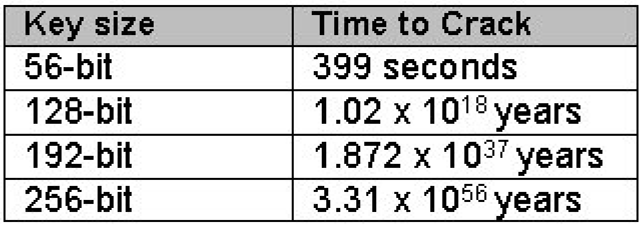
No. of seconds in one Year = 365 x 24 x 60 x 60 = 31536000

No. of Years to crack AES with 128-bit Key = (3.4 x 10^{38}) / [(10.51 x 10^{12}) x 31536000]

= (0.323 x 10^{26})/31536000

= 1.02 x 10^{18}

= 1 billion billion years



Key size	Time to Crack
56-bit	399 seconds
128-bit	1.02 x 10^{18} years
192-bit	1.872 x 10^{37} years
256-bit	3.31 x 10^{56} years

Figure 3: Time to crack vs. key size

As shown above, even with a supercomputer, it would take 1 billion billion years to crack the 128-bit AES key using brute force attack. This is more than the age of the universe (13.75 billion years). If one were to assume that a computing system existed that could recover a DES key in a second, it would still take that same machine approximately 149 trillion years to crack a 128-bit AES key.

There are more interesting examples. The following snippet is a snapshot of one the technical papers from Seagate titled "128-bit versus 256-bit AES encryption" to explain why 128-bit AES is sufficient to meet future needs. If you assume:

Every person on the planet owns 10 computers.

There are 7 billion people on the planet.

Each of these computers can test 1 billion key combinations per second.

On average, you can crack the key after testing 50% of the possibilities.

Then the earth's population can crack one encryption key in 77,000,000,000,000,000,000,000,000 years!

The bottom line is that if AES could be compromised, the world would come to a standstill. The difference between cracking the AES-128 algorithm and AES-256 algorithm is considered minimal. Whatever breakthrough might crack 128-bit will probably also crack 256-bit.

In the end, AES has never been cracked yet and is safe against any brute force attacks contrary to belief and arguments. However, the key size used for encryption should always be large enough that it could not be cracked by modern computers despite considering advancements in processor speeds based on Moore's law.

AES has never been cracked and so is appropriate for our encryption purpose. Now we have chosen the encryption method. We will be coding in JAVA as it’s the language in which most of the android application are made. Once the code is made we will implement on various forms of data and then on move on to according to the place of storage of data.

CHAPTER 2 – LITERATURE SURVEY

2.1 Overview of Research Papers/Journals

Title:	Announcing the ADVANCED ENCRYPTION STANDARD
Author(s):	Joan Daemen, Vincent Rijmen
Date of Conference:	November 26, 2001
Published in:	National Institute of Standards and Technology (NIST)
Location:	Gaithersburg

Summary:

This is the official paper of AES by Federal Information Processing Standards describing the AES encryption algorithm. The algorithm specified in this standard may be implemented in software, firmware, hardware, or any combination thereof. The specific implementation may depend on several factors such as the application, the environment, the technology used, etc.

This paper explains AES algorithm completely and also gives examples showing each step of the algorithm.The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext. Since cryptographic security depends on many factors besides the correct implementation of an encryption algorithm, Federal Government employees, and others, should also refer to NIST Special Publication 800-21, Guideline for Implementing Cryptography in the Federal Government.

Title:	ADVANCED ENCRYPTION STANDARD
Author(s):	Douglas Selent, Student, M.S. Program in Computer Science, Rivier College
Date of Conference:	2nd July, 2010
Published in:	RIVIER ACADEMIC JOURNAL, VOLUME 6, NOVEMBER 2, 2010
Conference Location:	New Hampshire, USA

Summary:

Advanced Encryption Standard (AES) is the current standard for secret key encryption. AES was created by two Belgian cryptographers, Vincent Rijmen and Joan Daemen, replacing the old Data Encryption Standard (DES). The Federal Information Processing Standard 197 used a standardized version of the algorithm called Rijndael for the Advanced Encryption Standard. The algorithm uses a combination of Exclusive-OR operations (XOR), octet substitution with an S-box, row and column rotations, and a MixColumn. It was successful because it was easy to implement and could run in a reasonable amount of time on a regular computer.

The Rijndael algorithm was chosen as the new Advanced Encryption Standard (AES) for several reasons. The purpose was to create an algorithm that was resistant against known attacks, simple, and quick to code. Choosing to use field GF(2^8) was a very good decision. The inverse of the addition operation was itself, making much of the algorithm easy to do. In fact, every operation is invertible by design. In addition, the block size and key size can vary making the algorithm versatile.

AES was originally designed for non-classified U.S. government information, but, due to its success, AES-256 is usable for top secret government information. As of July 2009, no practical attacks have been successful on AES.

Title:	Advanced Encryption Standard (AES) Instructions Set
Author(s):	Shay Gueron
Date of Conference:	July 14, 2008
Published in:	Intel Mobility Group,
Conference Location:	Israel Development Center, Israel

Summary:

The Advanced Encryption Standard (AES) is the United States Government standard for Symmetric encryption, defined by FIPS Publication #197 (2001). It is used in a large variety of applications, and high throughput and security are required. Intel offers a new set of Single Instruction Multiple Data (SIMD) instructions that will be introduced in the next generation of the Intel® processor family. These instructions enable fast and secure encryption and decryption using AES.

The new architecture introduces six Intel SSE instructions. Four instructions, namely AESENC, AESENCLAST, AESDEC, and AESDELAST facilitate high performance AES encryption and decryption. The other two, namely AESIMC and AESKEYGENASSIST, support the AES key expansion procedure.

Together, these instructions provide a full hardware for support AES, offering security, high performance, and a great deal of flexibility.

This white paper provides an overview of the AES algorithm and guidelines for utilizing the AES instructions to achieve high performance and secure AES processing. Some special usage models of this architecture are also described.

Title: MOBILE APPLICATION SECURITY ON ANDROID

Author(s): Jesse Burns

Date of Conference: June, 2009

Published in: Black Hat USA talk

Conference Location: United States of America

Summary:

Android has a unique security model, which focuses on putting the user in control of the device. Android devices however, don't all come from one place, the open nature of the platform allows for proprietary extensions and changes. These extensions can help or could interfere with security, being able to analyze a distribution of Android is therefore an important step in protecting information on that system. This paper takes the reader through the security model of Android, including many of the key security mechanisms and how they protect resources. This background information is critical to being able to understand the tools Jesse will be presenting at Black Hat, and the type of information you can glean from the tools, and from any running Android distribution or application you wish to analyze.

Before reading this paper you should already be familiar with Android's basic architecture and major abstractions including: Intents, Activities, BroadcastReceivers, Services, ContentProviders and Binder1. If you haven't yet encountered most of these platform features you might want to start with the Notepad2 tutorial. As Android is open source you should also have this code available to you. Both the java and C code is critical for understanding how Android works, and is far more detailed than any of the platform documentation. As discussed in the Black Hat talk, the difference between what the documentation and the source say is often critical to understanding the systems security.

Title:	A Study of Android Application Security
Author(s):	William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri
Date of Conference:	13-15 August 2014
Published in:	The Pennsylvania State University
Conference Location:	Pennsylvania, United States of America

Summary:

This paper tries to better understand smartphone application security by studying 1,100 popular free Android applications. Paper introduces the ded decompiler, which recovers Android application source code directly from its installation image. Paper shows design and execution of a horizontal study of smartphone applications based on static analysis of 21 million lines of recovered code.

Analysis uncovered pervasive use/misuse of personal/phone identifiers, and deep penetration of advertising and analytics networks. However, in paper there is no evidence of malware or exploitable vulnerabilities in the studied applications. Paper concludes by considering the implications of these preliminary findings and offer directions for future analysis.

