http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 4 (3): 204-221, 2005
ISSN 1812-5638

Secure Implementation of Message Digest, Authentication and Digital Signature

Kefa Rabah
Department of Physics, Eastern Mediterranean University,
Gazimagusa, North Cyprus, via Mersin 10, Turkey

Abstract: In this study we present the techniques that are useful in securing the data against tamper in between communicating parties by the man in the middle. This involves the use and implementation of message digest (or hashing), message authentication and digital signature schemes. A hash function can provide message authentication in a most satisfying manner when combined with digital signature algorithm, which does have a key. Digital signatures currently provide Internet applications with data authentication and non-repudiation services and is set to continue playing an important role in future as Internet services continues to grow. Typical digital signature schemes, however, have some performance overhead, which, while acceptable for the periodic setup of communication sessions, is often too large on a message-by-message basis. Thus, the need today is to focus on the message authentication based on shared secrete-key, which is ideally integrated into the hash function in some manner and that's the subject of this study.

Key words: Digital signature, internet security, message digest, authentication, nonrepudiation, online content management, Java, JCE, JCA, computer and wireless security

INTRODUCTION

Hash algorithms are one-way mathematical algorithms that take an arbitrary length input and produce a fixed length output string[1]. The hash value is a unique and extremely compact numerical representation of a piece of data. Some of the currently approved hash functions are: SHA1, MD5, RIPEM-128 and RIPEM-160 etc. MD5 produces 128-bits while SHA-1 produces 160-bits[2], respectively. It is computationally improbable to find two distinct inputs that hash to the same value (or collide). Hash functions have some very useful applications. They allow a party to prove they know something without revealing what it is and hence are seeing widespread use in password schemes. They can also be used in digital signatures and integrity protection.

A message digest is a compact digital signature for an arbitrarily long stream of binary data. An ideal message digest algorithm would never generate the same signature for two different sets of input, but achieving such theoretical perfection would require a message digest as long as the input file. Practical message digest algorithms compromise in favor of a digital signature of modest size created with an algorithm designed to make preparation of input text with a given signature computationally infeasible. Message digest algorithms have much in common with techniques used in encryption, but to a different end; verification that data have not been altered since the signature was published.

Many older programs requiring digital signatures employ 16 or 32-bit cyclical redundancy codes (CRC) originally developed to verify correct transmission in data communication protocols, but these short codes, while adequate to detect the kind of transmission errors for which they were intended, are insufficiently secure for applications such as electronic commerce and verification of security related software distributions[3].

The most commonly used present-day message digest algorithm is the 128-bit MD5 algorithm, developed by Ron Rivest of the MIT Laboratory for Computer Science[4]. Message digest algorithms such as MD5 are not deemed encryption technology and are not subject to the export controls some governments impose on other data security products. The MD5 algorithm was originally developed as part of a suite of tools intended to monitor large collections of files (for example, the contents of a Web site) to detect corruption of files and inadvertent (or perhaps malicious) changes.

MESSAGE AUTHENTICATION WITH MD5

Message authentication algorithm is currently playing an important role in a variety of applications, especially those related to the Internet Protocols (IP) and network management, where undetected manipulation of messages can have disastrous effects. There is no shortage of good message authentication codes, beginning with DES-MAC, as defined in FIPS PUB 113[5]. However, message authentication codes based on

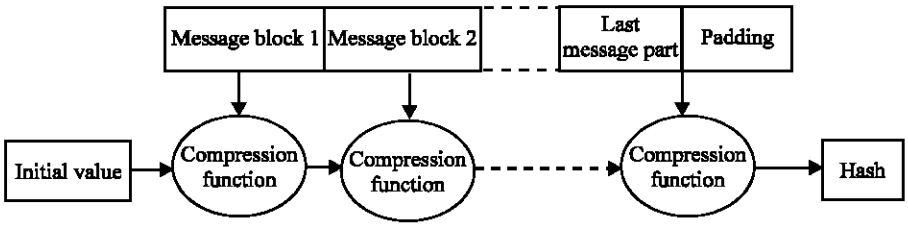

Fig. 1: A schematic of Damgård/Mekle iterative structure for hash functions

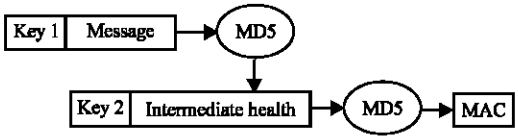

Fig. 2: Message authentication (MAC) with MD5

encryption function such as DES, which were originally designed for hardware implementation, may be somewhat limited in performance for software and there is also the question of US export restriction on high quality encryption functions. In standards applications such as the Simple Network Management Protocol (SNMP)[6] and proposals for Internet Protocol (IP) security, a more practical solution seemed to be to base the authentication codes not on data security standard (DES)[7] but on hash functions designed for fast software implementation which are widely available without restriction, such as MD5 message-digest algorithm, SHA-1 etc[7].

But how to do it? Hash functions are intended to resist inversion-finding a message with a given hash value - and collision - finding two messages with the same hash value. Message Authentication Codes (MAC), on the other hand, are intended to resist forgery - i.e., computing a message authentication code without knowledge of a secrete-key. Building a message authentication code on an encryption function thus seems a logical choice (and the security relationship has been recently settled-in the work by Bellare *et al.*[8]). Building one on a hash function, however, is not as simple, because the hash function doesn't have a key.

As an illustration of the challenges, consider the prefix approach where the message authentication code is computed simply as the hash of the concatenation of the key and the message, where they key comes first and which we denote as MD5(k·m). MD5 follows the Damgård/Merkle[9,10] iterative structure; where the hash is computed by repeated application of a compression function to successive blocks of the message (Fig. 1). For MD5, the compression function takes two inputs - a 128-bit chaining value and a 512-bit message block-and produces as output a new 128-bit chaining value, which is the input to the next iteration of the compression function. The message to be hashed is first padded to a multiple of 512 bits and then divided into sequence of 512-bit message blocks. Then the compression function is rapidly applied, starting with an initial chaining value and the first message block and continuing with each new chaining value and successive message blocks. After the last message block has been processed, the final chaining value is then passed to the output as the hash of the message.

Because of the iterative design, it is possible, from only the hash of a message, to compute the hash of longer messages that start with the initial message and include the padding required for the initial message to reach a multiple of 512 bits. Padding data helps to add random bytes to our data so that it is more difficult for a prospective attacker to find which bytes are the actual data. Padding also allows us to put our data into blocks, so that we can operate on pieces of data that are of the same size[11]. Applying this to the prefix approach, it follows that from MD5(k·m), one can compute MD5(k·m′) for any m′ that starts with m·p, where p is the padding on (k·m). In other words, from the message authentication code of m·p·x for any x, without even knowing the key k and without breaking MD5 in any sense. This is called a message extension or padding attack[12]. Taking into account the joint work of Kaliski *et al.*[13] and Bellare and Krawczyk of IBM[14] and a number of other approaches to message authentication with MD5, we are going to settle on three which are recommended to the Internet Protocol Security (IPSEC) working group: (i)MD5(k_1·MD5(k_2·m)) where, k_1 and k_2 are independent 128-bit keys; (ii) MD5(k·p·m·k), where, k is a 128-bit key and p is 384 bits of padding and (iii) MD5(k·MD5(k·m)), where, k is a 128-bit key.

The first and third approaches (Fig. 2) are similar and solves the message extension attack on the prefix approach by the outer application of MD5, which conceals the chaining value, that is needed for the attack. The outer MD5 also solves the concerns of cryptanalysis of the suffix approach, because the message

authentication code is a function of the unknown secrete-key and other varying values, which are unknown. These approaches also approximate certain provably secure constructions developed by Bellare *et al.*[14,15]. The message authentication code is computed by combining, perhaps bit-wise exclusive-or (XOR), the outputs of the pseudorandom (PRNG) function applied to the blocks of the message. A random block is also included for technical reasons[16].

Bellare *et al.*[14,15] techniques assume the existence of pseudorandom function, which takes two inputs, a key and a message block and produces one output. By assumption, if the key input is fixed and unknown, it is difficult to distinguish the pseudorandom function on the message block from a truly random one in any reasonable amount of time[17]. (This is similar to the idea that it is difficult to find collisions for a hash function-although it is possible because they exist, however, the amount of time required is large!) Bellare *et al.*[14,15] also showed that if an opponent can forge message authentication codes, even with the opportunity to request message authentication codes on many different messages, then the opponent could also distinguish the pseudorandom function from truly random one. Thus, under the assumption that it is difficult to distinguish the pseudorandom function from a truly random one, the message authentication code is secure[18].

