EasyChair Preprint

№ 3210

Cryptographic Algorithm Analysis and Implementation

Nandkumar Niture

April 20, 2020

Cryptographic Algorithm Analysis and Implementation

By: - Nandkumar A Niture

Abstract

The impact and necessity of information security has increased exponentially over the last few decades as the denial-of-service attacks are increasing, information is being stolen, hackers are using more sophisticated and smart methods with help of agile tools for stealing sensitive information. Do small/mid-size/large corporate organizations need the security of their system? Yes. They have sensitive user data, employee data, trading data, customer data and other sensitive confidential information stored in office systems. Do common people need the security of their systems at home? Yes. They may have their taxes files, social security card information, bank account details, private pictures, marketing strategy for their small business and many more private things.

Computer cryptography was the exclusive domain for long period of time since World War II but now is practiced outside of military agencies. Cryptography is both science and art, it uses known obscurity and mathematical formulae. Cryptographic systems should have ability to assure the authenticity of source from where message gets originated and proof of complete message delivery. It is sometimes insufficient to protect ourselves from the rules and laws, but we need to protect ourselves with applying sufficient mathematical equations. So, it is individuals and legal organizations responsibility to protect their own data.

By combining the digital signature with public-key cryptography, we can develop a protocol that combines the security of encryption with the authenticity of digital signatures. The signature is the proof of authenticity. The easiest way to get someone's public key is from a secure database from somewhere. The database has to be public, so that anyone can get anyone else's public key from secure database.

This paper will aim to assess the cryptographic algorithms and their ways of implementation with the core principles of cryptographic systems where they take a plaintext message and through a set of transposition, convert the same plaintext message into ciphertext and from ciphertext to plaintext. Also covers hashing functions and hashing algorithm techniques.

This paper will also focus on the classes and use of java programming language to implement the combination of algorithms and their output along with the algorithm time complexity and problem-solving methodology. This paper can be used as prime source in the field of cyberspace to get started with the security algorithm implementations.

Keywords: Cryptography, Encryption, Decryption, Ciphertext, Plaintext, Cryptographic Algorithms, Key, Public Key, Private Key, Hashing, Brute-force, Message-Digest, Padding, Cascad, Time-Complexity, Space-Complexity, Digital-Signature,

Abbreviations

AES – Advanced Encryption Standards

SKC – Secrete Key Cryptography

DES – Data Encryption Standard

NBS – National Bureau of Standards

EFF - Electronic Frontier Foundation

IACR - International Association for Cryptologic Research

3DES – Triple DES

NIST – National Institute of Standards and Technology

MD – Message Digest

JCA – Java Cryptographic Architecture

Figures and Tables

Tables

Figures

Table of Contents

Chapter 1
Introduction

The widespread use of the Internet, more connected models, and sophisticated computer and networking technologies have created the most important and critical collateral – "data". Currently, a big percentage of data or information exchanged over the Internet today is not secure or encrypted effectively (Intel® AES-NI Performance Testing on Linux*/Java* Stack, 2012). The data security is the biggest challenge of computer and information science. The security always comes with price. The communication is not always happening in a secured way. There are multiple reasons for this issue. Applications security is always big concern in the world of data and information security. Confidential data leaked many times because of not having tight application security. In the healthcare arena, data security becomes crucial as individual PHI can be compromised to criminals who can use that valuable information to wreak havoc to interconnected systems and personal lives of innocent citizens. Examples of some commonly used healthcare workflows are data storage in EMR (Electronic Medical Records), secure messaging, physician-patient communication, lab results transferred to EMR, Patient Portals and ePrescibing (Intel® AES-NI Performance Testing on Linux*/Java* Stack, 2012).

Does increased security provide comfort to paranoid people? Or does security provide some very basic protections that we are naive to believe that we don't need? During this time when the Internet provides essential communication between literally billions of people and is used as a tool for commerce, social interaction, and the exchange of an increasing amount of personal information, security has become a tremendously important issue for every user to deal with.

There are many aspects to security and many applications, ranging from secure commerce and payments to private communications and protecting health care information. One essential aspect for secure communications is that of cryptography. But it is important to note that while cryptography is necessary for secure communications, it is not by itself sufficient. The reader is advised, then, that the topics covered here only describe the first of many steps necessary for better security in any number of situations.

1.1Cryptography

The art and science of keeping message secure is cryptography, and it is practiced by cryptographers. The art and science of breaking ciphertext, that is, seeing through disguise. The branch of mathematics encompassing both cryptography and cryptoanalysis is cryptology (Schneier, 1996).

There are five primary functions of cryptography today:

1. *Privacy/confidentiality*
2. *Integrity*
3. *Non-repudiation*
4. *Key exchange*

Keys in cryptography:

What is key? A definition of a key from ISO/IEC 10116 (2nd edition): 1997 is

A sequence of symbols that controls the operation of a cryptographic transformation (e.g. encipherment, decipherment). In practice a key is normally a string of bits used by a cryptographic algorithm to transform plain text into cipher text or vice versa. The key should be the only part of the algorithm that it is necessary to keep secret (Australia., 2000-14).

1. Public key
2. Private key
3. Symmetric key cryptography
4. Asymmetric key cryptography

In cryptography, we start with the unencrypted data, referred to as plaintext. Plaintext is encrypted into ciphertext, which will in turn (usually) be decrypted back into usable plaintext. The encryptions and decryptions are based upon the type of cryptography scheme being employed and some form of key. For those who like formulas, this process is sometimes written as:

The basic argument from computer magazine of Whitfield Diffie and Martin Hellman (Stanford University)

Initial proposed standard of transformation of block of 64 plaintexts P bits into a block of 64 ciphertext bits of C and this is governed by key K which is 56 bits (Hellman, 1977).

$$C = E_k(P)$$
$$P = D_k(C)$$

where **P** = plaintext, **C** = ciphertext, **E** = the encryption method, **D** = the decryption method, and **k** = the key.

Let's discuss the known-plaintext attack for reasons of cryptoanalysis is based on variations of the known-plaintext attack, NBS has agreed upon the system should be protected from the knows plaintext attacks.

Let's Suppose Key K is 56 bits for initial computation.

To decipher C under the K (2 ^56 keys)

Let's say one key tried each microsecond it would take 10^11 Seconds or 10^6 Days to do search on keys.

The decreasing cost of computation has serious effect on breaking the keys, where in 10 years let's say $10 million machine should be $100,000 machine and days' time will cost $25.

What is latest on the time complexity of cracking DES - searched more than 88 billion keys every second, for 56 hours, before found the right 56-bit key to decrypt the answer to the RSA challenge, which was 'It's time for those 128-, 192-, and 256-bit keys.

To prove the insecurity of DES, EFF built the first unclassified hardware for cracking messages encoded with it. EFF DES Cracker, which was built for less than $250,000. It took the machine less than 3 days to complete the challenge, shattering the previous record of 39 days set by a massive network of tens of thousands of computers. The research results are fully documented in a book published this week by EFF and O'Reilly and Associates, entitled "Cracking DES: Secrets of Encryption Research, Wiretap Politics, and Chip Design." Although the cryptographic community has understood for years that DES keys are much too small, DES-based systems are still being designed and used today. (RELEASE, 2016)

1.2Types of Cryptography

Secret Key Cryptography (SKC): Uses one single key for both encryption and decryption; also called symmetric encryption. Primarily used for privacy and confidentiality.

Example of algorithms – DES, 3DES(triple DES), AES, Blowfish, RC4, RC4.

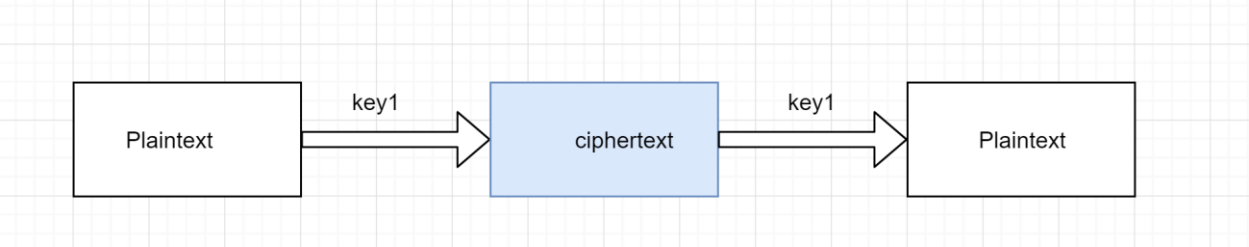


Figure 1.1: Secrete Key Cryptography

Public Key Cryptography (PKC): Uses one key for encryption and another for decryption; also called asymmetric encryption. Primarily used for authentication, non-repudiation, and key exchange.