The rapid growth of smartphones has led to a renaissance for mobile services. Go-anywhere applications support a wide array of social, financial, and enterprise services for any user with a cellular data plan. Application markets such as Apple’s App Store and Google’s Android Market provide point and click access to hundreds of thousands of paid and free applications. Markets streamline software marketing, installation, and update—therein creating low barriers to bring applications to market, and even lower barriers for users to obtain and use them.

Title:	Android Security 2014 Year in Review
Date of Conference:	15 December 2014
Published by:	Google Inc.
Conference Location:	Mountain View, California, United States

Summary:

In 2014, the Android platform made numerous significant improvements in platform security technology, including enabling deployment of full disk encryption, expanding the use of hardware protected cryptography, and improving the Android application sandbox with a SE Linux based Mandatory Access Control system (MAC). Developers were also provided with improved tools to detect and react to security vulnerabilities, including the no go to fail project and the Security Provider. We provided device manufacturers with ongoing support for fixing security vulnerabilities in devices, including development of 79 security patches, and improved the ability to respond to potential vulnerabilities in key areas, such as the updateable Web View in Android 5.0.

Google's security services for Android increased protection for users and improved visibility into attempts to exploit Android. Ongoing monitoring by Verify Apps found that efforts to deliver Potentially Harmful Applications (PHAs) continued at low levels throughout 2014, less than 1% of all devices had a PHA installed. Less than 0.15% of devices that download only from Google Play had a PHA installed. Expanded protection in Verify Apps and safe browsing also now provides insight into

Platform, network, and browser vulnerabilities affecting Android devices. Exploitation attempts were tracked for multiple vulnerabilities, and the data does not show any evidence of widespread exploitation of Android devices.

Title:	Understanding Android Security
Author(s):	William Enck, Machigar Ongtank and Patrick Mcdaniel
Date of Conference:	13 August 2012
Published in:	The Pennsylvania State University
Conference Location:	Pennsylvania, United States of America

Summary:

Developed by the Open Handset Alliance (led by Google), Android is a widely anticipated open source operating system for mobile devices that provides a base operating system, an application middleware layer, a Java software development kit (SDK), and a collection of system applications.

Although the Android SDK has been available since late 2007, the first publicly available Android ready “G1” phone debuted in late October 2008. Since then, Android’s growth has been phenomenal: T-Mobile’s G1 manufacturer HTC estimates shipment volumes of more than 1 million phones by the end of 2008, and industry insiders expect public adoption to increase steeply in 2009.

One of Android’s chief selling points is that it lets developers seamlessly extend online services to phones. The most visible example of this feature is, unsurprisingly, the tight integration of Google’s Gmail, Calendar, and Contacts Web applications with system utilities. Android users simply supply a username and password, and their phones automatically synchronize with Google services.

Other vendors are rapidly adapting their existing instant messaging, social networks, and gaming services to Android, and many enterprises are looking for ways to integrate their own internal operations (such as inventory management, purchasing, receiving, and so forth) into it as well.

2.2 Integrated Summary of Literature Review

By looking into all the applications available on Play Store, it becomes absolutely clear that there’s still a lot to be done in the field of security. There have been many attempts to overcome this barrier by using the permission in Apps but still the available applications such as MacAfee Antivirus are still not the perfect applications. The real time security is yet to be achieved and the emerging hardware standards and the continuous improvement in technology can surely overcome these issue.

In addition, the data security has not been taken up by many and there’s still a lot of scope of research and improvement needed in this space which can improves it to a great extent. This all motivated us to explore into this technology and make our contribution.

To sum up there have been a lot of connectivity options available in today’s smartphones, yet internet has been used widely but it has an additional data cost and having the bandwidth limitation especially in the developing countries like India is not the most feasible connectivity platform for heavy files will have security concern.

Therefore, narrowing our study to the other’s, we are left the Wi-Fi, Bluetooth, NFC, WLAN etc., which will provide much more transfer speed and being absolutely free as no additional cost being involved, are much better than Internet. Moreover, after studying these resources it become absolutely clear that Wi-Fi is still the most reliable and still desired connectivity parameter due to its speed, reliability and comparatively much better range than other connectivity parameters that are pre-equipped in the latest handsets.

CHAPTER 3 – ADVANCED ENCRYPTION STANDARD (AES)

As mentioned earlier we are going to use AES encryption algorithm and develop a system according to it in java. So following is the Advanced Encryption Standard (AES) algorithm.

3.1 Input and Output

Input and output to an AES algorithm is sequence of 128 bits .These sequences will sometimes be referred to as blocks and the number of bits they contain will be referred to as their length. The Cipher Key for the AES-128 algorithm is a sequence of 128 bits.

The bits within such sequences will be numbered starting at zero and ending at one less than the sequence length (block length or key length). The number i attached to a bit is known as its index and will be in range $0 <= i < 128$.

3.2 Bytes

The basic unit for processing in the AES algorithm is a byte, a sequence of eight bits treated as a single entity. The input, output and Cipher Key bit sequences described are processed as arrays of bytes that are formed by dividing these sequences into groups of eight contiguous bits to form arrays of bytes.

For an input, output or Cipher Key denoted by a, the bytes in the resulting array will be referenced using one of the two forms, a_n or $a[n]$, where n will be in the following range:

Key length = 128 bits, $0 <= n < 16$; Block length = 128 bits, $0 <= n < 16$;

All byte values in the AES algorithm will be presented as the concatenation of its individual bit values (0 or 1) between braces in the order {*b*7, *b*6, *b*5, *b*4, *b*3, *b*2, *b*1, *b*0}.

3.3 Array of Bytes

Arrays of bytes will be represented in the following form: a_0, a_1 … a_{15}

The bytes and the bit ordering within bytes are derived from the 128-bit input sequence

Input0 input1 input2 … input126 input127 as follows:

a0 = {input0, input1, …, input7};

a1 = {input8, input9, …, input15};

:

a15 = {input120, input121, …, input127}.

3.4 The State

Nb: Number of columns (32-bit words) comprising the State. For this standard, Nb = 4.

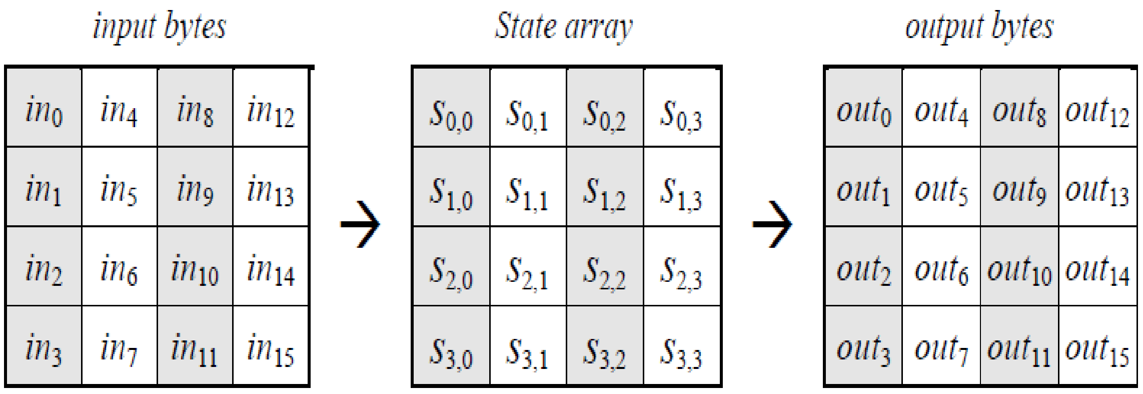
Nk: Number of 32-bit words comprising the Cipher Key. For this standard, Nk = 4.

Nr: Number of rounds, which is a function of Nk and Nb (which is fixed). For this standard, Nr = 10.

Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the State. The State consists of four rows of bytes, each containing Nb bytes, where Nb is the block length divided by 32.

In the State array denoted by the symbol s, each individual byte has two indices, with its row number r in the range 0<=r < 4 and its column number c in the range 0 <= c < Nb. This allows an individual byte of the State to be referred to as either sr,c or s[r,c]. For this standard, Nb=4, i.e., 0 <=c < 4.