The independent processing of the message blocks leads to the parallelizability of this approach. It seems that many of the concerns about designing a message authentication code from a hash function are a consequence of the fact that the key is processed only once, or maybe twice. As a result, the key is isolated and information about it can be manipulated independent of the key. By contrast, in message authentication codes based on encryption functions, such as DES-MAC, the key is processed at every step which is also the case with Bellare's *et al.*[13] techniques.

THE MECHANICS OF THE HASH ALGORITHM

It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given pre-specified target message digest. This is a fingerprint for the data. A digest has two main properties: In the first case, if even one single bit of data is changed, then the message digest changes as well and there is a very remote probability that two different arbitrary message can have the same fingerprint. Secondly, even if someone was able to intercept transmitted data and its fingerprint, that person would not be practically able to modify the original data so that the resulting data has the same digest as the original one. Hashing functions are often found in the context of digital signature. For secure electronic signatures, in a Public Key Infrastructure (PKI) procedure, a hash function must be collision-resistant, which means that it is computationally infeasible to find two different documents yielding the same hashcode (alternatively, it is infeasible to find a different document yielding the same hashcode as a given document). This is a method for authenticating the source of the message, formed by encrypting a hash of the source data. Public key encryption is used to create the signature so it effectively ties the signed data to the owner of the key pair that created the signature[19]. Some of the currently approved hash functions are: SHA-1, MD5, RIPEM-128 and RIPEM-160 etc. The MD5 and SHA1 algorithms are the most commonly used in 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[19,20].

A JAVA IMPLEMENTATION OF MESSAGE DIGEST ALGORITHMS

The program in Listing 1 shows how to use java.security to create a message digest (also called a hash value): in particular SHA-1, MD5, RIPEM-128 and RIPEM-160 are presented here. Some further information on the use of message digests in digital signatures may be found[21-25]: The program shown in Listing 1: reads in the a Plaintext data file and calculates a SHA-1, MD5, RIPEM-128 and RIPEM-160 hash values, which is printed in a file hashout. Listing 2 shows the helper code for hex manipulation.

Listing 1: Message digest source code (HashCodesGenerators.java)

```
package com.rsc.messagedigest;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.security.MessageDigest;

class HashCodesGenerators {

public static void main(String[] args) {
```

```
try {
        FileOutputStream outFile = new FileOutputStream("hashout");
        PrintStream output = new PrintStream(outFile);

        byte b;
        String w;

        //SHA-1 Hash value of data
        MessageDigest sha = MessageDigest.getInstance("SHA-1");

        FileInputStream fis = new FileInputStream(args[0]);
        while (fis.available() != 0) {
                b = (byte) fis.read();
                sha.update(b);
                };

 fis.close();

byte[] hash = sha.digest();
h = new String(HexCodec.bytesToHex(hash));
output.println("SHA-1 hash of " + args[0] + " is: " + h);

//MD5 Hash value of data
MessageDigest md5 = MessageDigest.getInstance("MD5");

fis = new FileInputStream(args[0]);
while (fis.available() != 0) {
        b = (byte) fis.read();
        md5.update(b);
        };

 fis.close();

hash = md5.digest();
h = new String(HexCodec.bytesToHex(hash));
output.println("MD5 hash of " + args[0] + " is: " + h);

//RIPEM128 Hash value of data
MessageDigest rpm128 = MessageDigest.getInstance("RIPEMD128");

fis = new FileInputStream(args[0]);
while (fis.available() != 0) {
        b = (byte) fis.read();
        rpm128.update(b);
        };

fis.close();

hash = rpm128.digest();
//new String(HexCodec.bytesToHex(timestampBytes));
h = new String(HexCodec.bytesToHex(hash));
output.println("RIPEM128 hash of " + args[0] + " is: " + h);

//RIPEM160 Hash value of data
MessageDigest rpm160 = MessageDigest.getInstance("RIPEMD160");

fis = new FileInputStream(args[0]);
while (fis.available() != 0) {
        b = (byte) fis.read();
        rpm160.update(b);
        };

fis.close();

hash = rpm160.digest();
h = new String(HexCodec.bytesToHex(hash));
output.println("RIPEM160 hash of " + args[0] + " is: " + h);

        } catch (Exception e) {
                System.err.println("Caught exception " + e.toString());
        }
}}
```

Listing 2: Helper class source code (HexCodec.java)

```
package com.rsc.messagedigest;

public class HexCodec {
        private static final char[] kDigits = {
                '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                'a', 'b', 'c', 'd', 'e', 'f'
                };

public static char[] bytesToHex(byte[] raw) {
        int length = raw.length;
        char[] hex = new char[length * 2];
        for (int i = 0; i<length; i++) {
        int value = (raw[i] + 256) % 256;
        int highIndex = value>>4;
        int lowIndex = value & 0x0f;
        hex[i * 2 + 0] = kDigits[highIndex];
        hex[i * 2 + 1] = kDigits[lowIndex];
        }
        return hex;
        }

public static byte[] hexToBytes(char[] hex) {
        int length = hex.length / 2;
        byte[] raw = new byte[length];
        for (int i = 0; i<length; i++) {
        int high = Character.digit(hex[i * 2], 16);
        int low = Character.digit(hex[i * 2 + 1], 16);
        int value = (high<<4) | low;
        if (value>127) value-= 256;
        raw[i] = (byte)value;
        }
        return raw;
        }

public static byte[] hexToBytes(String hex) {
        return hexToBytes(hex.toCharArray());
        }
}
```

Here is the data file of the message to be digested shown in Listing 3.

Listing 3: Sample text message input (data)

```
Miss Alice Johnson
Nairobi, Kenya

Dear Miss Johnson,

This letter is to inform you that we have received US$ 2.0 million as your
first installment due for your mortgage down payment and opened mortgage
servicing account no. 6688668024 on your behalf and deposited therein said
amount. To service your account you will need to use your new password
GKGQU78BR53.

Yours very truly,

James M Kavungu
Vice President, Finance
First Finance Bank of Nairobi
```

Compile the program using java interpreter: javac HashCodesGenerators.java. You will get two classes output: HashCodesGenerators.class and HexCodec.class. Next execute the program as follows: java HashCodesGenerators data. Listing 4 shows the output file hashout which is the output of the digested file data.

Listing 4: Hash values output (file: hashout)

```
SHA-1 hash of data is: e5e5137567a6a39d385d203222cc73052d9adb31

MD5 hash of data is: 7bcc3d267467a8f037650c0a6c43f018

RIPEM128 hash of data is: 20ebfa7e21bcb1f64c3643bad1531d33

RIPEM160 hash of data is: cddf358db7bbec17432e6670a2754bac9529595a
```

DIGITAL SIGNATURE AND AUTHENTICATION

For a digital signature, the main idea is no longer to disguise what a message says, but rather to prove that it originates with a particular sender. Digital signatures have been used in Internet applications to provide data authentication and non-repudiation services. Digital signatures will keep on playing an important role in future Internet applications. For example, if electronic mail systems are to replace the existing paper mail systems for business transactions, signing an electronic message must be possible. One way to address the authentication problem encountered in public-key cryptography is to attach digital signature to the end of each message that can be used to verify the sender of the message[26,27]. The

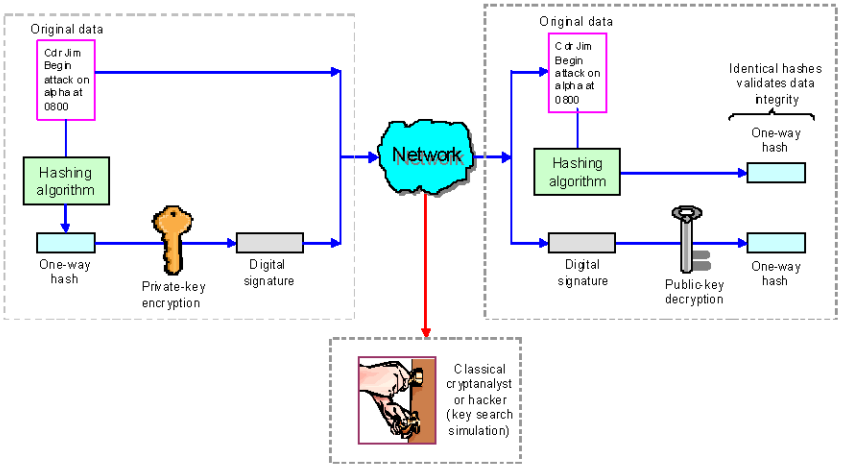

Fig. 3: An implementation of Digtal Signature alagorithm

significance of a digital signature is comparable to the significance of a handwritten signature[19]. In some situations, a digital signature may be as legally binding as a handwritten signature. Once you have signed some data, it is difficult to deny doing so later - assuming that the private-key has not been compromised or out of the owner's control. This quality of digital signatures provides a high degree of nonrepudiation - i.e., digital signatures make it difficult for the signer to deny having signed the data. This quality is stronger than mere authentication (where the recipient can verify that the message came from the sender); the recipient can convince a judge that the signer sent the message. To do so, he must convince the judge he did not forge the signed message himself! In authentication problem the recipient does not worry about this possibility, since he only wants to satisfy himself that the message came from the sender.