Example of Algorithms – RSA, Deffie-Hellman

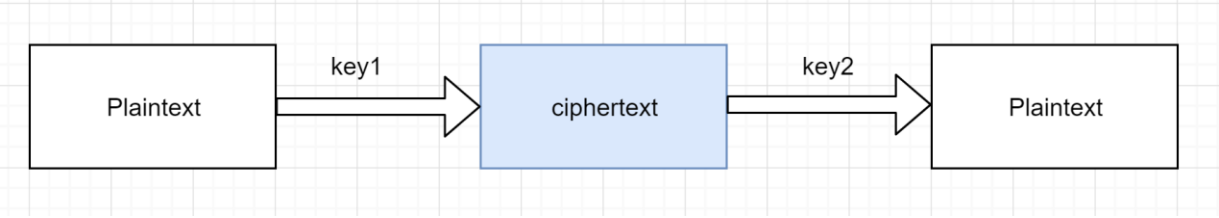


Figure 1.2: Public Key Cryptography

Hash Functions: Uses a mathematical transformation to irreversibly "encrypt" information, providing a digital fingerprint. Primarily used for message integrity.

Example of Algorithms – MD5, SHA256, SHA512.

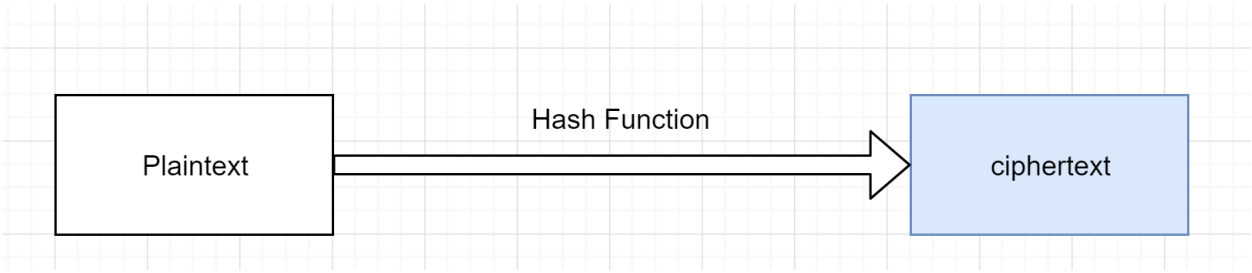


Figure 1.3: Hash Function

In digital signing, one-way hash functions are used as input for a signing algorithm. In RSA signing, a 36-byte structure of two hashes (one SHA and one MD5) is signed (encrypted with the private key) (A. Freier, 2011).

Digital Signature: A digital signature is intended to copy the hand-written signature on an important document such as contract. Its mathematical representation and conveys specific meaning in binary data.

As shown in figure digital signature can be created by encryption the entire message with the private key of the sender and even if future security is needed then the message can be encrypted with symmetric algorithm. (A. Freier, 2011)

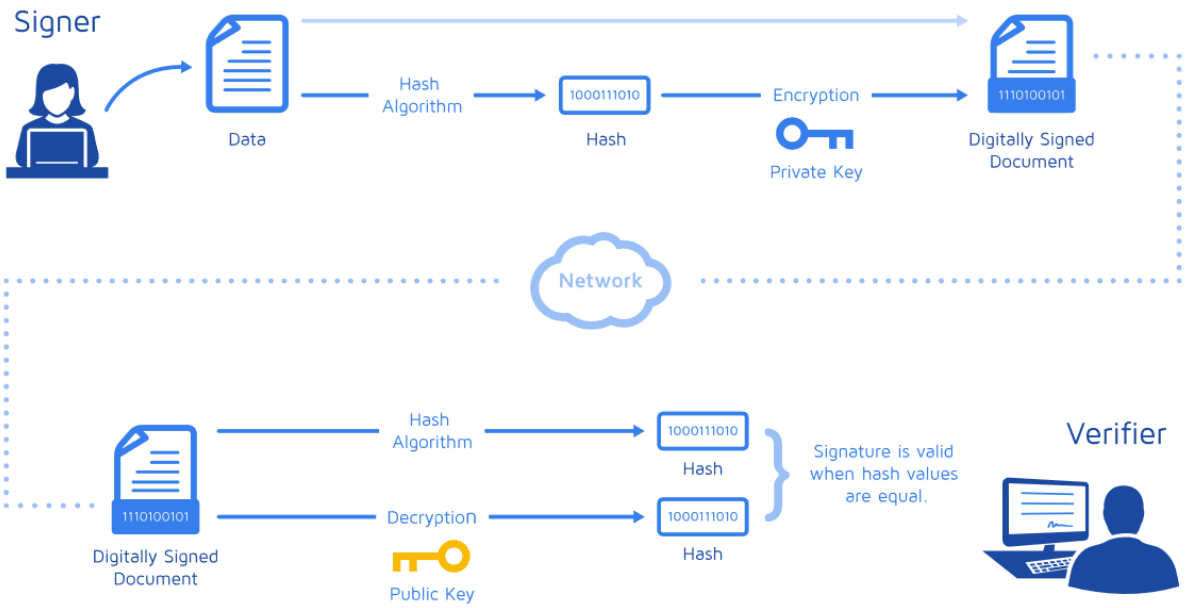


Figure 1.4: Digital Signature

1.3Cryptographic Algorithms

Symmetric		
Name of Algorithm	**Method of Algorithm**	**Key Size**
DES	64 bit block cipher	64 bit
3DES	64 bit block cipher	192 bit
Blowfish	64 bit block cipher	32 to 488 bit key
AES	128 bit block cipher	128, 192, 256 bit
Twofish	128 bit block cipher	128, 192, 256 bit
RC4	one bit per unit time	40 to 2048 bit key
RC5	block mode	from 1 to 2048
Asymmetric		
Name of Algorithm	**Method of Algorithm**	**Key Size**
RSA	key transport	1028 bit
Diffie-Hellman	key exchange	varies
El Gamal	key exchage	varies
Hashing		
Name of Algorithm	**Method of Algorithm**	**Key Size**
MD5	MD5 Block hashing	512 bit block creates 128 bit digest
SHA-1	Rivest SHA hashing	512 bit creates 160 bit digest
SHA-2	Hash	creates 224, 256, 384 bit hashes
HMAC-MD5	key digest	creates 128 bit hashes
HMAC-SHA1		creates 160 bit hashes

Table 1.1 – Cryptographic Algorithms

Table 1.1 describes the cryptographic algorithms and their categorization in symmetric, asymmetric and hashing types.
Based upon the NSIT recommendation, commonly used, key size, sharing, speed and complexity of these algorithms below table have an explanation.

Feature / Algorithm	Hash	Symmetric	Asymmetric
No. of Keys	0	1	2
NIST recommended Key length	256 bits	128 bits	2048 bits
Commonly used	SHA	AES	RSA
Key Management/Sharing	N/A	Big issue	Easy & Secure
Effect of Key compromise	N/A	Loss of both sender & receiver	Only loss for owner of Asymmetric key
Speed	Fast	Fast	Relatively slow
Complexity	Medium	Medium	High
Examples	SHA-224, SHA-256, SHA-384 or SHA-512	AES, Blowfish, Serpent, Twofish, 3DES, and RC4	RSA, DSA, ECC, Diffie-Hellman

Table 1.2 - Characteristics of the algorithms (Mehmood, 2017)

Due to the above characteristics, symmetric and asymmetric algorithms are sometimes used in a hybrid approach. Asymmetric ciphers are characteristically used for identity authentication performed via digital signatures & certificates, for the distribution of symmetric bulk encryption key, non-repudiation services and for key agreement. Symmetric ciphers are used for bulk encryption of data due to their fast speed. (Mehmood, 2017)
Hybrid approach of symmetric and asymmetric algorithms will be used in effective way to overcome the attacks and enhance the security in web applications.
This paper is going to elaborate on the use the algorithms in hybrid approach in java cryptographic architecture along with its analysis.

Problem Statement and Justification: How to prevent data breaches and secure enterprise applications using the combination of cryptographic algorithms with analysis in details. The combination of superseded algorithms will resolve the attack almost 90% in volume

The right combination of using AES, DES and other algorithms will prevent the data leak and vulnerabilities in enterprise applications. With this problem in mind, we need to analyze algorithms in detail and make the combination of these algorithm in programing language for proven security attacks.

Chapter 2
Literature Review

Security algorithms in cryptography is always very challenging and demanding area, as there are high demands to analyze for the new security standards and put the outdated as deprecated from the list. In this review I am going to present what has done so far, and the programing language standards to put the mathematical functions together.

2.1 Algorithms design history

DES and 3DES:

DES is block cipher and it encrypts data in 64-bit blocks. A 64-bit plaintext block goes as an input and 64-bit block of ciphertext comes out as output. As DES is symmetric key algorithm both the encryption and decryption use the same key and the key length is 54 bits.