At the start of the Cipher and Inverse Cipher described later the input – the array of bytes in0, in1, ……… in15 – is copied into the State array as illustrated in figure. The Cipher or Inverse Cipher operations are then conducted on this State array, after which its final value is copied to the output – the array of bytes out0, out1, ………………,out15.



in_0	in_4	in_8	in_{12}
in_1	in_5	in_9	in_{13}
in_2	in_6	in_{10}	in_{14}
in_3	in_7	in_{11}	in_{15}

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

out_0	out_4	out_8	out_{12}
out_1	out_5	out_9	out_{13}
out_2	out_6	out_{10}	out_{14}
out_3	out_7	out_{11}	out_{15}

Figure 4: State array input and output

3.5 Algorithm Specification

For the AES algorithm, the length of the input block, the output block and the State is 128 bits. This is represented by Nb = 4, which reflects the number of 32-bit words (number of columns) in the State.

For the AES-128 algorithm, the length of the Cipher Key, K, is 128 bits. The key length is represented by Nk = 4, which reflects the number of 32-bit words (number of columns) in the Cipher Key.

For the AES algorithm, the number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by Nr, where Nr = 10 when Nk = 4.

3.6 Cipher

At the start of the Cipher, the input is copied to the State array using the conventions described earlier. After an initial Round Key addition, the State array is transformed by implementing a round function 10 times, with the final round differing slightly from the first Nr -1 rounds. The final State is then copied to the output as described later.

The round function is parameterized using a key schedule that consists of a one-dimensional array of four-byte words derived using the Key Expansion routine described later.

The Cipher is described in the pseudo code. The individual transformations **-SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey()** – process the State and are described in the following subsections. In Fig, the array **w[]** contains the key schedule, which is described later.

As shown in algo, all Nr rounds are identical with the exception of the final round, which does not include the MixColumns()transformation.

Cipher pseudo code

```
Cipher(byte in [4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
  byte state[4,Nb]
  state = in
  AddRoundKey(state, w[0, Nb-1])
  For round = 1 step 1 to Nr-1
     SubBytes(state)
     ShiftRows(state)
     MixColumns(state)
     AddRoundKey(state, w[round * Nb, (round + 1) * Nb-1])
  End for
  SubBytes(state)
  ShiftRows(state)
  AddRoundKey(state, w[Nr * Nb, (Nr + 1) * Nb-1])
  out = state
end
```

3.6.1 SubBytes( ) Transformation

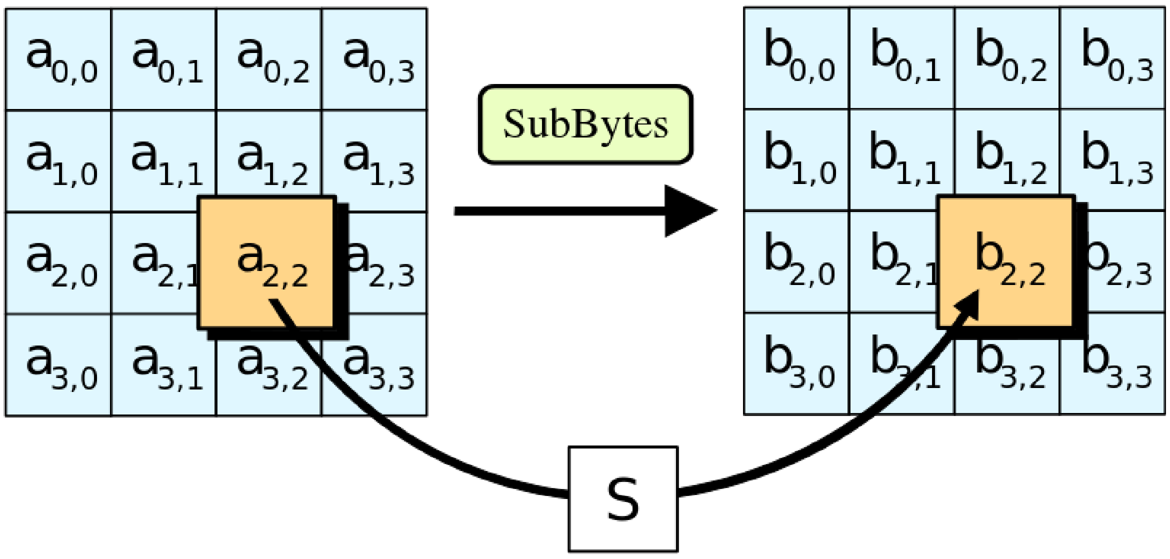
In the SubBytes step, each byte in the state matrix is replaced with a SubByte $S(a_{i,j})$ using an 8-bitsubstitution box, the Rijndael S-box. This operation provides the non-linearity in the cipher. The S-box used is derived from the multiplicative inverse over GF(2^8), known to have good non-linearity properties. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation.

The S-box is also chosen to avoid any fixed points (and so is a derangement), i.e., $S(a_{i,j}) \neq a_{i,j}$, and also any opposite fixed points, i.e., $S(a_{i,j}) \oplus a_{i,j} \neq 0\mathrm{xFF}$. While performing the decryption, Inverse SubBytes step is used, which requires first taking the affine transformation and then finding the multiplicative inverse (just reversing the steps used in SubBytes step).

The SubBytes() transformation is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (S-box). This S-box, which is invertible, is constructed by composing two transformations:

1) Take the multiplicative inverse in the finite field GF(2^8), described in Sec. 4.2; the element {00} is mapped to itself.
2) Apply the following affine transformation (over GF(2)): for 0 <=i <8, where bi is the ith bit of the byte, and ci is the ith bit of a byte c with the value {63} or {01100011}. Here and elsewhere, a prime on a variable indicates that the variable is to be updated with the value on the right.

$$b_i' = b_i \oplus b_{(i+4)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+6)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus c_i$$



$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
SubBytes
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
S

Figure 5: SubBytes transformation

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 6: S-Box with hexadecimal values

3.6.2 ShiftRows() Transformation

The ShiftRows step operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively. For blocks of sizes 128 bits and 192 bits, the shifting pattern is the same. Row n is shifted left circular by n-1 bytes. In this way, each column of the output state of the ShiftRows step is composed of bytes from each column of the input state. (Rijndael variants with a larger block size have slightly different offsets).

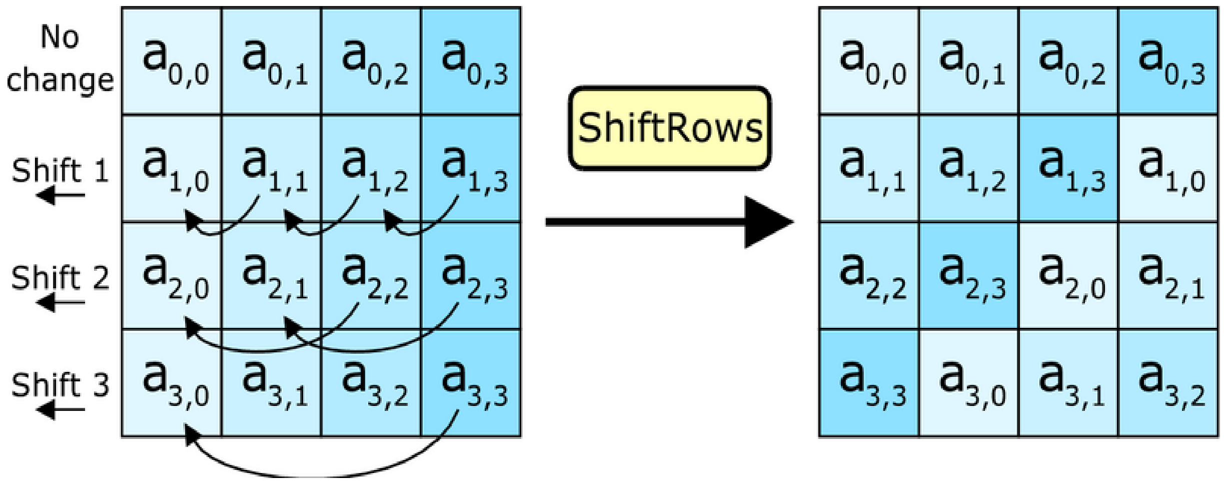
For a 256-bit block, the first row is unchanged and the shifting for the second, third and fourth row is 1 byte, 3 bytes and 4 bytes respectively—this change only applies for the Rijndael cipher when used with a 256-bit block, as AES does not use 256-bit blocks. The importance of this step is to avoid the columns being linearly independent, in which case, AES degenerates into four independent block ciphers.