Figure 3 shows two items transferred to the recipient of some signed data: the original data and the digital signature, which is basically a one-way hash (of the original data) that has been encrypted with the signer's private-key[28,29]. To validate the integrity of the data, the receiving software first uses the signer's public-key to decrypt the hash. It then uses the same hashing algorithm that generated the original hash to generate a new one-way hash of the same data. (Information about the hashing algorithm used is sent with the digital signature.) Finally, the receiving software compares the new hash against the original hash. If the two hashes match, the recipient can be certain that the public-key used to decrypt the digital signature corresponds to the private-key used to create the digital signature. If they don't match, the data may have been tampered with since it was signed, or the signature may have been created with a private-key that doesn't correspond to the public-key presented by the signer. Confirming the identity of the signer, however, also requires some way of confirming that the public-key really belongs to a particular person or other entity and this is achieved via the use of fingerprinting.

In short, an electronic signature must be a message-dependent, as well as signer-dependent. Otherwise the recipient could modify the message before showing the message-signature pair to the judge. Or he could attach the signature to any message whatsoever, since it is not possible to detect electronic cutting and pasting. To implement signatures the public-key cryptosystem must be implemented with trap-door one-way function, since the decryption algorithm will be applied to unenciphered messages. For a discussion of the way this works let's look at the communication between two entities, as depicted in Fig. 3. Recall that, a digital signature is analogous to an ordinary hand-written signature used for signing messages. Therefore, it must be unique and private to the signer. More specifically, suppose that entity B is the recipient of a message m signed by entity A[19]. Then, A's signature must satisfy three requirements: (i) B must be able to validate A's signature on m easily, (ii) It must be impossible for anyone, including B, to forge A's signature and (iii) It must be possible for a judge or third party to resolve any dispute between A and B.

A theoretical approach to digital signature scheme in a public-key cryptosystems: Suppose two entities Alice (A) and the Bob (B) wants to set up a communication channel capable of performing digital signature

procedure. So how can user B (Bob) send user A (Alice) a signed message m_B in a public-key cryptosystems? (Here, the subscript indicates the respective entity's initial; while m = message, E and D are encryption (or private-key) and decryption (or public-key) procedures, respectively, S represents signature algorithm.) To accomplish this, Bob first uses his own secret-key or D_B to encrypt his digital signature, S_B according to $D_B(S_B)$. (Deciphering the unenciphered message makes sense - a property of public-key cryptosystem: each message is the ciphertext for some other message.). For detail discussion readers are referred to ref.[19]. He then appends his encrypted digital signature to his personal message, m_B, to produce the signed message, $m_B \cdot D_B(S_B)$, where the dot denotes concatenation. Next Bob applies an encryption procedure to his signed message, using Alice's public-key E_A (for privacy) which is available on the public-key sever, to obtain the ciphertext: $C_B = E_A(m_B \cdot D_B(S_B))$ and transmits it to Alice.

When Alice receives C_B she first applies her private decryption procedure using her secret-key to produce, $D_B(S_B) = m_B \cdot D_B(S_B)$. Thus, the message m_B will appear, along with a portion of gibberish at the end of the message. To authenticate m_B, Alice uses Bob's public-key, E_B, (available on the public-key server) to perform, $E_B(D_B(S_B)) = S_B$. If Bob's digital signature appears, she knows the message is authentic.

She now possesses a message-signature pair (m_B, S_B) with properties similar to those of signed paper document. Bob cannot deny having sent Alice this message, since no one else could have created $S_B = D_B(m^B_{DS})$ (where m^B_{DS} is Bob's plaintext digital signature message). Alice can convince a judge that $E_B(S_B) = m^B_{DS}$ and so she has proof that Bob signed the document.

Clearly, Alice cannot modify m_B to a different version m'_B, since then she would have to create the corresponding signature, $S'_B = D_B(m'_B)$, as well. Therefore, Alice has received a message signed by Bob; which she can prove that he sent, but which she cannot modify. (Nor can she forge his signature for any other message).

The only remaining issue involves selecting the digital signature S_B. If Bob uses the same digital signature in every message, Eve (the person in the middle or attacker) will be able to detect this by looking for common string among Bob's transmissions. Even though by doing this Eve will only discover $D_B(S_B)$, this all she needs in order to sign a rogue message and misrepresent herself as Bob to Alice.

Therefore, it is important for Bob to use a different S_B in every message. One strategy is to make S_B depend on the message m_B. Hash functions are commonly used to implement this strategy. In this setting a public hash function h is required to transform a variable-length message into a fixed-length message fingerprint F, i.e., $h: T_B \rightarrow F$. Bob's ciphertext message to Alice is now encrypted using, $C_B = E_A(T_B \cdot D_B(F))$. After applying D_A to this ciphertext, Alice can authenticate it by first computing $E_B(D_B(F)) = F$ and then comparing this result to the result she obtains by applying the hash function h to m_B.

However, there are some instances in which the main security issue is authentication and not secrecy. For example, a financial institution may be content with sending and receiving their transactions unencrypted, as long as they can guarantee that these transactions are not altered. Specifically, if Bob sends messages, $m_B \cdot D_B(F)$ to Alice, then even though Eve can read m_B, however, she will not be able to alter it unless she is able to determine F.

In order for a public hash function h to be effective, it must posses at least the following two properties: First, since h is known to all, for any y it must be computationally intractable to find m_B such that, $h(m_B) = y$. In other words, Eve should have great difficulty in trying to invert h in order to obtain m_B. Second, it should be computationally intractable to find messages that collide[30]. To see why, assume we have a hash function h that does not satisfy this property. Now suppose Eve constructs two messages m_B and m'_B such that, $h(m_B) = h(m'_B)$ and Bob is perfectly happy to sign m_B but not m'_B. If Eve can convince Bob to sign m_B, then Eve will also be able to achieve her fraudulent goal of signing m'_B with Bob's digital signature.

Many digital hashing schemes are based on the following idea. Let h′ be a hash function that maps s-bit keys to k-bit values, for some fixed s>k. From h′ we construct a public hash function that produces a k-bit messages fingerprint by first breaking the message T_B into blocks, T_{B1}, T_{B2},....., T_{Bt} each containing, s-k bits. Next let: $F_i(T_{Bi}) = h'(F_{i-1} \cdot T_{Bi})$, where the dot denotes concatenation and F_0 is a k-bit initialization value, often chosen as all zeros. The message fingerprint is then given by F_1. Now that we have a kind of a basic idea how digital signature is accomplished in theoretical sense, how can it be packaged to allow for its use in real application! Further, you might have observed that this technique implements digital signature scheme using encryption algorithm, a subject which is dear to US-export regulator, who do not allow the export of high quality cryptographic procedures. So how do we go around this? And that is the subject of the next section looking at how Digital Signature Algorithm (DSA) came to be. But before taking on the DSA, we will take a plunge and have a look at the Discrete Logarithm Problem (DLP), another mathematical tool that is useful for implementing public-key crypto-algorithm procedures.

The mechanics of Discrete Log Problem (DLP): The most important tool necessary for the implementation of public-key cryptosystems is the Discrete Log Problem (DLP). Many popular modern crypto-algorithms base their security on the DLP. Based on the difficulty of this problem, Diffie-Hellman[31] proposed the well-known Diffie-Hellman key agreement scheme in 1976. Since then, numerous other cryptographic protocols whose security depends on the DLP have been proposed, including: the ElGamal encryption and signature scheme[32], the U.S. government digital signature (DSA)[33] is perhaps the best known example of a DLP system, the Schnorr signature scheme[34] and the Nyberg-Reuppel signature scheme[35]. Due to interest in these applications, the DLP has been extensively studied by mathematicians for the past 20 years. The mathematical challenge here lies in computing discrete logarithms in finite fields of type Z_p, which consist of the integers modulo a large prime p. Although this problem can be considered difficult, there are known sub-exponential time algorithms for solving it, such as the number field sieve. In practical terms, sub-exponential time means that a determined hacker with enough processing power can break the system in a few months.