Almost 30 years after first publishing DES, the National Institute of Standards and Technology (NIST) finally withdrew the standard in 2005, reflecting a long-established consensus that DES is insufficiently secure. By 2008, commercial hardware costing less than USD 15,000 could break DES keys in less than a day on average. DES is long past its sell-by date (L. Hornquist Astrand, 2012).

Some DES implementations use triple-DES since DES is not a group, as resultant ciphertext is much harder to break using exhaustive search 2^112 attempts as usually 2^56 in DES.

Plaintext P and Ciphertext C

Plaintext →DES(Key1)→DES(key2)→DES(key3)→Ciphertext

Ciphertext →DES(key3)→DES(key2)→DES(key1)→Plaintext

For a brute-force attack on 3DES, however, the outlook is far less optimistic. Consider the problem: we know C and P, and we are trying to guess k1, k2, and k3 in the following relation (Kelly, 2006)

C = E_k3(D_k2(E_k1(p)))

To guess the keys, we must execute something like the following (assuming k1, k2, and k3 are 64-bit values keys), E is encryption function, D is decryption function, C is Cyphertext

```
for ( k3 = 0 to 2^56 step 1 )
   compute C2 = D_k3(C1)
   for ( k2 = 0 to 2^56 step 1 )
      compute C3 = E_k2(C2)
      for ( k1 = 0 to 2^56 step 1 )
         begin
            compute p = D_k1(C3) xor IV
            if ( p equals p-expected )
                exit loop; we found the keys
         end
```

The correct combination should have to try 2^168 operations. But as the computing power has raised over the time, the more operations taking place per microsecond. Building triple DES cracker today is not impossible.

DES and 3DES both are very much susceptible for brute-force attacks which are well accomplished in the moderated financial circumstances. Because of this cryptographer must look out for more strong and robust algorithm to deal with the computing attacks.

For this matter the DES has deprecated and replaced by AES. Still many applications including still rely on DES for security (Kelly, 2006).

AES:

The block cipher used in AES algorithm encryption protocol is 128 bits [AES MODE]. The plaintext P is divided into 128-bit blocks in AES standard. The last block may have fewer than 128 bits, and no padding is required.

Advanced Encryption Standard (AES) as a primitive to securely encrypt plaintext key(s) with any associated integrity information and data, such that the combination could be longer than the width of the AES block size (128-bits) where ciphertext bit should be a highly non-linear function of each plaintext bit, and (when unwrapping) each plaintext bit should be a highly non-

linear function of each ciphertext bit, and it is sufficient to approximate an ideal pseudorandom permutation to the degree that exploitation of undesirable phenomena is as unlikely as guessing the AES engine key. (Schaad, 2002)

Inputs: Plaintext, n 64-bit values {P1, P2, ..., Pn}, and
Key, K (the KEK).

Outputs: Ciphertext, (n+1) 64-bit values {C0, C1, ..., Cn}.

1) Initialize variables.

Set A0 to an initial value (see 2.2.3)

For i = 1 to n

R[0][i] = P[i]

2) Calculate intermediate values.

MSB is most significant bits

LSB is least significant bits

R Register of an 64 bit

For t = 1 to s, where s = 6n

A[t] = MSB(64, AES(K, A[t-1] | R[t-1][1])) ^ t

For i = 1 to n-1

R[t][i] = R[t-1][i+1]

R[t][n] = LSB(64, AES(K, A[t-1] | R[t-1][1]))

3) Output the results.

Set C[0] = A[t]

For i = 1 to n

C[i] = R[t][i] //array of 2 dimensions.

Diffie-Hellman Algorithm –

"As the Diffie-Hellman algorithm is asymmetric key algorithm and being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters. For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables one prime P and G (a primitive root of P) and two private values a and b. P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly, the opposite person received the key and from that generates a secret key after which they have the same secret key to encrypt (Implementation of Diffie-Hellman Algorithm, 2017)"

Public Keys available = P, G

Private Key Selected = a, b

Key generated = x = G^a mod P Key generated = x = G^b mod P

Exchange of generated keys takes place

Key received = y key received = x

Generated Secret Key = k_a = y^a mod P Generated Secret Key = k_b = x^b mod P

Algebraically it can be shown that k_a = k_b

Users now have a symmetric secret key to encrypt

MD5 Algorithm –

This algorithm is hashing algorithm called Message Digest Algorithm. The algorithm takes as input message of arbitrary length and produce 128-bit output called fingerprint. This algorithm is basically intended for digital signatures applications. A large file be compressed in secure manner before encrypted with private key under public-key cryptosystem such as RSA.

It has five steps and four rounds

1. Append padding bits
2. Append length
3. Initialize Message Direst buffer
4. Process this message into 16-word blocks
5. Output

this is for processing 16-word block

```
For i = 0 to N/16-1 do
  For j = 0 to 15 do
    Set X[j] to M[i*16+j].
  End
```

MD5 is simple to implement, the only difficulty when its comes the same message digest for 2 messages on the order of 2^64 operations, but this will rarely happen.

Most of copyrights are with RSA data security for this hashing algorithm. License will be granted for derivative work with this algorithm from RSA data security.

SHA-

The United States of America has adopted Secure Hash Algorithms(SHAs). SHA-224, SHA-256, SHA-384 and SHA-512 are used for computing a condensed representation of message or data file. When a message size is less than 2^64 bits we use SHA-224 and SHA-256 but when a message size is less than 2^128 bits we use SHA-384 and SHA-512, the result of the output is called message digest. It is not computationally feasible to produce same message digest by the two different messages. Message padding is used to make the total length of the message multiple of 512 (for SHA-224, SHA-256) or a multiple of 1024(for SHA-384, SHA-512) (RFC 4634 US Secure Hash Algorithms (SHA and HMAC-SHA), 2006)

RC4 :

RC4 is a variable key-sized stream cipher developed by Ron Rivest in 1987. RC4 works in output-feedback (OFB) mode, so that the key stream is independent of the plaintext. The algorithm is described in detail in Schneier's *Applied Cryptography*, 2/e, pp. 397-398.

RC4 employs an 8x8 substitution box (S-box). The S-box is initialized so that S[i] = i, for i=(0,255).

A permutation of the S-box is then performed as a function of the key. The K array is a 256-byte structure that holds the key (possibly supplemented by an Initialization Vector), repeating itself as necessary so as to be 256 bytes in length (obviously, a longer key result in less repetition).

```
j = 0
for i = 0 to 255
  j = j + S[i] + K[i]
  swap (S[i], S[j])
```

Encryption and decryption are performed by XORing a byte of plaintext/ciphertext with a random byte from the S-box in order to produce the ciphertext/plaintext, as follows:

```
Initialize i and j to zero
```

For each byte of plaintext (or ciphertext):

i = i + 1
j = j + S[i]
swap (S[i], S[j])
z = S[i] + S[j]

Decryption: plaintext [i] = S[z] XOR ciphertext [i]
Encryption: ciphertext [i] = S[z] XOR plaintext [i]

2.2 Protocols

SSL:

SSL should be used to establish a secure connection between two devices. By utilizing SSL 3.0 to successfully exchange cryptographic parameters without knowledge of another's code.

Basic block size is 8 bits,

value = (byte[0] << 8*(n-1)) | (byte[1] << 8*(n-2)) | ...| byte[n-1];

The asymmetric algorithms are used in the handshake protocol to authenticate parties and to generate shared keys and secrets (A. Freier, 2011).

The Client list contains the following items:

- 1. Client hello
- 7. Certificate *optional*
- 8. Client key exchange
- 9. Certificate verify *optional*
- 10. Change cipher spec
- 11. Finished
- 14. Encrypted data
- 15. Close messages

Five empty lines separate item 1. Client hello and item 7. Certificate. Two empty lines separate item 11. Finished and item 14. Finished.

The Server list contains the following items:

- 2. Server hello
- 3. Certificate *optional*
- 4. Certificate request *optional*
- 5. Server key exchange *optional*
- 6. Server hello done
- 12. Change cipher spec
- 13. Finished
- 14. Encrypted data
- 15. Close messages

Five empty lines separate item 6. Server hello done and item 12. Change cipher spec.

2.3 Certification

Creating a Keystore to Use with JSSE

Create a new keystore and self-signed certificate with corresponding public and private keys.

keytool -genkeypair -alias nameofstore -keyalg RSA -validity 7 -keystore keystore

Examine the keystore. Notice that the entry type is keyEntry, which means that this entry has a private key associated with it).

keytool -list -v -keystore keystore

Enter keystore password: password

Export and examine the self-signed certificate

keytool -export -alias nameofstore -keystore keystore -rfc -file duke.cer

Enter keystore password: password

import the certificate into a new truststore.

keytool -import -alias nameofstorecert -file nameofstore.cer -keystore truststore

Enter keystore password: trustword

Before you can use the Java Signature class you must create a Signature instance. You create a Signatureinstance by calling the static getInstance() method. Here is an example that creates a Java Signatureinstance:

Signature signature = Signature.getInstance("SHA256WithDSA");

The String passed as parameter to the getInstance() method is the name of the digital signature algorithm to use.