In the ShiftRows() transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, r = 0, is not shifted. Specifically, the ShiftRows()transformation proceeds as follows:

$$s'_{r,c} = s_{r,(c+shift(r,Nb)) \bmod Nb} \quad \text{for } 0 < r < 4 \quad \text{and} \quad 0 \le c < Nb,$$

where the shift value shift(r,Nb) depends on the row number, r, as follows (recall that Nb = 4):

shift(1,4) =1; shift(2,4) =2; shift(3,4) =3 .This has the effect of moving bytes to “lower” positions in the row (i.e., lower values of c in a given row), while the “lowest” bytes wrap around into the “top” of the row (i.e., higher values of c in a given row). Figure 7 illustrates the ShiftRows() transformation.


No change
Shift 1
Shift 2
Shift 3
ShiftRows
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,0}$
$a_{2,2}$ $a_{2,3}$ $a_{2,0}$ $a_{2,1}$
$a_{3,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$
Figure 7: ShiftRows() transformation

3.6.3 MixColumns() Transformation

In the MixColumns step, the four bytes of each column of the state are combined using an invertible linear transformation. The MixColumns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes. Together with ShiftRows, MixColumns provides diffusion in the cipher.

During this operation, each column is transformed using a fixed matrix (matrix multiplied by column gives new value of column in the state):

Matrix multiplication is composed of multiplication and addition of the entries. Entries are 8 bit bytes treated as coefficients of polynomial of order x7. Addition is simply XOR. Multiplication is modulo irreducible polynomial x8+x4+x3+x+1.

If processed bit by bit then after shifting a conditional XOR with 0x1B should be performed if the shifted value is larger than 0xFF (overflow must be corrected by subtraction of generating polynomial). These are special cases of the usual multiplication in GF(2^8).

In more general sense, each column is treated as a polynomial over GF(2^8) and is then multiplied modulo x4+1 with a fixed polynomial c(x) = 0x03 · x3 + x2 + x + 0x02. The coefficients are displayed in their hexadecimal equivalent of the binary representation of bit polynomials from GF(2)[x]. The MixColumns step can also be viewed as a multiplication by the shown particular MDS matrix in the finite field GF(2^8). This process is described further in the article Rijndael mix columns.

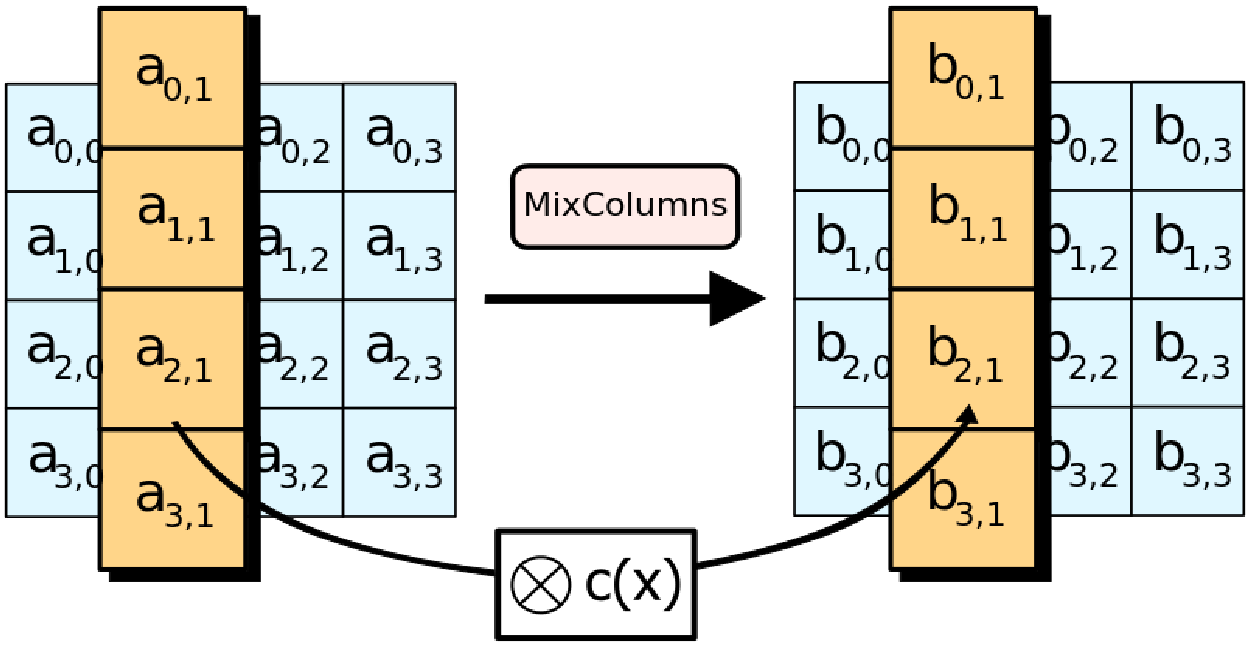


Figure 8: MixColumns() transformation

3.6.4 AddRoundKey() Transformation

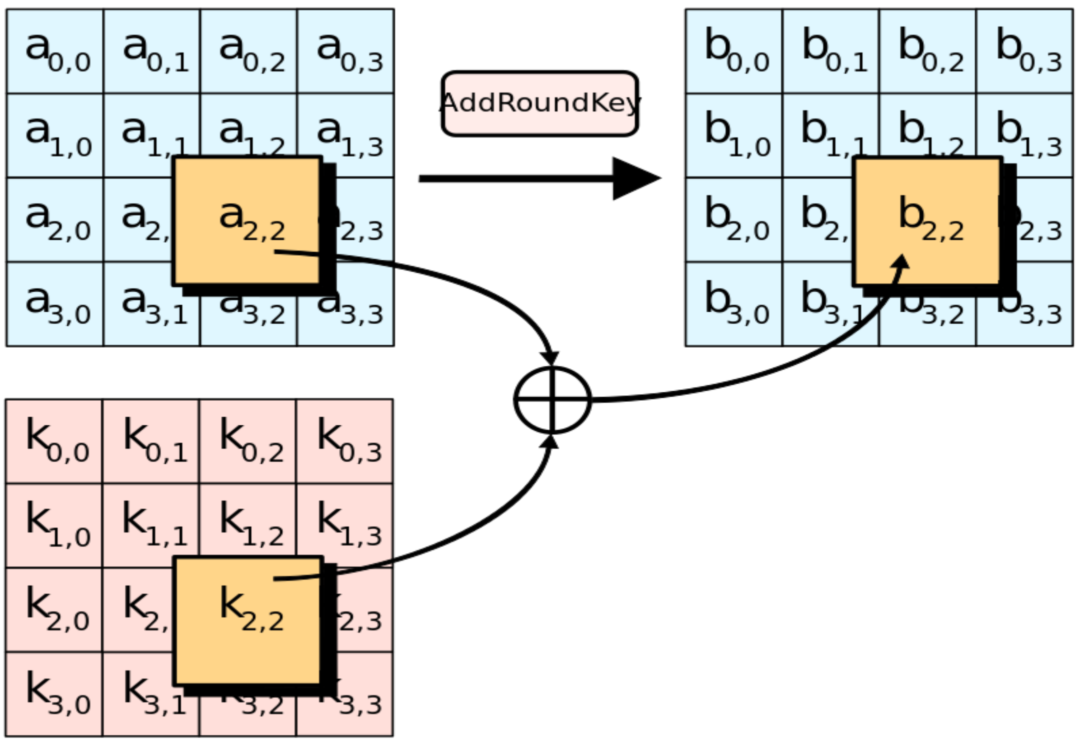
In the AddRoundKey() transformation, a Round Key is added to the State by a simple bitwise

XOR operation. Each Round Key consists of Nb words from the key schedule. Those Nb words are each added into the columns of the State, where [wi] are the key schedule words described, and round is a value in the range 0 <=round <=Nr. In the Cipher, the initial Round Key addition occurs when round = 0, prior to the first application of the round function.