In an (abelian) group G (multiplicatively written) we can consider the equation $y = x^n$, $x, y \in G$, $n \in Z$. If x and y are known real numbers and it is also known that x is some power (say, n) of y, then logarithms can be used to find n("$= \log_x(y)$") in an efficient manner. However, if x and y are given such that: $y = x^n = x \cdot x \cdot \ldots \cdot x$ (n-times), then in general it is technically much harder and hence the determination of n cannot be carried out in a reasonable amount of time. This is equivalent to the well-known real logarithm, we call n the discrete logarithm of y related to the base x[34]. The operation exponentiation $x \neg y: = x^n$ can be implemented as a quick, efficient algorithm.

The Discrete Logarithm Problem (DLP) is the following: given a prime p, a generator g of Z_p and a non-zero element $\beta \in Z_p$, find the unique integer k, $0 \le k \le p-2$, such that $\beta = g^k$. The integer k is called the discrete logarithm of β to the base g. Here, Z_p denotes the set of integers {0, 1, 2,....., p-1}, where addition and multiplication are performed modulo p. It is well-known that there exists a non-zero element $g \in Z_p$ such that each non-zero elements in Z_p can be written as a power of g; such that an element g is called a generator of Z_p.

The corresponding problem in additive (i.e., abelian) groups is: given P and kP (P added to itself k times), find the integer k. This is much more difficult! There is no one-step operation like taking logarithms that we can use to get the solution. So we may know P and kP and yet not be able to find k in a reasonable amount of time. This is called the Discrete Log Problem for abelian groups. We could always repeatedly subtract P from kP till we got 0. But if k is large, this will take us a very long time. Several important cryptosystems are based on the difficulty of solving the DLP over finite abelian groups. The solution is even tougher if the underlying group arises from an elliptic curve over a finite field.

Standard DLP cryptosystems are based on multiplicative groups with the main operation of exponentiation. In elliptic curve cryptography (ECC)[36], the multiplicative group is replaced by the additive group of elliptic curve points and exponentiation operation by scalar multiplication of a point (i.e. calculation of $g^k = g \cdot g \cdot \ldots \cdot g$ for a generator g of a multiplicative group is replaced by calculation of $[k]P = P + P + \ldots + P$ (k-times) for a generator point P of an additive group of elliptic curve points). Thus, the computational performance of cryptographic protocols based on elliptic curves strongly depends on efficiency of the scalar multiplication.

Digital signature algorithm: The Digital Signature algorithm (DSA) was proposed in August 1991 by the U.S. National Institute of Standards and Technology (NIST) for use in their Digital Signature Standard (DSS) and, was later specified in a U.S. Government Federal Information Processing Standards (FIPS 186) called the Digital Signature Standard (DSS). It was designed at the NSA as part of the Federal Government's attempt to control high security involving cryptography. Part of that policy included prohibition (with severe criminal penalties) of the export of high quality encryption algorithms. The DSS was intended to provide a way to use high security digital signatures across borders in a way which did not allow encryption. Those signatures required high security asymmetric key encryption algorithms, but the DSA (the algorithm at the heart of the DSS) was intended to allow one use of those algorithms, but not the other. It didn't work. DSA was discovered, shortly after its release, to be capable of encryption (prohibited high quality encryption, at that), however, it is so slow when used for encryption as to be even more than usually impractical.

The US government based their Digital Signature Algorithm (DSA) on much of ElGamal's study[32] and is the best known example of a large system where the Discrete Logarithm (DL) algorithm is used. Its security is based on the intractability of the discrete logarithm problem (DLP) in prime-order subgroup of Z^*_p. As with the RSA algorithm, these transformations raise the computational complexity of the problem. The discrete logarithm system relies on the discrete logarithm problem modulo p for security and the speed of calculating the modular exponentiation for efficiency. In terms of computational difficulty, the discrete logarithm problem seems to be on a par with factoring[29].

The mechanics of Digital Signature Algorithm (DSA): The Signature-Creation Data (SVD) consists of the public parameter an integer y computed as: $y = g^x \bmod p$, as per the DLP above. Note that p and q are large prime numbers[37] When computing a signature of a message M, no padding of the hashcode is necessary. However, the hashcode must be converted to an integer by applying the method described in Appendix 2.2[37].

The basic idea of DSA is for the signer of message M - that is, the possessor of the value x behind the publicly known, $g^x \bmod p$ - to append a pair of numbers r and s obtained by secretly picking another number k between 1 and q, computing $r = (g^k \bmod p) \bmod q$ (i.e., computing $g^k \bmod p$ and then taking the remainder of that number mod q) and $s = k^{-1}(SHA(M) + xr) \bmod q$, where k^{-1} is the multiplicative inverse of k(mod q) and SHA is the Secure Hash Algorithm. He then sends (M, r, s) to the communicating partner. Another NIST standard, SHA (official acronym is SHA-1) reduces a character string of any length to a 160-bit string of gibberish. In the implementation of DSA, q is a 160-bit prime divisor of p-1 and g is an element of order q in F^*_p.

The receiver of (M, r, s) from person g^x computes $u = s^{-1} SHA(M) \bmod q$ and $v = s^{-1} r \bmod q$ and then checks that $((g^u)(g^x)^v \bmod p) \bmod q$ equals r. If it doesn't, then, by elementary number theory, something definitely went wrong. If it does, then, according to NIST, you can safely assume that message M came from the presumably unique individual who knows the discrete logarithm of g^x. Table 1 shows the sequence of DSA scheme.

Key and parameters generation algorithm: The prime numbers p and q shall be generated following the accepted procedure suitable for cryptographic prime number generators[38]. The integer x generated by applying a random number generation method that satisfies the requirements for true random number generator (TRNG) or using a method satisfying pseudorandom number generator (PRNG)[39] with an appropriate size seed. Each value of x shall effectively be influenced by EntropyBits bits of true randomness or a seed of appropriate length. Finally, generate k using one of these methods; k does not have to be generated using exactly the same method as x. The DSA requires that q be a 160-bit prime and p a prime with between 512 and 1024 bits.

Random number requirements: As already observed above, there are two main types of random number generators used in cryptography: the true random number generator (TRNG) and the pseudorandom number generators (PRNG). The aim of a TRNG is to generate individual bits, with uniform probability and without any correlation between those bits. Consequently, the knowledge of some bits does not give any information (in a strong information theoretic sense) about the other generated bits. However, achieving this task in real practice appears to be a difficult possibility. Consequently, the cryptographic designers and implementers often do resort to pseudorandom bit generators (PRNG) in many applications. However, due to the deterministic nature of the PRNG, they are not generators of truly random bits but, starting with a random seed, they are able to generate sequences of bits that are random in behavior.

So how does one get the sense and feel of what a random number is like?: For a start, a physical random generator is based on a physical noise source (usually a primary noise) which is coupled to a cryptographic or mathematical post-treatment of the primary noise. Next,

Table 1: Digital Signature Algorithm (DSA)