SecureRandom secureRandom = new SecureRandom();

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("DSA");

KeyPair keyPair = keyPairGenerator.generateKeyPair();

signature.initSign(keyPair.getPrivate(), secureRandom);

When the Signature instance is initialized you can use it to create digital signatures. You create a digital signature by calling the update() method one or more times, finishing with a call to sign(). Here is an example of creating a digital signature for a block of binary data:

byte[] data = "abcdefghijklmnopqrstuvxyz".getBytes("UTF-8");

signature.update(data);

byte[] digitalSignature = signature.sign();

If you want to verify a digital signature created by someone else, you must initialize a Signature instance into verification mode (instead of signature mode). Here is how initializing a Signature instance into verification mode looks:

```
Signature signature = Signature.getInstance("SHA256WithDSA");
signature.initVerify(keyPair.getPublic());
```

Notice how the Java Signature instance is now initialized into verification mode, passing a public key of a public / private key pair as parameter.

Once initialized into verification mode you can use the Signature instance to verify a digital signature. Here is how verifying a digital signature looks:

```
byte[] data2 = "abcdefghijklmnopqrstuvxyz".getBytes("UTF-8");
signature2.update(data2); (Corporation, 2016)

boolean verified = signature2.verify(digitalSignature);
```

In some cases, cost associated with changing digital certificates and cryptographic keys are high. Examples include decryption and subsequent re-encryption of very high databases and legacy systems, decryption and re-encryption of distributed systems has very huge number of keys. In such a case, the expense of security measures necessary to support longer crypto periods may justified. In some situations, we must revoke certification when employee or resource moving from certain trusted job.

Chapter 3

Research Methodology

3.1 Java Cryptographic Architecture

There are many programming languages in which the security algorithms implemented for the application security. The most common platform for today's application development is Java. The Java platform strongly emphasizes security with core libraries inbuilt, including language safety, cryptography, public key infrastructure, authentication, secure communication. The JCA (Java Cryptographic Architecture) is a major piece of the platform and contains a "provider" architecture and a set of APIs for digital signatures, message digests (hashes), certificates and certificate validation, encryption (symmetric/asymmetric block/stream ciphers), key generation and management, and secure random number generation.

These APIs allow developers to easily integrate security into their application code. The architecture was designed around implementation independence, implementation interoperability and algorithm extensibility.

As java interoperable across applications, not bound to specific provider and provider is not bound to specific application. Java platform has many built-in sets of security which used widely today.

Java included packages javax.crypto, javax.crypto.spec, javax.crypto.interfaces

Java.security.provider is the base class for security providers. When instance of particular algorithm needed, the JCS consult the providers database.

Suppose you want to get the statement request for SHA-512 from installed provider

md = MessageDigest.getInstance("SHA-512");

to get an instance of an engine class to implement AES

import javax.crypto.*;

Cipher ctxt = Cipher.getInstance("AES");

ctxt.init(ENCRYPT_MODE, key);

ctxt init(DECRYPT_MODE, key); A secure class loading, and verification mechanism ensures that only legitimate java code will get executed. It's better to use java cryptographic architecture mechanism while executing the application transactions on application and web servers.

Method to put security algorithms into programming language.

AES (Advanced Encryption Standard) can encrypt and decrypt information using a common password of 128, 192 or 256 bits. In this program I am using a random 128 bits password in this case (16 bytes). This is useful for protect classified information you need to bring back the original form later. AES is a symmetric block cipher used by the U.S. government adopted. MD5: Useful for databases, the text encrypted by MD5 theoretically can't be decrypted then only the encrypted info would match the same result. Better to store passwords in databases, can be hacked only by force breaker and it can be impossible as well and it cannot be decrypted

Sample Java Code

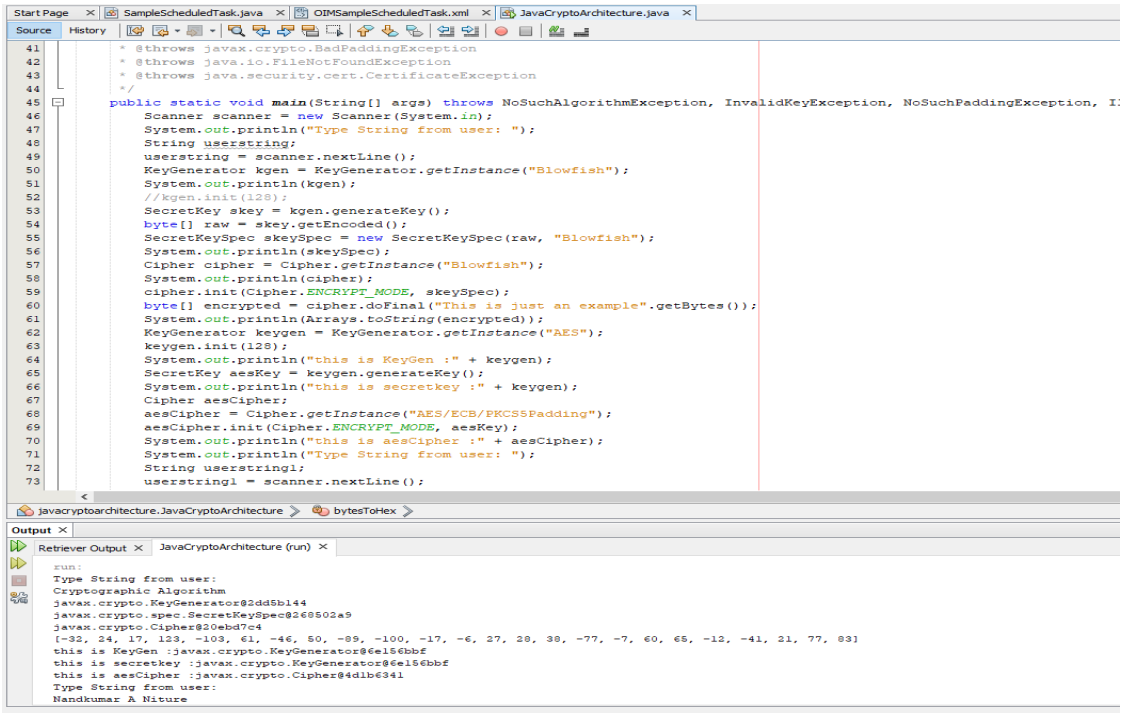
```
package javacryptoarchitecture;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.security.*;
import java.util.Arrays;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.util.Scanner;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.spec.*;
import java.security.MessageDigest;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.util.Collection;
import java.util.Iterator;
import java.util.logging.Logger;
import javax.xml.bind.DatatypeConverter;

/**
 *
 * @author Nandkumar
 */
public class JavaCryptoArchitecture {

   /**
    * @param args the command line arguments
    * @throws java.security.NoSuchAlgorithmException
    * @throws java.security.InvalidKeyException
    * @throws javax.crypto.NoSuchPaddingException
    * @throws javax.crypto.IllegalBlockSizeException
    * @throws javax.crypto.BadPaddingException
    * @throws java.io.FileNotFoundException
    * @throws java.security.cert.CertificateException
    */
   public static void main(String[] args) throws NoSuchAlgorithmException, InvalidKeyException, NoSuchPaddingException,
IllegalBlockSizeException, BadPaddingException, FileNotFoundException, CertificateException, IOException {
       Scanner scanner = new Scanner(System.in);
       System.out.println("Type String from user: ");
```

```
    String userstring;
    userstring = scanner.nextLine();
    KeyGenerator kgen = KeyGenerator.getInstance("Blowfish");
    System.out.println(kgen);
    //kgen.init(128);
    SecretKey skey = kgen.generateKey();
    byte[] raw = skey.getEncoded();
    SecretKeySpec skeySpec = new SecretKeySpec(raw, "Blowfish");
    System.out.println(skeySpec);
    Cipher cipher = Cipher.getInstance("Blowfish");
    System.out.println(cipher);
    cipher.init(Cipher.ENCRYPT_MODE, skeySpec);
    byte[] encrypted = cipher.doFinal("This is just an example".getBytes());
    System.out.println(Arrays.toString(encrypted));
    KeyGenerator keygen = KeyGenerator.getInstance("AES");
    keygen.init(128);
    System.out.println("this is KeyGen :" + keygen);
    SecretKey aesKey = keygen.generateKey();
    System.out.println("this is secretkey :" + keygen);
    Cipher aesCipher;
    aesCipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
    aesCipher.init(Cipher.ENCRYPT_MODE, aesKey);
    System.out.println("this is aesCipher :" + aesCipher);
    System.out.println("Type String from user: ");
    String userstring1;
    userstring1 = scanner.nextLine();
    byte[] cleartext;
    cleartext = userstring1.getBytes();
    System.out.println("this is cleartext :" + Arrays.toString(cleartext));
    byte[] ciphertext;
    ciphertext = aesCipher.doFinal(cleartext);
    System.out.println("this is ciphertext :" + Arrays.toString(ciphertext));
    aesCipher.init(Cipher.DECRYPT_MODE, aesKey);
    byte[] cleartext1;
    cleartext1 = aesCipher.doFinal(ciphertext);
    System.out.println("this is cleartext1 :" + Arrays.toString(cleartext1));

    System.out.println("Type String from user for SHA256 : ");
    Scanner sn = new Scanner(System.in);

    System.out.print("Please enter data for which SHA256 is required:");

    String data = sn.nextLine();
    JavaCryptoArchitecture jc = new JavaCryptoArchitecture();
    String hash = jc.getSHA256Hash(data);
    System.out.println("The SHA256 (hexadecimal encoded) hash is:" + hash);
    MessageDigest md = MessageDigest.getInstance("MD5");
    int MDciphertext = md.getDigestLength();
    System.out.println("This is MD5 Digest Length" + MDciphertext);
    Provider MDCipher = md.getProvider();
    System.out.println("This is MD5 Digest Provider" + MDCipher);
    String name = "Nandkumar";
    byte[] bytesOfMessage = name.getBytes("UTF-8");
    byte[] MDC1 = md.digest(bytesOfMessage);
    System.out.println(" This is MD5 cipher" + Arrays.toString(MDC1));
  }

  private String getSHA256Hash(String data) {
String result = null;

    try {

      MessageDigest digest = MessageDigest.getInstance("SHA-256");

      byte[] hash = digest.digest(data.getBytes("UTF-8"));
```

```
        return bytesToHex(hash);

    } catch (UnsupportedEncodingException | NoSuchAlgorithmException ex) {
    }

    return result;
  }

  private String bytesToHex(byte[] hash) {

    return DatatypeConverter.printHexBinary(hash);

  }
```