The application of the AddRoundKey() transformation to the Nr rounds of the Cipher occurs when 1 <=round <=r.

The action of this transformation is illustrated in Fig. 9, where l = round * Nb. The byte address within words of the key schedule was described.

In the AddRoundKey step, the subkey is combined with the state. For each round, a subkey is derived from the main keyusing Rijndael's key schedule; each subkey is the same size as the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR.



$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
AddRoundKey
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,0}$ $k_{2,2}$ $k_{2,3}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$

Figure 9: AddRoundKey() transformation

3.7 Key Expansion

The AES algorithm takes the Cipher Key, K, and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of Nb (Nr + 1) words: the algorithm requires an initial set of Nb words, and each of the Nr rounds requires Nb words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted [wi], with i in the range 0 <=i < Nb(Nr + 1).

The expansion of the input key into the key schedule proceeds according to the pseudo code

SubWord() is a function that takes a four-byte input word and applies the S-box to each of the four bytes to produce an output word. The function RotWord() takes a word [a0,a1,a2,a3] as input, performs a cyclic permutation, and returns the word [a1,a2,a3,a0]. The round constant word array, Rcon[i], contains the values given by [xi-1,{00},{00},{00}], with x i-1 being powers of x (x is denoted as {02}) in the field GF(2^8), as discussed (note that i starts at 1, not 0).

From code it can be seen that the first Nk words of the expanded key are filled with the Cipher Key. Every following word, w[i], is equal to the XOR of the previous word, w[i-1], and the word Nk positions earlier, w[i-Nk]. For words in positions that are a multiple of Nk, a transformation is applied to w[i-1] prior to the XOR, followed by an XOR with a round constant, Rcon[i]. This transformation consists of a cyclic shift of the bytes in a word (RotWord()), followed by the application of a table lookup to all four bytes of the word (SubWord()).

Key Expansion pseudo Code

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    word temp
    i = 0
    while (i < Nk)
        w[i]=word(key[4*i],key[4*i+1],key[4*i+2],key[4*i+3])
        i = i+1
    end while
    i = Nk
    while (i < Nb * (Nr+1)]
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp)
        end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
    end while
end
```

3.8 Inverse Cipher

The Cipher transformations can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the AES algorithm. The individual transformations used in the Inverse Cipher -InvShiftRows(), InvSubBytes(),InvMixColumns(), and AddRoundKey() – process the State and are described in the following subsections.

Inverse Cipher pseudo Code

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]
    state = in
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

    for round = Nr-1 step -1 down to 1
        InvShiftRows(state)
        InvSubBytes(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
        InvMixColumns(state)
    end for

    InvShiftRows(state)
    InvSubBytes(state)

    AddRoundKey(state, w[0, Nb-1])

    out = state

end
```

3.8.1 InvShiftRows() Transformation

InvShiftRows() is the inverse of the ShiftRows() transformation. The bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, r = 0, is not shifted. The bottom three rows are cyclically shifted by $\boldsymbol{Nb} - shift(r, Nb)$ bytes, where the shift value shift(r,Nb) depends on the row number, and is given in equation

Specifically, the InvShiftRows() transformation proceeds as follows:

$$s'_{r,(c+shift(r,Nb)) \bmod Nb} = s_{r,c} \text{ for } 0 < r < 4 \text{ and } 0 \leq c < \boldsymbol{Nb}$$

Figure 10 illustrates the InvShiftRows() transformation

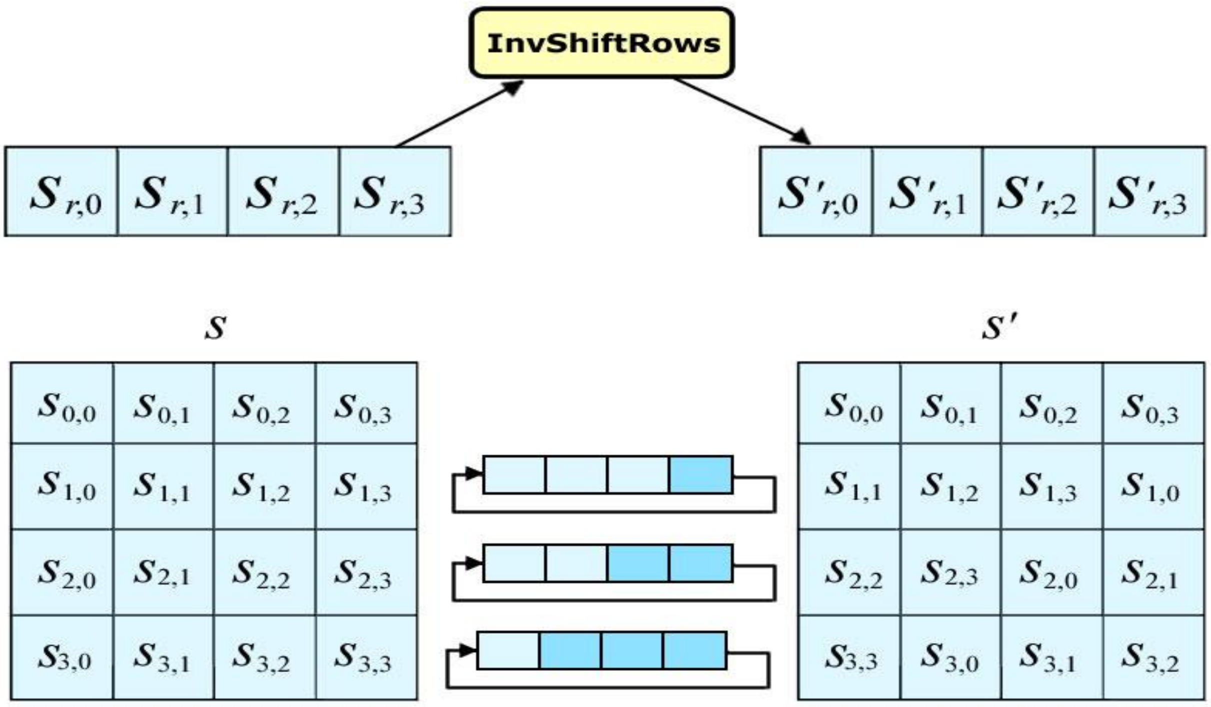

InvShiftRows
$S_{r,0}$ $S_{r,1}$ $S_{r,2}$ $S_{r,3}$
$S'_{r,0}$ $S'_{r,1}$ $S'_{r,2}$ $S'_{r,3}$
S
S'
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,1}$ $S_{1,2}$ $S_{1,3}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,1}$ $S_{1,2}$ $S_{1,3}$ $S_{1,0}$
$S_{2,2}$ $S_{2,3}$ $S_{2,0}$ $S_{2,1}$
$S_{3,3}$ $S_{3,0}$ $S_{3,1}$ $S_{3,2}$

Figure 10: InvShiftRows() transformation

3.8.2 InvSubBytes() Transformation

InvSubBytes() is the inverse of the byte substitution transformation, in which the inverse S-box is applied to each byte of the State. This is obtained by applying the inverse of the affine transformation followed by taking the multiplicative inverse in GF(2^8).