Digital Signature Algorithm (DSA)
Key Generation
1. Choose an L-bit prime p, where $512 \le L \le 1024$ and L is divisible by 64
2. Choose a 160-bit prime q, such that p-1 = qz, where z is any natural number
3. Choose h, where $1<h<p-1$ such that $g = h^z \bmod p > 1$
4. Choose x by some random method, where $0<x<q$
5. Calculate $y = g^x \bmod p$
6. Public key is (p, q, g, y). Private key is x
Note that (p, q, g) can be shared between different users of the system, if desired
Signing
1. Choose a random per message value k (called a nonce), where $1<k<q$
2. Calculate $r = (g^k \bmod p) \bmod q$
3. Calculate $s = k^{-1}(H(m)+xr) \bmod q$, where H(m) is the SHA-1 hash function applied to the message m
4. Signature is (m, r, s)
Nonce means 'for the present time' or 'for a single occasion or purpose'
Verifying
1. Calculate $w = s^{-1} \pmod q$
2. Calculate $u1 = wH(m) \pmod q$
3. Calculate $u2 = wr \pmod q$
4. Calculate $v = (g^{u1} * y^{u2} \bmod p) \bmod q$
5. Signature valid if v = r
DSA is similar to ElGamal discrete logarithm cryptosystem signatures[32].
Implementation of Digital Signature Algorithm (DSA)
Here we will use an overly small prime and integer numbers to show how DSA can be implemented in real applications.
Key Generation
7. Choose a prime number: p = 767813, which gives q = 191953 and z = 4.
8. Choose h = 67, such that $g = h^z \bmod p = 187983$
9. Choose x = 23 and compute: $y = g^x \bmod p = 187983^{23} \bmod p = 460280$
10. Public key is: (p, q, g, y). Private key is: x = 23.
Signing
5. Choose a random per message value k = 79 (called a nonce), where $1<k<q$
6. Compute $r = (g^k \bmod p) \bmod q = 143753$
7. Choose m = H(m) = 4076 and compute $s = k^{-1}(H(m)+xr) \bmod q = 95359$
8. Signature is (m, r, s) = (4076, 143753, 95359)
Nonce means 'for the present time' or 'for a single occasion or purpose'
Verifying
6. Compute: $u1 = s^{-1} H(m) \pmod q = 16935$
7. Compute: $u2 = s^{-1}r \pmod q = 157837$
8. Compute $v = (g^{u1} * y^{u2} \bmod p) \bmod q = 143753$
9. Signature valid, since: v = r = 143753.

the primary noise must be subjected to an adapted statistical test on a regular basis. Following this approach, the expected cryptanalyst effort of guessing a cryptographic key shall be at least equivalent to guessing a random value that is EntropyBits long[21]. The notion of entropy has to be used very carefully in practice, since it applies to probability distributions and not to actual bit strings.

So how do we overcome this difficulty in practice?: Due to unavailability of perfect random sources, the practicing cryptographers go round this problem, by using a source of bits that may not be perfectly random (e.g., PRNG) followed by hashing the bits in order to obtain really random bits. Here, we assume that a hash function, such as SHA-1, is able to extract the randomness from a biased bit string. Usually, the amount of randomness of such a string is measured by its entropy. From a more practical point of view, it is useful to consider that a random source generates sequences of bits and that only a ratio is random. For example, on evaluation we may find that for eight generated bits we have one bit of randomness; consequently, hashing 1280 generated bits with SHA-1 will hopefully produce 160 truly random bits.

A java implementation of secure random generator: In the implementation of DSA, the class java.security. SecureRandom is used. The class provides a cryptographically strong Pseudo-Random Number Generator (PRNG). The package java.security also offers the class SecureRandomSpi, which defines the SPI for SecureRandom. Let's consider the following instruction:

```
SecureRandom r = new SecureRandom();
```

This obtains a SecureRandom object containing the implementation from the highest-priority installed security provider (SUN, in our case) that has a SecureRandom implementation. Another way to instantiate a SecureRandom object is via the static method getInstance(), supplying the algorithm and optionally the provider implementing that algorithm:

```
SecureRandom random = SecureRandom.getInstance ("SHA1PRNG", "SUN");
```

CRYPTOGRAPHIC DESIGN CRITERIA AND PACKAGING

The DSA cryptosystem, as we have already seen, requires a high level of mathematical abstraction to implement. The good news about implementing any cryptography is that we already have the algorithms and protocols we need to secure our systems[40,41]. The bad news is that that was the easy part; implementing the protocols successfully requires considerable expertise from the software developers and designers. Cryptographic algorithms do not in themselves guarantee security. There is an enormous difference between a mathematical algorithm and its concrete implementation in hardware or software. Moreover, cryptographic system designs are fragile. Just because a protocol is logically secure doesn't necessarily mean it will stay secure when a designer starts defining message structures and passing bits around. The entire systems design must be implemented exactly, perfectly, or they will fail. A poorly designed user interface can make a hard-drive encryption program completely insecure. A false reliance on tamper-resistant hardware can render an electronic commerce system all but useless. Since these mistakes aren't apparent in testing, they do end up in finished products. Hence, a designer must strike a balance between security and accessibility, anonymity and accountability, privacy and availability. One significant practical problem if this system is to be useful is how can it be packaged in a user-friendly way so that developers can incorporate it into their applications with minimal knowledge of its inner workings. A solution to this problem would be the provision of a relatively simple interface to provide security while hiding the details from users. This interface should be able to support and swap cryptographic algorithms with ease and support related cryptographic concepts like key management in an easy to use way. Further, the interface should also be flexible enough to allow future incorporation of new cryptographic algorithms as the need arises. Fortunately, this task is very easy to accomplish in Java using the Java Cryptography Architecture (JCA)[42] and Java Cryptographic Extension (JCE)[43] to develop and deploy our interface framework.

The JCA and JCE in Java: The beauty of using Java programming for cryptographic design and deployment is that the Java Cryptography Architecture (JCA) is already included in the Java 2 run-time environment distributed by Sun. It includes algorithms to perform message digests, create digital signatures and generate key pairs. The classes included in the JCA are available in java.security package, including: MessageDigest, Digital Signature, KeyPairGenerator and SecureRandom. There are no algorithms to perform a ciphers process which is necessary for the implementation of encryption/ decryption processes, because when JCA was first released, export restrictions would not allow Sun to distribute such algorithms. The lack of cipher algorithms

later led to the release of the Java Cryptographic Extension (JCE), which include the encryption and decryption cipher algorithms. It also includes algorithms to generate single keys and secret-keys. The usage of the classes in the JCA and JCE are virtually identical. The classes included in the SunJCE are located in the javax.crypto package, which include: CipherSuits, KeyGenerator, PublicKey and SecretKey. The JCA and JCE classes are all very well perfected and time tested with both the classical and traditional symmetric and asymmetric cryptographic algorithms. Support for encryption includes symmetric, asymmetric, block and stream ciphers. The software also supports secure streams and sealed objects. The default provider shipped with the JCA and JCE is Sun's provider, java.security.provider.Sun. The JCA and JCE use the factory pattern, which is a pattern that defines an interface for creating an object, but lets the subclasses decide which class to actually instantiate[40]. For example, we can generate key pair using KeyGenerator class as follows:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance
                       ("Algorithm", "CryptoProviderName");
```

Here, the String Algorithm refers to respective algorithm and String CryptoProviderName refers to security provider name. This is simple enough and allows one to easily change to algorithm of interest without necessarily using new operator, as is normally true with objects in order to create an instance of a class. Every algorithm must be associated with a provider and multiple providers can support any single given algorithm. A provider is the underlying implementation of a particular security mechanism. If no provider is specified, then the Java Virtual Machine (JVM) will use the first implementation it finds, according to the preference list in the java.security file. As you can see all the hard stuff are implement in the background and the user and/or developer is left to concentrate on software development. Therefore, to implement ones new algorithm, you only need to change the string that refers to the algorithm e.g.,

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA", "SUN");
```

Here, the String DSA is the algorithm and String SUN refers to java.security.provider with the provider from Sun Microsystems, for example, which comes as a default provider with Java 2. There are several existing providers, some of which are freely available and others that are quite expensive, if say, one is interested to implements ones own provider. Companies that offers java.security providers include IBM[44] and RSA[45] both under commercial license and, Bouncy Castle which is an Open Source license (freely available for download and use)[45].

A JAVA IMPLEMENTATION OF DSA KEY PAIR GENERATION

Recall from above that the package java.security does provide APIs for the message digest and digital signatures processing. Using this package, one can generate the pair of keys required to process digital signature schemes by creating the instance of a KeyPairGenerator object via the static method getInstance(), supplying the DSA algorithm and, optionally the provider implementing the algorithms. Next you initialize it with the desired key size in number of bits and, optionally a secure random provider. Then you call the generateKeyPair() method to generate the DSA key pair:

```
KeyPairGenerator KPG = KeyPairGenerator. getInstance("DSA", "SUN");
KPG. initialize(1024, r);
KeyPair KP = KPG.generateKeyPair();
```

The algorithm is passed to the factory getInstance() method as a String. If the algorithm is not supported by the installed provider(s), a NoSuchAlgorithmException is thrown. Each provider must supply (and document) a default initialization. If the provider default suits your requirements, you don't have to save the intermediate KeyPairGenerator object.

If you need to generate more than one key pair, you can reuse the KeyPairGenerator object; otherwise, you can simply generate the key pair with one line of code. This gives you much better performance than using a new KeyPairGenerator object every time. Listing 5 shows the complete listing of our above code fragments.