Output of the program:

```
Type String from user:
Cryptographic Algorithm
javax.crypto.KeyGenerator@2dd5b144
javax.crypto.spec.SecretKeySpec@268502a9
javax.crypto.Cipher@20ebd7c4
[-32, 24, 17, 123, -103, 61, -46, 50, -89, -100, -17, -6, 27, 28, 38, -77, -7, 60, 65, -12, -41, 21, 77, 83]
this is KeyGen :javax.crypto.KeyGenerator@6e156bbf
this is secretkey :javax.crypto.KeyGenerator@6e156bbf
this is aesCipher :javax.crypto.Cipher@4d1b6341
Type String from user:
Nandkumar A Niture
this is cleartext :[78, 97, 110, 100, 107, 117, 109, 97, 114, 32, 65, 32, 78, 105, 116, 117, 114, 101]
this is ciphertext :[21, -49, -92, -88, -47, 60, 109, 113, 126, -58, 73, 35, 74, 93, 106, -95, 103, -124, -99, -50, 91, -56, 99, 1, -126, 17, -54, 26, 113, -125, -93, 98]
this is cleartext1 :[78, 97, 110, 100, 107, 117, 109, 97, 114, 32, 65, 32, 78, 105, 116, 117, 114, 101]
Type String from user for SHA256 :
Please enter data for which SHA256 is required:Nandkumar Niture
The SHA256 (hexadecimal encoded) hash is:997CCD6D8B600D62A5DF98F99E7E7C9A7AA677386DAD4A0F81242D7F86994380
This is MD5 Digest Length16
This is MD5 Digest ProviderSUN version 1.7
This is MD5 cipher[53, 16, 83, 107, 60, 77, 30, 58, -23, -122, -65, -66, -25, 21, -94, 100]
```




```
public static byte[] encrypt(final SecretKeySpec key, final byte[] iv, final byte[] message) throws GeneralSecurityException {
    final Cipher cipher = Cipher.getInstance(AES_MODE);
    IvParameterSpec ivSpec = new IvParameterSpec(iv);
    cipher.init(Cipher.ENCRYPT_MODE, key, ivSpec);
    byte[] cipherText = cipher.doFinal(message);

    log("cipherText", cipherText);

    return cipherText;
  }
public static String decrypt(final String password, String base64EncodedCipherText) throws GeneralSecurityException {

    try {
      final SecretKeySpec key = generateKey(password);

      log("base64EncodedCipherText", base64EncodedCipherText);
      byte[] decodedCipherText = Base64.decode(base64EncodedCipherText, Base64.NO_WRAP);
      log("decodedCipherText", decodedCipherText);

      byte[] decryptedBytes = decrypt(key, ivBytes, decodedCipherText);

      log("decryptedBytes", decryptedBytes);
      String message = new String(decryptedBytes, CHARSET);
      log("message", message);


      return message;
    }
```

3.2 Time and Space Complexity

Complexity theory provides a methodology for analyzing the computational complexity of an algorithm. This can be measured by two variables Time(T) and Space(S)

Both T and S are expressed as function of n and where n is a size of an input. This computational complexity measured in big O notation.

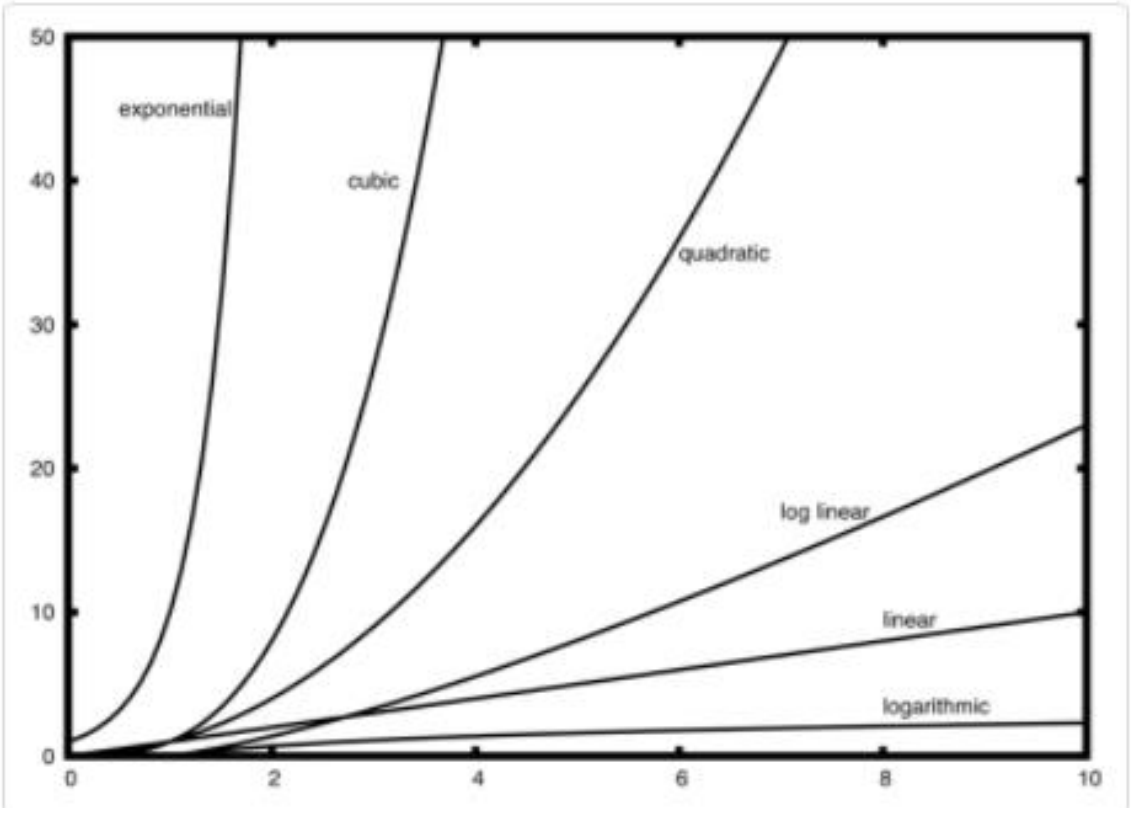
This is measured in terms of polynomial, lets say the quadratic equation polynomial is having complexity as

T = $O(n^2)$

Polynomial algorithms with index m is having time or space complexity as O(n raise to m)

This class of algorithm having time complexity is called polynomial-time algorithms.

In cryptography ideally a algorithm considered as best algorithm to break is algorithm of exponential-time complexity. The super polynomial time complexity algorithms is ideal for the cryptographer and security engineers to design.