The inverse S-box used in the InvSubBytes()transformation is presented in Fig. 11:

Figure 12 shows InvSubBytes() transformation.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
x	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 11: InvSubBytes() transformation inv S-Box

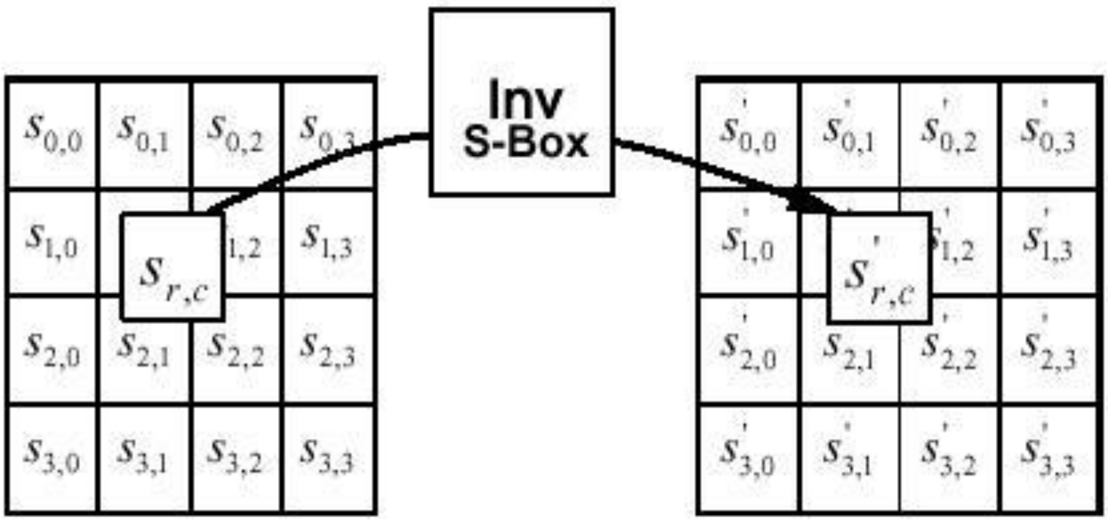

$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,2}$ $S_{1,3}$
$S_{r,c}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$
Inv S-Box
$S'_{0,0}$ $S'_{0,1}$ $S'_{0,2}$ $S'_{0,3}$
$S'_{1,0}$ $S'_{1,2}$ $S'_{1,3}$
$S'_{r,c}$
$S'_{2,0}$ $S'_{2,1}$ $S'_{2,2}$ $S'_{2,3}$
$S'_{3,0}$ $S'_{3,1}$ $S'_{3,2}$ $S'_{3,3}$

Figure 12:InvSubBytes() transformation

3.8.3 InvMixColumns() Transformation

InvMixColumns() is the inverse of the **MixColumns()** transformation. **InvMixColumns()** operates on the State column-by-column, treating each column as a four-term polynomial as described in Sec. 4.3. The columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a^{-1}(x)$, given by

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}. \quad (5.9)$$

As described in Sec. 4.3, this can be written as a matrix multiplication.

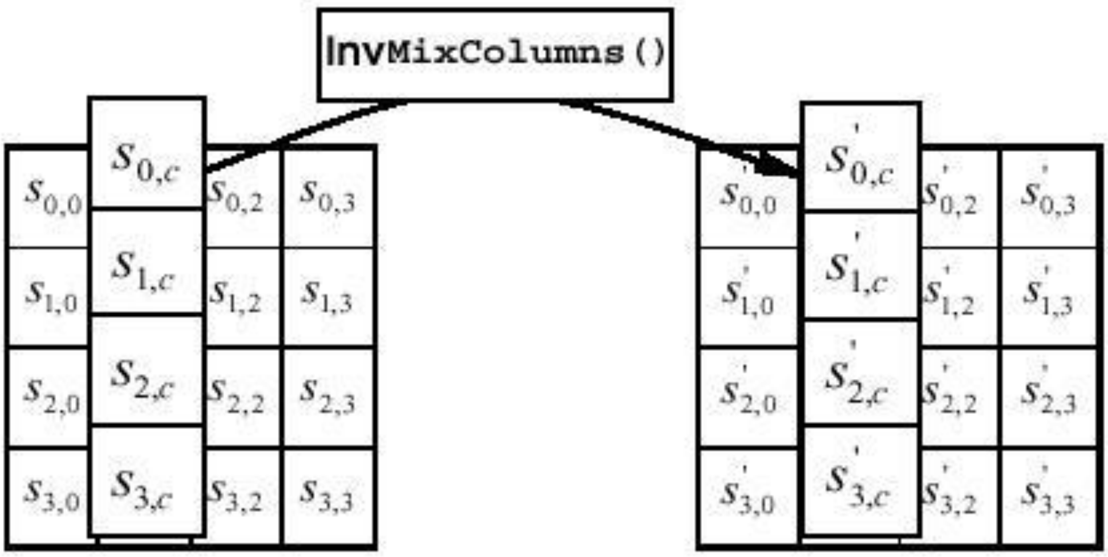

InvMixColumns()
$s_{0,0}$ $s_{0,c}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,c}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,c}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,c}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,c}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,c}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,c}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,c}$ $s'_{3,2}$ $s'_{3,3}$

Figure 13: InvMixColumns() transformation

CHAPTER 4 – PROJECT DEVELOPMENT

4.1 Project Design

Project is implementation of AES algorithm to encrypt data which is going to the database of an android application and decrypt data which is coming out of the database to the application.

This AES code is developed in Java and code can be implemented for any application of android. AES code can also be used for web application developed in JSP.

To demonstrate AES encryption in android I will build an android application. This is a simple application. Application contains only one activity i.e. Main_Activity in its UI. A table is maintained in the SQLite database to save the entries made. Activity contains two Edit Text boxes, three Buttons and three Text boxes.

From the top there is one Edit text box then a Button then a Text box. In this Edit text box you can enter a string that is encrypted and stored in the table on pressing the button. In text box the encrypted cipher will be shown to demonstrate what value is entered in the table. After this some space is left and then there is an edit text box, button and text box in the same order. In edit text box you enter an index of the table and fetch the value in that index. The value that is fetched is a cipher which is to be decrypted. On pressing the button data is fetched from the table's specified index and it is decrypted. This decrypted value is shown in the text box below the button. After this is the third button and the text box. On pressing this button whole table is shown and the values that are in it are the encrypted ones. Diagram shown on the next page shows the design of the application.