Listing 5: DSAKeyMaker for generating key pair.

```
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.KeyPair;

    public class DSAKeyMaker {

    public static void main(String[] args) {
                String algorithm = "";
                if (args.length == 1) algorithm = args[0];

                try {
                KeyPair keyPair = KeyPairGenerator
                .getInstance(algorithm)
                .generateKeyPair();

     System.out.println(keyPair.getPublic());
     System.out.println(keyPair.getPrivate());

  } catch (NoSuchAlgorithmException e) {
     System.err.println("usage: java DSAKeyMaker DSA");
  }

  }
}
```

Listing 6: the output from running DSAKeyMaker

```
Sun DSA Public Key
        Parameters:
        p:
        9925ed8d a7a8a3af e5e84dc8 0d4b34bf c706d006 291123eb 221c6d0f 04134cd7
        b4f910cc 62b1f2bb f5998247 a7e31804 36783c3f 551b5e45 a5a606b6 c2275527
        40161cfb fccb0298 554c58e9 78fdc1c6 7cab85f2 9da393ed 0d792d92 c61d83b5
        174bf813 8579ac32 fb0101a0 fb069d19 5f5a081e 69dfe8fe c03bc58d a109287f
        q:
        86882717 7f91150e e47d0937 aadc2d48 900e43d1
        g:
        5ac700d0 084b4c1c d4913ac2 f99bcf76 9e806dfd f2f934da 00383beb e38bb5b5
        89bb229a 8c4f8b3c ff971b04 4ceb0dcb 3696d5e1 59f916d6 c33d84e3 82c1e67e
        206aef91 5b9e07a1 f2e0f460 8f8299d0 12a9ebae 0b296771 4627b00e 3dfacf99
        d74abe91 0e66c732 f1546018 2ba4da51 2b05a690 ac3b996d a687f932 d2f10109
        y:
        956bd776 10d29fe4 3b812819 5ba12491 0ff90fb0 96ef0ee7 083baaf6 6e5554ec
        95869082 531056e7 134a09cd 5c86349e 16019beb a75e3cb2 6b9838bf 132aa90e
        298088ef a69f44ac 244bd647 45b2228f 969263e6 bf69ed6c 7d0c72d0 53665762
        c7521e6c 92058b97 86836d7f aedec4d5 f4c82222 84b9a3eb fd84440b 6f8b9926

Sun DSA Private Key
parameters:
        p:
        9925ed8d a7a8a3af e5e84dc8 0d4b34bf c706d006 291123eb 221c6d0f 04134cd7
        b4f910cc 62b1f2bb f5998247 a7e31804 36783c3f 551b5e45 a5a606b6 c2275527
        40161cfb fccb0298 554c58e9 78fdc1c6 7cab85f2 9da393ed 0d792d92 c61d83b5
        174bf813 8579ac32 fb0101a0 fb069d19 5f5a081e 69dfe8fe c03bc58d a109287f
        q:
        86882717 7f91150e e47d0937 aadc2d48 900e43d1
        g:
        5ac700d0 084b4c1c d4913ac2 f99bcf76 9e806dfd f2f934da 00383beb e38bb5b5
        89bb229a 8c4f8b3c ff971b04 4ceb0dcb 3696d5e1 59f916d6 c33d84e3 82c1e67e
        206aef91 5b9e07a1 f2e0f460 8f8299d0 12a9ebae 0b296771 4627b00e 3dfacf99
        d74abe91 0e66c732 f1546018 2ba4da51 2b05a690 ac3b996d a687f932 d2f10109
        x:
        569eeea5 4ca7525a 3a1d68b3 6edcc5d1 7fd0357a
```

The program compiled using java interpreter on command line as: javac DSAKeyMaker.java. Next execute the program as follows: java DSAKeyMaker DSA to get the values for p, q and g in both the public and private keys. A value for y is provided in the public key and for x in the private key, which should look like Listing 6.

The output above highlights the value of properly overloading the toString() method in Java programming. Two important points to note from Listing 6: the lines beginning with SunJSSE DSA public key: and SunJSSE DSA private key, these are the results of calling the toString() method in the class DSAPublicKey and class DSAPrivateKey, respectively with each class generating their respective keys and parameters required for signature algorithm or encryption processes. If your interest is to encrypt data you should be careful about how you transmit the public key values to your communicating partners. However, if the values are being used to authenticate a transmission, you should make them publicly available so that others can verify that a file, for example, originated with you and was delivered unaltered.

So how do you use this key pair to sign messages?: In the case of DSA, you begin by taking some text that you want to sign and turn it into a number m and follow the procedure given in Table 1. The RSA, for example, has a simple way of doing this. If the owner of a code (N, e) wants to prove that she's the sender of a message m, she can use her private decoding exponent d to compute $c = m^d \bmod N$ and then send both C and m, i.e., (C, m)[19]. The receiver can then persuade himself that the message truly originated with the owner of d by computing C^e and checking that it's the same as m. For example, take a message m = 3. Digital signature scheme with RSA requires you to initially perform decryption using your private-key, m^d (mod N) = 3^101(mod 559) = 542. Then you send the deciphered message 542. The receiver then performs the encryption process by calculating. C^d(mod N) = 542^5(mod 559) = 3.

Signature suites for secure electronic signatures: Due to possible interactions which may influence security of electronic signatures; algorithms and parameters for secure electronic signatures shall be used only in

predefined combinations referred to as the signature suites. A signature suite consists of the following components: signature algorithm with parameters, a key generation algorithm, a padding method and a cryptographic hash function.

Why pad messages?: The message block padding is quite common practice in the implementation of cryptographic algorithms. There are several reasons as to why we do this, the first of which is most likely the most important, security. Since security is the whole reason we have cryptography in the fist place, it only makes sense to use padding to our advantage. It helps us by camouflaging the data inside of the encryption, which, in other words, means that it adds random bytes to our data so that it is more difficult for a prospective attacker to find which bytes are the actual data. Padding also helps us by putting our data in blocks, so that we can operate on pieces of data that are of the same size. This makes our job of cryptography simpler to use in a practical environment. Finally, it provides a standard way to block our data so that we can transport it to other users in a form that they can recognize and use effectively. There are two commonly used types of padding and which you can implement with your own provider, these are OAEP and PKCS1[17].

Stream ciphers: It is important to note that in applications such as the generation of secret keys for symmetric algorithms, which is usually used in conjunction with asymmetric keys, the random data must be random in a very strong sense. For example, it should not be possible to derive any knowledge of generated data from previously generated data, even if the previously generated data is known. This situation may also occur in the context of signatures, e.g., if an authority generates secret keys and an attacker tries to gain information on some of those keys after having obtained some others. Consequently, there must not be any usable link between generated keys of different kinds.

Algorithmic countermeasures to improve security: Some general algorithmic solutions may be used to increase the security if a good source of randomness is not available. For example, consider the DSA signature scheme; the signature algorithm involves a secret key k related to the public verification key g^x mod p and a temporary secret key k that has to be refreshed for each signature. FIPS 186-2[14] says that k may be true or pseudo randomly generated. This means that the values of k do not have to be perfectly independent (note that if the discrete logarithm problem is hard, k is never revealed). The secret key x is generated once and usually outside of the devices such as smart card, so we can assume it has good randomness properties. Consequently, when a bit string is generated from the available source, the following transformations may be used to increase the security: (i) encrypt the bit strings with a stream cipher in order to hide possible repetitions while keeping the same number of available bits; (ii) combine the obtained bits with the secret key x (using a hash function or a block cipher for example) and (iii) other data, such as a counter and/or a smart card unique serial number, can be added to increase security.

A Java Implementation of DSA Scheme: Recall from above that the package java.security provides APIs for the message and digital signatures. It also offers DigestInputStream and DigestOutputStream classes for reading and writing to I/O streams. The signature class provides applications with functionality of the signature algorithms, e.g., SHA-1/DSA while the SignatureSpi class defines the SPIs for the signature.

In the signature class you can generate an object using a getInstance() method. You must supply the algorithm or the algorithm and the provider. A signature object must also be initialized by a private-key using initSign() if it is for signing and by public-key using initVerify() if it is for verification. Further, the signature provides an update() method that you can use to update MessageDigest objects and Signature objects with the data to be digested or signed/verified, respectively. Lastly you can digest the data using the digest() method of the MessageDigest class and you can sign or verify the data using the sign() or verify() method in the Signature class, respectively.