3.2.1Polynomial equation Time Complexity

quadratic	N^2	`for (int i = 0; i < N; i++)`
		`  for (int j = i+1; j < N; j++)`
		`  if (a[i] + a[j] == 0)`
		`  cnt++;`
			double loop	*check all pairs*

cubic	N^3	`for (int i = 0; i < N; i++)`
		`  for (int j = i+1; j < N; j++)`
		`  for (int k = j+1; k < N; k++)`
		`  if (a[i] + a[j] + a[k] == 0)`
		`  cnt++;`
			triple loop	*check all triples*

Time evaluation for 3DES is cubic ---

for (k3 = 0 to 2^56 step 1)

compute C2 = D_k3(C1)

for (k2 = 0 to 2^56 step 1)

compute C3 = E_k2(C2)

for (k1 = 0 to 2^56 step 1)

begin

compute p = D_k1(C3) xor IV

if (p equals p-expected)

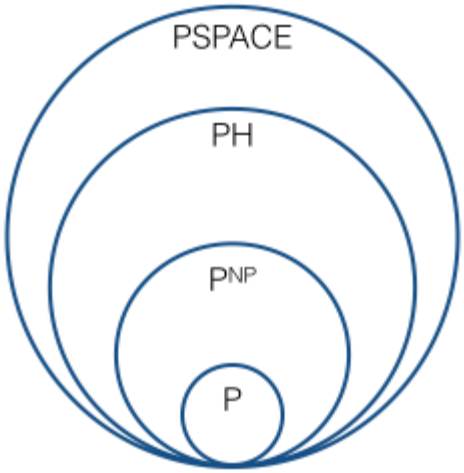
exit loop; we found the keys

end

3DES has 3 loops, so the time complexity would go high and its almost impossible to break the security algorithm.

P = Problem solved

NP = Problem Not Solved



3.2.2 NP and P diagram

When P=NP the algorithms are breakable by feasible and deterministic.

In security we always must and make sure P should never be equal to NP

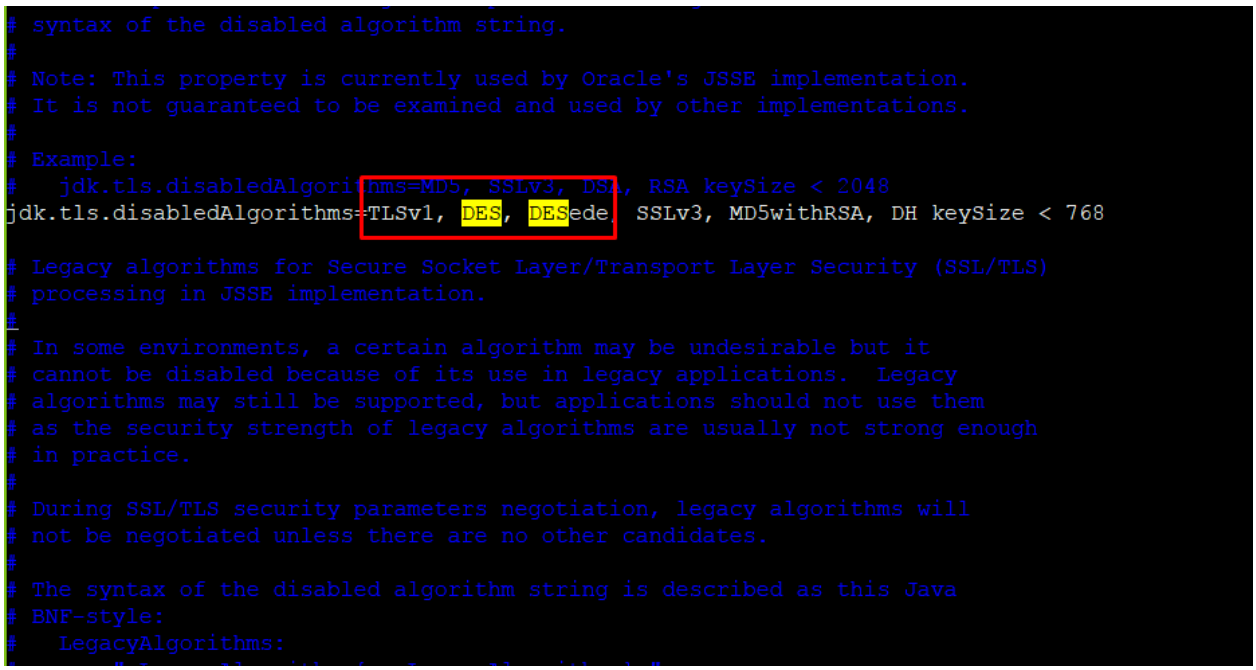
3.3 Java Application Security

To enable and disable the cryptographic algorithms in java application security –

Edit java.security file under jre/lib/security/java.security

If you want to enable or disable the algorithm

add values into jdk.tls.disabledAlgorithms= TLSv1, DES, DESede



Test to make sure TLSv1 is not enabled

openssl s_client -connect appserver-dev:7002 -tls1_1

This is test to make sure tls1_1 is enabled -

openssl s_client -connect appserver-dev:7002 -tls1_1

This is test to make sure DES and 3DES not enabled -

openssl s_client -connect oig01-dev.ual.com:7002 -cipher "DES:3DES"

Creating an SSLEngine object – From Java Oracle Reference

import javax.net.ssl.*;

import java.security.*;

KeyStore ksKeys = KeyStore.getInstance("JKS");

```
ksKeys.load(new FileInputStream("testKeys"), passphrase);
KeyStore ksTrust = KeyStore.getInstance("JKS");
ksTrust.load(new FileInputStream("testTrust"), passphrase);
KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
kmf.init(ksKeys, passphrase);
// TrustManagers decide whether to allow connections
TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
tmf.init(ksTrust);
sslContext = SSLContext.getInstance("TLS");
SSLEngine engine = sslContext.createSSLengine(hostname, port);
engine.setUseClientMode(true);
```

How to grant the permission on algorithms in Java application security.

Local_Policy.jar setting

```
// Some countries have import limits on crypto strength. This policy file
// is worldwide importable.

grant {
    permission javax.crypto.CryptoPermission "DES", 64;
    permission javax.crypto.CryptoPermission "DESede", *;
    permission javax.crypto.CryptoPermission "RC2", 128,
                                    "javax.crypto.spec.RC2ParameterSpec", 128;
    permission javax.crypto.CryptoPermission "RC4", 128;
    permission javax.crypto.CryptoPermission "RC5", 128,
          "javax.crypto.spec.RC5ParameterSpec", *, 12, *;
    permission javax.crypto.CryptoPermission "RSA", *;
    permission javax.crypto.CryptoPermission *, 128;
    };
```

With no restrictions –

```
// Country-specific policy file for countries with no limits on crypto strength.
grant {
    // There is no restriction to any algorithms.
    permission javax.crypto.CryptoAllPermission;
    };
```

Below program implement all the secure hash functions in one program

```
public class HashFun {
```

```
	// algorithms: MD2, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512(most Secure)

	public static void main(String[] args) {
		String anyString = "Hash this String";
		System.out.println(getHash(anyString.getBytes(), "SHA-512"));

		try {
			File image = new File("fam.jpg");
			byte[] imageBytes = Files.readAllBytes(image.toPath());
			System.out.println(getHash(imageBytes, "SHA-512"));
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
	public static String getHash(byte[] inputBytes, String algo) {
		String hashVal = "";
		try {
			MessageDigest messageDigest = MessageDigest.getInstance(algo);
			messageDigest.update(inputBytes);
			byte[] digestedBytes = messageDigest.digest();
			hashVal = DatatypeConverter.printHexBinary(digestedBytes).toLowerCase();
		} catch (Exception e) {
			e.printStackTrace();
		}
		return hashVal;
	}
```

Chapter 4

Analysis and Discussion

4.1 Analysis

If performance is not an issue, then there is no reason not to use multiple streams ciphers and cascade them. The output of each generator can be again combined into another input output will get the resultset of more complicated encrypted data. Now this standard can be applied in to the data protection and make application more secured. The security of the cascade is at least as secure as the strongest algorithm. Stream ciphers can be combined in all the same ways as block ciphers. Stream ciphers can be combined in all the same ways block ciphers. Stream ciphers can be cascaded with other stream ciphers, or together with block ciphers.

The most clever trick is to use the multiple algorithms either a block of stream or block cipher.

The ideal hash function has three main properties:

1. It is extremely easy to calculate a hash for any given data.
2. It is extremely computationally difficult to calculate an alphanumeric text that has a given hash.
3. It is extremely unlikely that two slightly different messages will have the same hash.

The MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA (Network Working Group, 1992).

In Java Message-Digest class provides the functionality of a message digest algorithm, such as MD5 or SHA. Message digests are secure one-way hash functions that take arbitrary-sized data and output a fixed-length hash value.

Like other algorithm-based classes in Java Security, Message-Digest has two major components:

Message Digest API (Application Program Interface)
: This is the interface of methods called by applications needing message digest services. The API consists of all public methods.