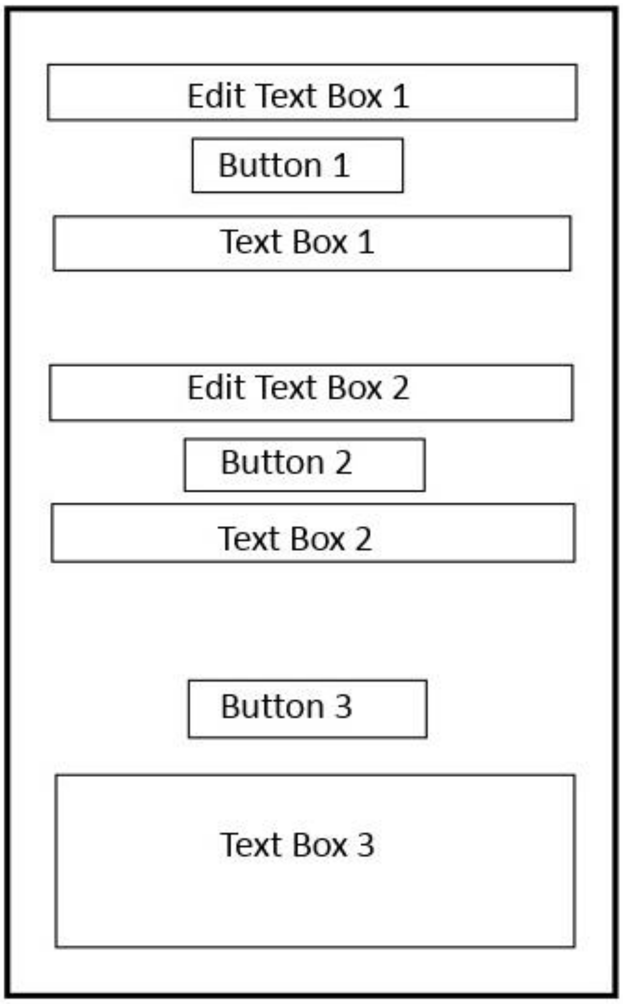
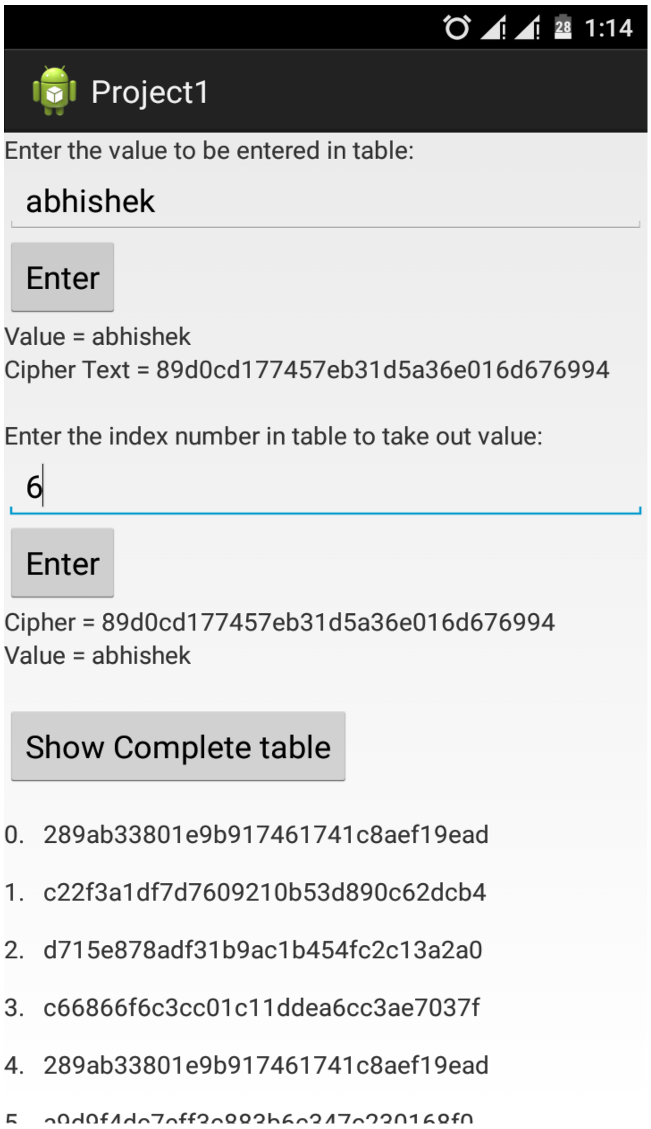


Figure 14: Application Interface

4.2 Project Implementation

As explained earlier AES algorithm is implemented and an application is used to demonstrate its use in SQLite database of android. AES code is written in Java. Code contains two functions that are most important. First is getAESEncryption() which takes two strings , first is the string that needs to be encrypted and second is the key used to encrypt. Second is getAESDecryption() which takes two strings , first is the string that needs to be decrypted and second is the key used to decrypt which is same as key used to encrypt. Code also contains functions for key expansion, sub bytes, shift rows, mix columns and add round key. Code also contains code for their inverses.

Code of AES will be included in code of the application. Application contains one activity. Table that's is in database contains only on column in which entries are made which are encrypted values. Key used is the key obtained from IEMI id of the device in which we are installing the application. 0 is added to the IEMI id to make it 16 digits i.e. 16 bytes.



1:14

Project1

Enter the value to be entered in table:

abhishek

Enter

Value = abhishek
Cipher Text = 89d0cd177457eb31d5a36e016d676994

Enter the index number in table to take out value:

6

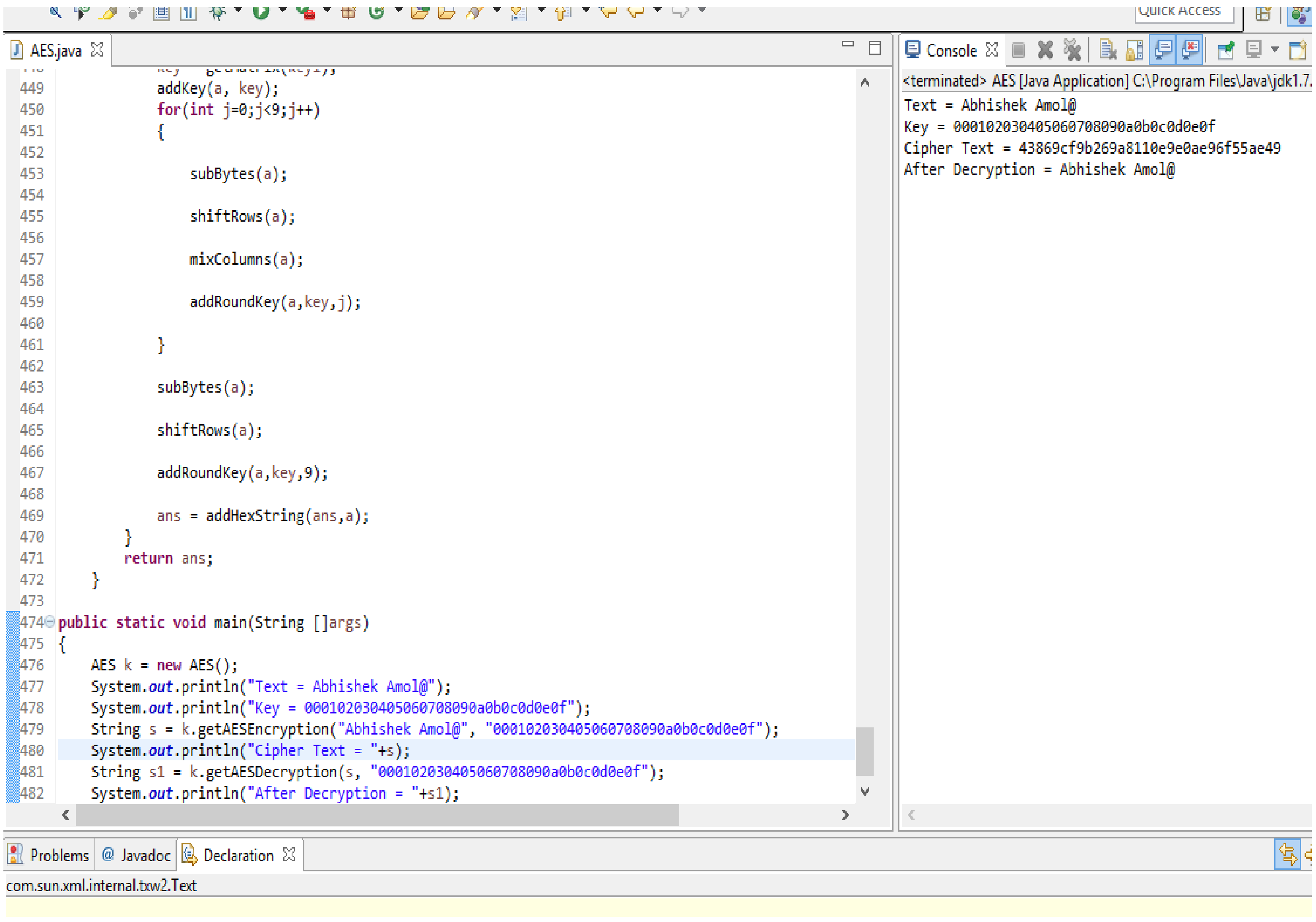
Enter

Cipher = 89d0cd177457eb31d5a36e016d676994
Value = abhishek

Show Complete table

0. 289ab33801e9b917461741c8aef19ead
1. c22f3a1df7d7609210b53d890c62dcb4
2. d715e878adf31b9ac1b454fc2c13a2a0
3. c66866f6c3cc01c11ddea6cc3ae7037f
4. 289ab33801e9b917461741c8aef19ead
5. a9d9f4dc7eff3c883b6c347c230168f0