The Signature class manipulates digital signatures using a key produced by the KeyPairGenerator class. The following methods are used in the example below:

- KeyPairGenerator.getInstance("DSA","SUN"); //algorithm and provider supplied
- initialize(1024, r); //initialize KPG with secure random
- generateKeyPair() //Generates the keys.
- Signature.getInstance("SHA1withDSA") //Creates the Signature object.
- initSign(key.getPrivate()) //Initializes the Signature object.
- update(plainText) and sign() //Calculates the signature with a plaintext string.
- initVerify(key.getPublic()) and verify(signature) //Verifies signature.

The following program, Listing 7, shows the implementation for generating keypair, signing a file and then verifying the signature.

Listing 7: Implementation of DSA to sign and verify message (SignVerifyFileDSA.java)

```
package com.rsc.dsasignverify;

import java.io.*;
import java.security.*;
import java.security.spec.*;

class SignVerifyFileDSA
 {
      public static void main(String arg[])
 {
      if (arg.length != 3)
      System.out.println("Usage: java SignVerifyFileDSA DataFile SignatureFile PublicKeyFile");
      else
      try
 {
      FileInputStream fis = new FileInputStream(arg[0]);
      FileInputStream sfis = new FileInputStream(arg[1]);
      FileInputStream pfis = new FileInputStream(arg[2]);

      // We create the keypair-Key strength can be 1024 inside the United States
      KeyPairGenerator KPG = KeyPairGenerator.getInstance("DSA", "SUN");
      SecureRandom r = new SecureRandom();
      KPG.initialize(1024, r);
      KeyPair KP = KPG.generateKeyPair();

      //print out on the command line the provider used
      System.out.println( "\nProvider is: " + KPG.getProvider().getInfo() );

      // We get the generated keys
      PrivateKey priv = KP.getPrivate();
      PublicKey publ = KP.getPublic();

      System.out.println("\nAlgorithm is: " + publ.getAlgorithm() + "\n");

      // We initialize the signature
      Signature dsasig = Signature.getInstance("SHA1withDSA", "SUN");
      dsasig.initSign(priv);

      // We get the file to be signed
      BufferedInputStream bis = new BufferedInputStream(fis);
      byte[] buff = new byte[1024];
      int len;

      // We call the update() method of Signature class->Updates the data to be signed
      while (bis.available() != 0)
  {
      len=bis.read(buff);
      dsasig.update(buff, 0, len);
  }

      // We close the buffered input stream and the file input stream
      bis.close();
      fis.close();

      // We get the signature
      byte[] realsignature = dsasig.sign();

      // We write the signature to a file
      FileOutputStream fos = new FileOutputStream(arg[1]);
      fos.write(realsignature);
      fos.close();

      // We write the public key to a file
      byte[] pkey = publ.getEncoded();
      FileOutputStream keyfos = new FileOutputStream(arg[2]);
      keyfos.write(pkey);
      keyfos.close();
```

```
        //Get the public key of the sender
        byte[] encKey = new byte[pfis.available()];
        pfis.read(encKey);
        pfis.close();

        //Import the encoded public-key bytes
        X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encKey);
        KeyFactory KeyFac = KeyFactory.getInstance("DSA", "SUN");
        PublicKey pubkey = KeyFac.generatePublic(pubKeySpec);

        // Get the signature on the file-This will be verified
        byte[] sigToVerify = new byte[sfis.available()];
        sfis.read(sigToVerify);
        sfis.close();

        // Initialize the signature-update() method used to update the data to be verified
        dsasig.initVerify(pubkey);
        FileInputStream fis1 = new FileInputStream(arg[0]);
        BufferedInputStream buf = new BufferedInputStream(fis1);
        byte[] buff1 = new byte[1024];
        int len1;
        while(buf.available() != 0)
    {
        len1 = buf.read(buff1);
        dsasig.update(buff1, 0, len1);
    }
        buf.close();
        fis.close();

        // Verify the signature
        boolean verifies = dsasig.verify(sigToVerify);
        if (verifies)
        System.out.println("Verified: Valid signature.");
        else
        System.out.println("Warning: Invalid signature.");
    }

        catch (Exception e)
    {
        System.out.println("Caught Exception: " + e);
    }
  }
}
```

The comments embedded in the code, Listing 7, explain what the code does. Notice that we first must write the public-key to file then import the encoded public-key bytes from the file containing the public-key and convert them to a PublicKey. Hence, we read the key bytes, instantiate the DSA publickey using KeyFactory class and generate the public-key from it, i.e.,:

```
//Get the public key of the sender
byte[] encKey = new byte[pfis.available()];
pfis.read(encKey);
pfis.close();

//Import the encoded public-key bytes
X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encKey);
KeyFactory KeyFac = KeyFactory.getInstance("DSA", "SUN");
PublicKey pubkey = KeyFac.generatePublic(pubKeySpec);
```

The X509EncodedKeySpec class represents the Distinguished Encoding Rules (DER) encoding of a public or private key, encoded to the format specified in the X.509 standard[47].

Notice that the names of the three files used in this program should be passed by the user on the command line when executing the program. They are:

1. DataFile - Input data file to be signed.
2. SignatureFile - File where the signature will be written
3. PublicKeyFile - File where the public-key will be written

The program is compiled with following command: javac SignVerifyFileDSA.java

The program is executed in two steps:

Step I-Signs file, creates public key file and verifies the signature: Execute the program by using the Java

interpreter java and passing the names of the three files on the command line, as follows:

```
java SignVerifyFileDSA DataFile SignatureFile PublicKeyFile
```

The program signs file and creates public-key file and verifies the signature and displays the following:

```
Provider is: SUN (DSA key/parameter generation; DSA signing; SHA-1, MD5 digests; SecureRandom; X.509 certificates; JKS keystore; PKIX CertPathValidator; PKIX CertPathBuilder; LDAP, Collection CertStores)
Algorithm is: DSA
Verified: The signature on the file is correct.
```

Here we have used the same data of Listing 3 as our DataFile. Listing 8 and 9 show the derived PublicKeyFile and SignatureFile, respectively on executing our program.

Listing 8: Content of the PublickeyFile (It is all Gibberish to human eye)

```
0,□,0,□,□□*†HÎ8□□0,-
□□□□□ýÇS□□u□)RßJœ.ìäçõ□•R<ïD□Ã-?□¶Q&iE]@"QûY=□Xú¿
Åõ°0õË›Ul×□;□□4oòf˜•k™P¥©ŸŸè□{□"ÂO»©
× þ • Æ □ ø ; W ç Æ ¨ | □ □ □ û ƒ õ Ó Å - Ã □ 5 T □ Z □ ‘ 2 õ u ó ® +
a×*Ïò"□□□ÑH□Ç□□□—`P□□#
Ì²³¹,¢ë,,
ðX□õ□□□□÷á …Ö›=ÞË¼«\6,W˛”¯»ú:ê,ùWL
=□,gQYW□°ÔYOæq□□□□Ì□q#èL(□□•Í 2ŒÈ|á<□z‹T|□(à£®-
+³|u‘n£Ç
ú!5bñûbz□$;Ì¤ñ¾¯Q□‰¨ƒßáZåŸ□’‹f^□{U%d□L;þÎI*□□…□□□□□‘
Œ□ˆœ\iEœH.□4J ª©ú□□●Ï□ê,®Åèh Ão□ÇÆ
```

Listing 9: Content of the SignatureFile

```
0-□□: 0Vƒ□š2o¾kwinEx ½{»□□ —\□³•2Uœ³Ën«
□Ý  YK–□
```

Step II-Tamper with any of the three files: In this step we rerun the program but this time we tamper with any of the three files. To test this, this time we must comment out the sections implementing, write the signature to a file and write the public-key to a file, since we have already done so in Step I, i.e.,:

```
//We write the signature to a file
FileOutputStream fos = new FileOutputStream(arg[1]);
fos.write(realsignature);
fos.close();

//We write the public key to a file
byte[] pkey = publ.getEncoded();
FileOutputStream keyfos = new FileOutputStream(arg[2]);
keyfos.write(pkey);
keyfos.close();
```

And instead we now get the signature file and use the existing public-key to perform the verification procedure.

The output is as expected:

1. If none of the three files have been altered after the signature was applied, the program displays the following

```
Verified: Signature is valid.
```

2. If you change the contents of any of the three files, the program displays the following message:

```
Warning: Invalid Signature.
```

3. If you modify the signature file, so that it no longer respects the signature format, this is the message displayed:

```
Caught Exception: java.security.SignatureException:
invalid encoding for signature
```

The program demonstrate how you can successfully use the Java 2 APIs to send documents with proof of data integrity and authenticity using high quality hash function and DSA scheme.