Message Digest SPI (Service Provider Interface)
: This is the interface implemented by providers that supply specific algorithms. It consists of all methods whose names are prefixed by *engine*. Each such method is called by a

correspondingly-named public API method. For example, the `engineReset` method is called by the `reset` method. The SPI methods are abstract; providers must supply a concrete implementation.

A MessageDigest object starts out initialized. The data is processed through it using the update methods. At any point reset can be called to reset the digest. Once all the data to be updated has been updated, one of the digest methods should be called to complete the hash computation.

The `digest` method can be called once for a given number of updates. After `digest` has been called, the MessageDigest object is reset to its initialized state.

Implementations are free to implement the Cloneable interface, and doing so will let client applications test cloneability using `instanceof Cloneable` before cloning:

```
 MessageDigest md = MessageDigest.getInstance("SHA");
 if (md instanceof Cloneable) {
     md.update(toChapter1);
     MessageDigest tc1 = md.clone();
     byte[] toChapter1Digest = tc1.digest;
     md.update(toChapter2);
     ...etc.
 } else {
     throw new DigestException("couldn't make digest of partial
content");
 }
```

The MD5 message-digest algorithm is simple to implement and provides a "fingerprint" or message digest of a message of arbitrary length. It is conjectured that the difficulty of coming up with two messages having the same message digest is on the order of 2^64 operations, and that the difficulty of coming up with any message having a given message digest is on the order of 2^128 operations. The MD5 algorithm has been scrutinized for weaknesses. It is, however, a relatively new algorithm and further security analysis is of course justified, as is the case with any new proposal of this sort (Network Working Group, 1992).

Hashing is a one-way function. It's "irreversible". Unlike the other private or public key which requires same key on client and server. Hashing is mostly used for integrity purposes with the output hash given to anyone who request the hash function. Example: You download a Oracle IOS image, you then download the hash, the hash is in a base 16 format. The hash you download will either MD5 or a version of SHA, this will be given by the

source of the IOS download. You can run the IOS image file through the same hash mechanism (MD5 or SHA) using free software and you should get the same exact output that Cisco did. This is the beauty of hashing; 1000 people can hash the same file 1000 times and the exact same output should be created. With the hash being the same you have just proven integrity of the data. If the hash is different than the data has been tampered with.

Encryption is a complex subject. Encryption takes plain text data and converts it to unreadable data. There are multiple algorithms (symmetric & asymmetric) that can be used to encrypt, it depends on the reason of encryption. Symmetric encryption algorithms (AES, DES, 3DES) use the same key (public key) to encrypt and decrypt data. Asymmetric encryption uses both private and public key to encrypt and decrypt data (commonly used for digital signatures).

A digital signature is when the sender of data hashes text, encrypts the text with his "Private Key," then transmits the data (plain text & hash). The receiver has access to the senders "Public Key" (receiver being validated and validating through a trusted CA) who then decrypts the data, hashing the same text that the sender hashed which should come to the same output. When this happens the receiver just verified the sender's identity. Private and public keys come in pairs and can only work within themselves, so if the obtained public key decrypts data that was encrypted by the sender's private key (sender is the only person who should have it on the planet), non-repudiation and verification of identity was just performed through that digital signature.

4.2 Discussion

Why hash functions are so important in today's cryptographic standards, because hash functions play a fundamental role in modern cryptography in many ways. While related to conventional hash functions commonly used in non-cryptographic computer applications – in both cases, larger domains are mapped to smaller ranges – they differ in several important aspects. Our focus is restricted to cryptographic hash functions (hereafter, simply hash functions), and to their use for data integrity and message authentication. Hash functions take a message as input and produce an output referred to as a hash code, hash-result, hash-value, or simply hash. (Lecture Outline UT Dallas, 2015)

High level View and Detailed view of hash function. Most unkeyed hash functions h are designed as iterative processes and repeated the output processed data which hash arbitrary length inputs by processing successive fixed-size blocks of the input, as illustrated in Figure (Menezes, 1996)

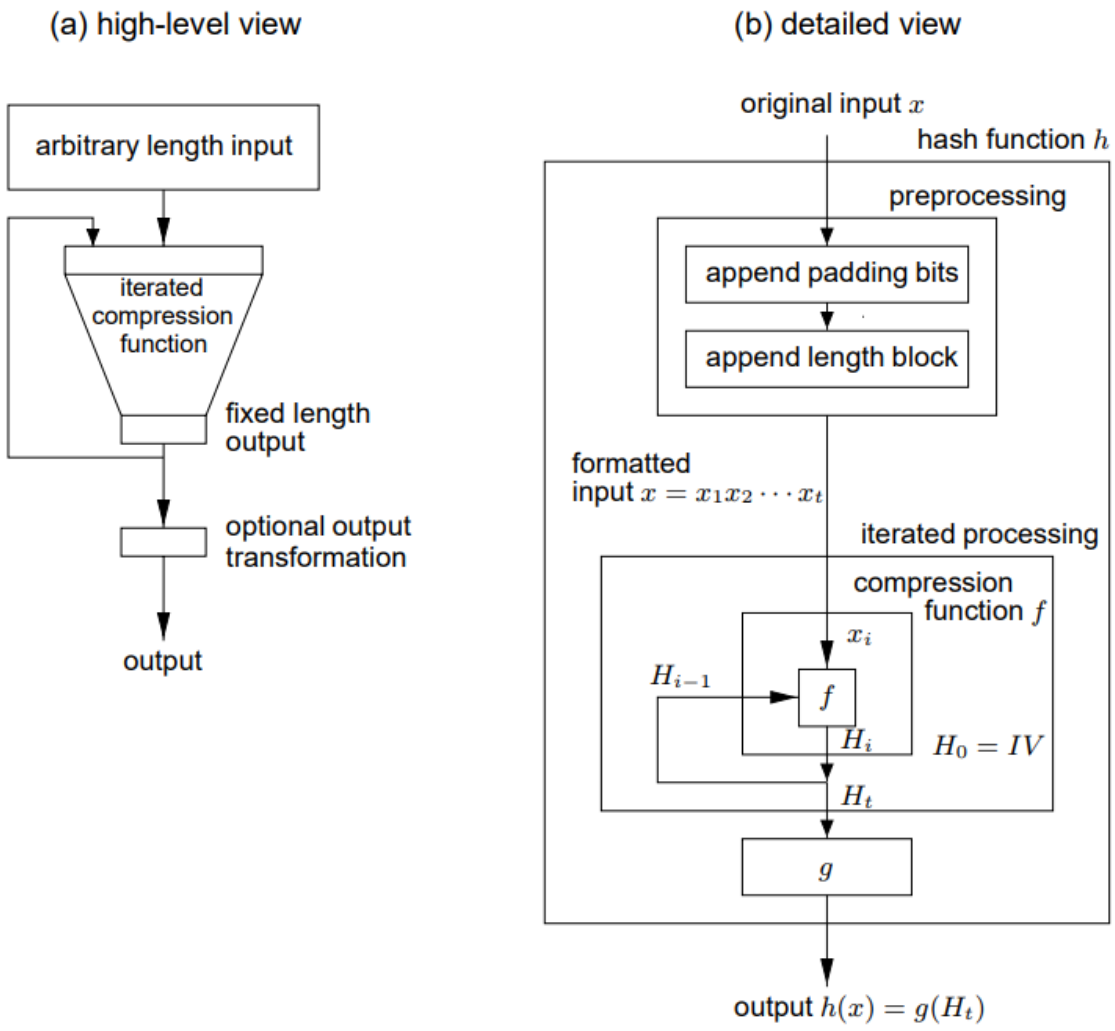


Figure 4.2.1: Hash Function – High Level View and Detailed View (A. Freier, 2011)

A hash input x of arbitrary finite length is divided into fixed-length r-bit blocks xi. This preprocessing typically involves appending extra bits (padding) as necessary to attain an overall bitlength which is a multiple of the block length r, and often include a block or partial block indicating the bitlength of the unpadded input. Each block xi then serves as input to an internal fixed-size hash function f, the compression function of h, which computes a new intermediate result of bitlength n for some fixed n, as a function of the previous n-bit intermediate result and the next input block xi. Letting Hi denote the partial result after stage i, the general process for an iterated hash function with input x = x1x2 ...xt can be modeled as follows: H0 = IV ; Hi = f(Hi−1, xi), 1 ≤ i ≤ t; h(x) = g(Ht). Hi−1 serves as the n-bit chaining variable between stage i − 1 and stage i, and H0 is a pre-defined starting value or initializing value (IV). An optional output transformation g is used in a final step to map the n-bit

chaining variable to an m-bit result g(Ht); g is often the identity mapping g(Ht) = Ht. Hash functions are distinguished by the nature of the preprocessing, compression function, and output transformation. (Menezes, 1996)

Chapter 5

Conclusion and Recommendations

5.1 Result of Analysis

Computer security is based on the use of cryptography. Robust cryptography is based on two things: Good algorithms (e.g., AES) and High-quality keys (e.g., good random numbers).