Figure 15: Snapshot of Application



```java
        addKey(a, key);
        for(int j=0;j<9;j++)
        {

            subBytes(a);

            shiftRows(a);

            mixColumns(a);

            addRoundKey(a,key,j);

        }

        subBytes(a);

        shiftRows(a);

        addRoundKey(a,key,9);

        ans = addHexString(ans,a);
    }
    return ans;
}

public static void main(String []args)
{
    AES k = new AES();
    System.out.println("Text = Abhishek Amol@");
    System.out.println("Key = 000102030405060708090a0b0c0d0e0f");
    String s = k.getAESEncryption("Abhishek Amol@", "000102030405060708090a0b0c0d0e0f");
    System.out.println("Cipher Text = "+s);
    String s1 = k.getAESDecryption(s, "000102030405060708090a0b0c0d0e0f");
    System.out.println("After Decryption = "+s1);
```

```
<terminated> AES [Java Application] C:\Program Files\Java\jdk1.7.
Text = Abhishek Amol@
Key = 000102030405060708090a0b0c0d0e0f
Cipher Text = 43869cf9b269a8110e9e0ae96f55ae49
After Decryption = Abhishek Amol@
```

com.sun.xml.internal.txw2.Text

Figure 16: Implementation of AES in JAVA

CHAPTER 5 – PERFORMANCE ANALYSIS

5.1 Implementation Issues

5.1.1 Key Length Requirements

An implementation of the AES algorithm shall support *at least one* of the three key lengths 128, 192, or 256 bits (i.e., Nk = 4, 6, or 8, respectively). Implementation may optionally support two or three key lengths, which may promote the interoperability of algorithm implementations.

5.1.2 Keying Restrictions

No weak or semi-weak keys have been identified for the AES algorithm, and there is no restriction on key selection.

5.1.3 Parameterization of Key Length, Block Size, and Round Number

This standard explicitly defines the allowed values for the key length (Nk), block size (Nb), and number of rounds (Nr). However, future reaffirmations of this standard could include changes or additions to the allowed values for those parameters. Therefore, implementers may choose to design their AES implementations with future flexibility in mind.

5.1.4 Implementation Suggestions Regarding Various Platforms

Implementation variations are possible that may, in many cases, offer performance or other advantages. Given the same input key and data (plaintext or cipher text), any implementation that produces the same output (cipher text or plaintext) as the algorithm specified in this standard is an acceptable implementation of the AES.

5.2 Hardware used

High speed and low RAM requirements were criteria of the AES selection process. Thus AES performs well on a wide variety of hardware, from 8-bit smart cards to high-performance computers.

On a Pentium Pro, AES encryption requires 18 clock cycles per byte, equivalent to a throughput of about 11 MB/s for a 200 MHz processor. On a 1.7 GHz Pentium M throughput is about 60 MB/s.

On Intel Core i3/i5/i7 and AMD APU and FX CPUs supporting AES-NI instruction set extensions, throughput can be over 700 MB/s per thread.

5.3 Result Analysis

In AES we take 16 or multiple of 16 byte of input and key size 128, 192 or 256 bits. When input size is not multiple of 16 we have to adjust and make it multiple of 16. I have done adjustment by adding spaces at the end of the string that is to be encrypted. The table given in the next page is showing encryption of various strings and based on the table analysis will be done. Encryption is done using "000102030405060708090a0b0c0d0e0f" as key.

S.No.	Plain Text	Cipher Text
1	abhi	dc2428576dae91c0b75f766752905769
2	Abhishek Amola	0eaf8f89a43229566c3955a3bb341d95
3	abhishekamola123	04f8b10911fe8d6f7dd172513b9b84fe
4	abhishekamola1234	04f8b10911fe8d6f7dd172513b9b84fe6e85fa38563e1934f2164f94f9ada068
5	Abhishek Amola Ankit Sharma 12	396327e0a4b56e27fdb8f75880442bbbbc9b43887b7db61337afabcb9b7d7975

In the given table the first two entries are less than 16 byte so spaces are added after them and we get 16 byte cipher text. Third entry has exact 16 bytes so result is 16 byte cipher. In fourth entry I just added one letter more i.e. ‘4’ and we get cipher of 32 byte because by adding ‘4’ length of plain text increased from 16 to 17 bytes and to get 17 bytes encrypted we need to make it 32 byte by adding spaces at the end. Last entry is 32 bytes long and cipher text is also 32 byte long. This will increase the complexity of the code. Complexity analysis is done below.

5.4 Complexity Analysis

Advanced Encryption Standard (AES) is working on **fixed block size**, so its independent of input, thus for short texts its O(1).

Considering long messages, let length of plain text be n. In the code we send message and key to the encrypting function. Firstly spaces are added to plaintext to make it multiple of 16 bytes. This will increase the complexity of the code. Then a for loop runs from 1 to n/16 which picks 16 bytes on each iteration and perform key expansion, sub bytes, shift rows, mix columns and add round key. Key expansion depends on length of key which is constant. All operations are having limited iteration loops which are counted as constant. So basically complexity depends on n (length of plain text). The spaces we added at the starting of the code increases the length of the plain text. As length of plaintext increases the complexity of code also increases. This is extra factor that I came across while writing the code. For large values of n its effect on complexity will be negligible. So complexity comes out to be O(n). Same is the case of decryption it also has complexity of O(n).

Overall complexity of the AES comes out to be O(n) where n is length of plain text. Android application has responsibility of executing query, storing the data and retrieving the data so complexity of AES is the overall complexity.

CHAPTER 6 – CONCLUSION

We are making a system for android application which can secure data by using encryption and decryption. For encryption we are using Advanced Encryption Standard (AES - 128) algorithm because it’s the best and most secure algorithm known till date.

Report starts with the introduction to the project. In introduction we have covered android security, type of security and features provided by android. Then there is problem statement and objective. Then methodology which states AES is being use and describes some features of AES.

Then we move on to Literature survey. We have mentioned 7 paper which we are using to make the project. Most important of them is AES FIPS -197 which is original paper of AES.

After Literature survey we moved on to the chapter where AES is described in elaborated form. Cipher, Key Expansion and Inverse Cipher are explained. Then comes Project Development in which there is design of the project and its implementation. Then comes the performance analysis of the project. Major part of project is AES so it’s the performance analysis of AES.

All big vendors are already using security features in their applications. This will help us give security to new vendors. This will be extended to sever where data is stored and can be done for the data like audio, image, etc.

References

Papers:-

[1] Joan Daemen, Vincent Rijmen, "*Announcing the ADVANCED ENCRYPTION STANDARD*". Gaithersburg, USA, National Institute of Standards and Technology (NIST), November 26, 2001.

[2] Douglas Selent, "ADVANCED ENCRYPTION STANDARD," New Hampshire, USA, RIVIER ACADEMIC JOURNAL, VOLUME 6. November 2, FALL 2010

[3] Shay Gueron, "Advanced Encryption Standard (AES) Instructions Set*".* Israel Development Center, Israel, Intel Mobility Group. July 14, 2008

[4] Jesse Burns, "MOBILE APPLICATION SECURITY ON ANDROID," in Black Hat USA talk, *United States of America*. June, 2009

[5] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri, "A Study of Android Application Security". The Pennsylvania State Universit, USA, August 2014

Websites:-

http://developer.android.com/training/articles/security-tips.html

http://searchsecurity.techtarget.com/definition/Advanced-Encryption-Standard

https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

http://csrc.nist.gov/groups/STM/cavp/documents/aes/aesval.html

http://searchsecurity.techtarget.com/definition/Advanced-Encryption-Standard

http://aesencryption.net/

http://www.webopedia.com/TERM/A/AES.html

https://www.techopedia.com/definition/1763/advanced-encryption-standard-aes

https://blog.agilebits.com/2013/03/09/guess-why-were-moving-to-256-bit-aes-keys/

http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html