OTHER POSSIBLE APPLICATION OF DS

Electronic checking: An electronic checking system could be based on signature system such as the above. It is easy to imagine a futuristic encryption device in your home terminal allowing you to sign checks that get sent by electronic mail to the payee[19]. It would only be necessary to include a unique check number in each check so that even if the payee copies the check the bank will only honor the first version it sees. Many other secure transactions requiring message authentication and DSA schemes can be implemented using coprocessor assisted devices like smart cards etc.

CONCLUSIONS

We have shown how to implement message digest using secure hash function, MAC and DSA. We have also shown how one can successfully use the power of Java 2 APIs to send documents with proof of data integrity and authenticity using high quality hash function and DSA scheme. We presume that in future, MAC and DSA schemes can be implemented using hardware/software coprocessor assisted devices like smart cards etc.

REFERENCES

1. The Secure Hash Algorithm Directory MD5, SHA-1 and HMAC Resources: http://www.secure-hash-algorithm-md5-sha-1.co.uk/
2. SHA1 Secure Hash Algorithm-Version 1.0, http://www.w3.org/PICS/DSig/SHA1_1_0.html
3. Comer, D.E., 1999. Computer Network and Internet. Prentice Hall, 2nd Edition.
4. Rivest, R., 1992. RFC 121: The MD5 Message-Digest Algorithm RSA Data Security, Inc.
5. National Institute of Standards and Technology (formerly National Bureau of Standards). FIPS PUB 113: Computer Data Authentication.
6. Gavin, J. and K. McCloghrie, 1993. RFC 1446: Security Protocols of Version 2 of the Simple Network Management Protocol (SNMPv2). Trusted Information System and Hughes LAN Systems.
7. National Institute of Standards and Technology, FIPS PUB 180: Secure Hash Standard (SHS).
8. Mihi Bellare, Joe Killian and Phillip Rogaway, 1990. The Security of Cipher Block Chaining. In: Yvo, G. Desmedt Eds., Advances in Cryptology-Crypto'94, LNCS, Springer-Verlag, New York, pp: 341-358.
9. Damgård, T.B., 1990. A Design Principle for Hash Functions. In: Brassard, G., Ed., Advances in Cryptology. Proceedings of Crypto'89, Lecture Notes in Computer Science. Springer-Verlag, New York, pp: 416-427.
10. Merkle, R., 1990. One Way Hash Function and DES. In: Brassard, G., Ed., Advances in Cryptology: Proceedings of Crypto'89, LNCS, Springer-Verlag, New York, pp: 428-446.
11. RSA Labs. Public-key Cryptographic Standards (PKCS). No. 1, Ver. 2.0, Ver. 2.1.
12. Gene, T., 1992. Message authentication with one-way hash function. ACM Computer Communications Review, 22: 2938.
13. Kaliski, B. and M. Robshaw, 1995. Message Authentication with MD5. Cryptobytes, The Technical Newsletter of RSA Laboratories.
14. Bellare, R. Canetti and H. Kaawczyk, 1998. A modular approach to the design and analysis of authentication and key exchange protocols. Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing 1998, STOC 1998: 419-428. http://theory.lcs.mit.edu/'tcryptol,
15. Mahir Bellare, Roch Guérin and Philip Rogaway, 1995. XOR MACs: New methods of message authentication using block cipher. Accepted to Crpto'95.
16. AIS 20: Application Notes and Interpretation of the Scheme: Functionality classes and evaluation methodology for deterministic random number generators. http://www.bsi.bund.de/aufgaben/ii/zert/jil_ais/ais20e.pdf.
17. Kelsey, J., B. Schneier, D. Wagner and C. Hall, 1998. Cryptanalytic Attacks on Pseudorandom Number Generators, Fast Software Encryption '98, LNCS 1372, pp: 168-188.
18. Blum, M. and S. Micali, 1984. How to generate cryptographically strong sequences of pseudo-random bits. SIAM J. Computing, 4: 850-863.
19. Kefa, R., 2004. Data security and cryptographic techniques-a review. Pak. J. Inform. Technol., 3: 106-132.
20. Rivest, R., A. Shamir and L.M. Adleman, 1983. Cryptographic Communications Systems and Method. U.S. Patent 4,404,829,20 Sept.
21. National Institute of Standards and Technology, 1995. NIST: FIPS Publication 180-1: Secure Hash Standard (SHS-1).
22. International Organization for Standardization, 1998. ISO/IEC 10118-3: Information technology-Security techniques-Hash functions-Part 3: Dedicated hash functions.
23. SECOID tree structure, http://www.darmstadt.gmd.de/secude/Doc/htm/oidgraph.htm
24. Dobbertin, H., A. Bosselaers and B. Preneel, 1996. RIPEMD-160: A strengthened version of RIPEMD, in Fast software encryption. Proceedings of the Third International Workshop, Cambridge, UK, February 21-23, 1996, Gollmann, D. (Ed.), LNCS 1039, Springer-Verlag, pp: 71-82,
25. Menezes, A., P. van Oorschot and S. Vanstone, 1997. Handbook of Applied Cryptography (Chapter 5), CRC Press, http://www.cacr.math.uwaterloo.ca/hac.
26. Christoffersson, P., S.A. Ekahll, V. Fak, S. Herda, P. Mattila, W. Price and H.O. Widman, 1998. Crypto Users Hand Book, A Guide for Implementers of Cryptographic Protection in Computer Systems. North Holland: Elsevier Science Publishers.
27. Rivest, R., A. Shamir and L. Adleman, 1978. A method for obtaining digital signatures and public key cryptosystems. Communications of the ACM., 21: 120-126.
28. Rivest, R., A. Shamir and L. Adleman, 1978. A method for obtaining digital signatures and public key cryptosystems. Comm. ACM., 21: 120-126.
29. Rabin, M.O., 1979. Digital Signature and Public-Key Functions as Intractable as Factorization, MIT Laboratory of Computer Science, Technical Report, MIT/LCS/TR-212.

30. Preneel, 1993. Analysis and design of cryptographic hash functions. Ph.D. Thesis, Katholieke University Leuven.
31. Diffie, W. and M.E. Hellman, 1996. New directions in cryptography. IEEE Transaction on Information Theory, 22: 644-654.
32. ElGamal, T., 1985. A public-key cryptosystem and a signature scheme based on discrete logarithms. IEEE Trans. Info. Theory, 31: 469-472.
33. FIPS 186, 1993. National Institute of Standards and Technology, Digital Signature Standard, FIPS PUB 186.
34. Schnorr, C., 1991. Efficient Signature generation by smart cards. J. Cryptol., 4: 161-174.
35. Nyberg, K. and Rueppel, 1996. Message recover for signature schemes based on the discrete logarithm proble designs. Codes and Cryptography, 7: 61-81.
36. Rabah, K., 2005. Theory and implementation of elliptic curve cryptography. J. Applied Sci., 5: 604-633.
37. Rivest, R.L., 1991. Finding Four Million Random Primes. In: Advances in Cryptology-Crypto '90, Menezes, A.J. (Ed.), LNCS 537, Springer-Verlag, pp: 625-626.
38. National Institute of Standards and Technology, 2000. NIST: FIPS Publication 186-2: Digital Signature Standard (DSS).
39. Algorithms and Parameters for Secure Electronic Signatures (ALGO), Directive 1999/93/EC of the European Parliament and of the Council of 13 December 1999 on a Community Framework for Electronic Signatures.
40. Menezes, A., P. Van Oorschot and S. Vanstone, Handbook of Applied Cryptography. CRC Press, 1997.
41. Heileman, G.L., 1996. Data Structures, Algorithms and Object Oriented Programing. MGraw-Hill International Edition, New York.
42. JCA, Java Cryptography Architecture, http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html
43. JCE, Java Cryptography Extension (JCE), http://java.sun.com/products/jce/doc/guide/API_users_guide.html
44. IBM-JCE: http://www-106.ibm.com/developerworks/eserver/articles/java_crypto.html
45. RSA-JCE: RSA BSAFE Crypto-J, RSA Security, Inc. http://www.rsasecurity.com/node.asp?id=1202
46. BC-Bouncy Castle, www.bouncycastle.org
47. Housley, R. *et al*., 1999. Internet X.509 Public Key Infrastructure. Certificate and CRL Profile, Internet Request for Comment (RFC) 2459.