There is always a right way of doing things, and which involves a cost, time and quality work from security engineers while building security around applications.

There are many ways for double to power of encryption, the method of cascading will construct the double encryption in programmatic way to encrypt the data. As I have elaborated the programming methods and structures of the deriving the encryption standards there are multiple ways of encrypting the data.

5.2 Recommendations

Data breached can be avoided with the strong security algorithms in application security. One of the most efficient way to implement security is at the time of application development and making sure the quality of product rating based on its security in the application domain.

While doing customization at client side there need to be certain principles followed from the security engineers and product development team. Security is not product or feature to be released in the market, but it is the best practices to be implemented across each domain of application development.

5.3 Open Questions

In this dissertation, I analyzed public-key and identity-based encryption schemes that are secure against memory attacks. The first question that arises from our work is whether it is possible to (define and) construct other cryptographic primitives such as signature schemes, identification schemes and even protocol tasks that are secure against memory attacks. The second question is whether it is possible to protect against memory attacks that measure an arbitrary polynomial number of bits. Clearly, this requires some form of (randomized) refreshing of the secret-key, and it would be interesting to construct such a mechanism. Finally, it would be interesting to improve the parameters of our construction, as well as the complexity assumptions, and to design encryption schemes against memory attacks under other cryptographic assumptions.

Electronics surveillance has become a powerful tool in the law enforcement and cybersecurity law implementation. However, there is an increasing realization that new technology may be helping hackers and criminals more than low and ethical enforcements. There are many open-ended discussions on the privacy and security of sensitive data. Data encryption and using highly secured algorithms is one of the way but not the only way!

Appendices

1. *Privacy/confidentiality*: Ensuring that no one can read the message except the intended receiver.
2. *Authentication*: The process of proving one's identity.
3. *Integrity*: Assuring the receiver that the received message has not been altered in any way from the original.
4. *Non-repudiation*: A mechanism to prove that the sender really sent this message.
5. *Key exchange*: The method by which crypto keys are shared between sender and receiver.
6. Public key: Public keys are those keys that are made available to anyone who needs it and is used to encrypt the data.
7. *Private key*: A Private key is safe and is not available to anyone except the creator and is used to decrypt data encrypted by the public key.
8. *Symmetric key cryptography*: In symmetric key cryptography the same key is used for both encryption and decryption.
9. *Asymmetric key cryptography*: In asymmetric key cryptography a different key is used for encryption and decryption.
10. *Encryption algorithm*: technique or rules selected for encryption.
11. *Key*: is secret value used to encrypt and/or decrypt the plain-text
12. *Cryptanalysis*: The study of cryptographic algorithms
13. *Cryptology:* Cryptography and cryptanalysis combined constitute the area of cryptology

Java Security Engine Classes

The following engine classes are available in Java Cryptographic Architecture (Corporation, 2016)

1. *SecureRandom*: used to generate random or pseudo-random numbers.
2. *MessageDigest*: used to calculate the message digest (hash) of specified data.
3. *Signature*: initialized with keys, these are used to sign data and verify digital signatures.
4. Cipher: initialized with keys, these are used for encrypting/decrypting data. There are various types of algorithms: symmetric bulk encryption (e.g. AES), asymmetric encryption (e.g. RSA), and password-based encryption (e.g. PBE).

5. *Message Authentication Codes* (MAC): like MessageDigests, these also generate hash values, but are first initialized with keys to protect the integrity of messages.
6. *KeyFactory*: used to convert existing opaque cryptographic keys of type Key into key specifications (transparent representations of the underlying key material), and vice versa.
7. *SecretKeyFactory*: used to convert existing opaque cryptographic keys of type SecretKey into key specifications (transparent representations of the underlying key material), and vice versa. SecretKeyFactorys are specialized KeyFactorys that create secret (symmetric) keys only.
8. *KeyPairGenerator*: used to generate a new pair of public and private keys suitable for use with a specified algorithm.
9. *KeyGenerator*: used to generate new secret keys for use with a specified algorithm.
10. *KeyAgreement*: used by two or more parties to agree upon and establish a specific key to use for a particular cryptographic operation.
11. *AlgorithmParameters*: used to store the parameters for a particular algorithm, including parameter encoding and decoding.
12. *AlgorithmParameterGenerator* : used to generate a set of AlgorithmParameters suitable for a specified algorithm.
13. *KeyStore*: used to create and manage a keystore. A keystore is a database of keys. Private keys in a keystore have a certificate chain associated with them, which authenticates the corresponding public key. A keystore also contains certificates from trusted entities.
14. *CertificateFactory*: used to create public key certificates and Certificate Revocation Lists (CRLs).
15. *CertPathBuilder*: used to build certificate chains (also known as certification paths).
16. *CertPathValidator*: used to validate certificate chains.
17. *CertStore*: used to retrieve Certificates and CRLs from a repository.

Key Size with possible combinations -

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}

Symmetric Encryption Algorithm	Key Length (in bits)
DES	56
3DES	112 and 168
AES	128, 192, and 256
SEAL	160
RC	RC2 (40 and 64)
	RC4 (1 to 256)
	RC5 (0 to 2040)
	RC6 (128, 192, and 256)

Elliptic-Curve Digital Signature Algorithm (ECDSA)

NIST Guidelines for Public Key Sizes for AES			
ECC key size (bits)	**RSA key size (bits)**	**Key size ratio**	**AES key size (bits)**
163	1,024	1:6	
256	3,072	1:12	128
384	7,680	1:20	192
512	15,360	1:30	256

Elliptic curve is attractive for postage systems because the printed signatures are very compact.

ECC-based Signature

RSA-based Signature

Asymmetric Encryption Algorithm	Key Length (in bits)
DH	512, 1024, 2048, 3072, 4096
DSS and DSA	512–1024
RSA	512–2048
ElGamal	512–1024
Elliptical curve techniques	160

References

A. Freier, P. K. (2011, August). *The Secure Sockets Layer (SSL) Protocol Version 3.0*. Retrieved from www.tools.ietf.org: https://tools.ietf.org/html/rfc6101

Australia., D. M. (2000-14). *DI Management*. Retrieved from DI Management Cryptography: https://www.di-mgt.com.au/cryptokeys.html

Corporation, O. (2016, June). *Engine Classes and Algorithms*. Retrieved from www.oracle.com: https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html

Hellman, M. a. (1977). Exhaustive Cryptanalysis of NBS Data Encryption Standard. *https://ee.stanford.edu/~hellman/publications/27.pdf*, 1-2.

Implementation of Diffie-Hellman Algorithm. (2017). Retrieved from https://www.geeksforgeeks.org: https://www.geeksforgeeks.org/implementation-diffie-hellman-algorithm/

Intel® AES-NI Performance Testing on Linux/Java* Stack*. (2012, June 1). Retrieved from software.intel.com: https://software.intel.com/en-us/articles/intel-aes-ni-performance-testing-on-linuxjava-stack

Kelly, S. (2006, December). *Network Working Group RFC*. Retrieved from www.ietf.org: https://www.ietf.org/rfc/rfc4772.txt

L. Hornquist Astrand, A. I. (2012, July). *Request for Comments: 6649* . Retrieved from www.ietf.org : https://tools.ietf.org/html/rfc6649

Lecture Outline UT Dallas. (2015). Retrieved from http://www.utdallas.edu/~muratk/courses/crypto07_files/hash.pdf: http://www.utdallas.edu/~muratk/courses/crypto07_files/hash.pdf

Mehmood, A. (2017, oct 9). *Differences between Hash functions, Symmetric & Asymmetric Algorithms*. Retrieved from www.cryptomathic.com: https://www.cryptomathic.com/news-events/blog/differences-between-hash-functions-symmetric-asymmetric-algorithms

Menezes. (1996). *Handbook of Applied Cryptography*. Retrieved from Math Waterloo.ca: http://cacr.uwaterloo.ca/hac/about/chap9.pdf

Network Working Group, R. f. (1992, April). *The MD5 Message-Digest Algorithm*. Retrieved from IETF : https://tools.ietf.org/html/rfc1321

RELEASE, P. (2016, Aug 9). *EFF DES CRACKER MACHINE BRINGS HONESTY TO CRYPTO DEBATE*. Retrieved from www.eff.org: https://www.eff.org/press/releases/eff-des-cracker-machine-brings-honesty-crypto-debate

RFC 4634 US Secure Hash Algorithms (SHA and HMAC-SHA). (2006, July). Retrieved from https://tools.ietf.org: https://tools.ietf.org/html/rfc4634

Schaad, J. (2002, September). *Advanced Encryption Standard (AES) Key Wrap Algorithm*. Retrieved from https://tools.ietf.org/html/rfc3394: https://tools.ietf.org/html/rfc3394

Schneier, B. (1996). *Applied Cryptography*. Hoboken, NJ: John Wiley & Sons Inc.