Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 6 (3): 482-510, 2006
ISSN 1812-5654

Implementing Secure RSA Cryptosystems Using Your Own Cryptographic JCE Provider

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

Abstract: Today and without a doubt, we live in a net-centric world in which new information technologies arrive at lightning speed, allowing us to share information locally to globally faster than ever before. The bad guys are equally out there to eavesdrop on our data transactions over the insecure communication channels. Cryptography can be used to help us secure our data communication. In this study we describe a Java implementation of secure RSA cryptosystems using your cryptographic provider, Java Cryptographic Extension (JCE). We then illustrate the use of this implementation in a working prototype. We then explain the naïve procedure for implementation of RSA cryptosystems and signature algorithm followed by the real time Java implementation using Open Source Bouncy Castle JCE provider.

Key words: Cryptography, RSA, DSA, DH, ECC, DLP, ElGamal, public and private keys, digital signature, confidentiality, authentication, integrity, Java, JCE, JCA, SSL

INTRODUCTION

Cryptography can be used to provide: Confidentiality which ensure data is read only by authorized parties, Data integrity which ensure data wasn't altered between sender and recipient and Authentication which ensure data originated from a particular party. There are two basic types of cryptographic systems: symmetric (private-key) and asymmetric (public-key)[1]. The symmetric keys include DES, Blowfish, AES etc.[2]. Symmetric cryptography algorithms are typically fast and are suitable for processing large streams of data. The disadvantage of symmetric cryptography is that it presumes two parties have agreed on a key and been able to exchange that key in a secure manner prior to communication. This is a significant challenge. Since in real practice, a private conversation is a common occurrence in the world of business, however and it is unrealistic to expect initial business contacts to be postponed long enough for keys to be transmitted by some secure physical means. The cost and delay imposed by this key distribution problem is a major barrier to the transfer of business communications to large teleprocessing networks. Public key crypto-algorithms offer different approach to eliminate the need for a secure key distribution channel. Hence, in real practice symmetric algorithms are usually mixed with public key algorithms to obtain a blend of key distribution, crypto-network security and speed.

PUBLIC-KEY PROTOCOL

In the public-key (or asymmetric) protocols we have two keys-a secret key that must be kept from unauthorized users and a public key that can be made public to anyone. Both the public-key and the private-key are mathematically linked; only the associated private-key can decrypt data encrypted with the public-key and data signed with the private-key can only be verified with the associated public-key. Both keys are unique to the communication session. Public-key algorithms cannot be used to chain data together into streams like symmetric key algorithms can. Different public-key crypto-systems are used to provide public-key security. Among these we can mention the RSA[3], Diffie-Hellman (DH)[4], Digital Signature Algorithm (DSA)[5-7], ElGamal cryptosystem[8] and in recent years the Elliptic Curve Cryptography (ECC)[9-11]. These systems provide these services by relying on the difficulty of different classical mathematical problems, hence provide the services in different ways. In this work we will concentrate on RSA protocol.

The public-key cryptosystems compared to symmetric ones provide arbitrary high levels of security and do not require an initial private-key exchange between two communicating parties. Because of this feature, these cryptosystems are considered to be indispensable for secure communication and authentication over open (insecure) networks[12]. It is designed to be computationally intractable to calculate a

private-key from its associated public-key; that is, it is believed that any attempt to compute it will fail even when up-to-date technology and equipment are used[5]. In real applications, both symmetric and asymmetric protocols are used. The public-key algorithm is first used for establishing a common symmetric-key over insecure channel. Then the symmetric cryptosystem is used for secure communication with high throughput. Due to the comparative slowness of the public-key algorithms, dedicated hardware is sometimes desirable for efficient implementation and operation of the cryptographic systems.

The mechanics of modern public-key cryptosystems: One should note that most of the cryptosystems described here and the remainder of the article are crypto-primitive in nature. In cryptography the term primitive means a basic ingredient in a cryptosystems. In practice, one generally has to modify and combine these primitives in a careful way so as to simultaneously achieve various objectives related to efficiency and strong notions of crypto-security. For example, modern asymmetric key cryptography uses mathematical operations that are fairly easy to do in one direction, but extremely hard to do in reverse. That is, a one-to-one function f: $X \rightarrow Y$ is one-way if it is easy to compute f(x) for any $x \in X$ but hard to compute f^{-1} (y) for most randomly selected y in the range of f. Therefore, the goal in designing a cryptosystem is to make the enciphering and deciphering operations inexpensive, but also to ensure that any successful cryptanalytic operation is too complex to be economical. In this respect, we will call a task computationally infeasible if its cost as measured by either the amount of memory used or the runtime if finite by impossibly large. The standard example used (indeed, the one that is almost synonymous with public key encryption) is that of prime factorization. Large primes have at least one practical application-they can be used to construct public key cryptosystems (also known as asymmetric cryptosystems and open encryption key cryptosystems)[1]. Two basic types of public-key schemes emerged in the mid 1970s; Diffie-Hellman (DH) for key agreement protocol proposed in 1975 which relies on the hardness of the discrete logarithm problem (DLP): given p, g and g^a or $g^a = y \pmod p$, find a, i.e., $a = \log_g y$[4]. In general, if the modulus of discrete logarithm problem (DLP) is replaced with a product of two primes (i.e., $n(= p \cdot q)$), then finding the solution becomes naturally infeasible for large moduli, simply because the factorization of large integer number is infeasible. This is the backbone on which the security of public key like RSA cryptosystems developed two year later after DH, relies on[1,3]. The RSA takes it security from the hardness of the integer factorization problem (IFP), that is: given two large prime numbers p and q, it is a straightforward task to multiply them together and find the resulting multiplicand, $n(= p \cdot q)$. However, given a large composite integer that is a product of two large prime factors, it is extremely difficult to find those two primes[3].

THE RSA CRYPTOGRAPHIC SYSTEMS

In 1977, three professors in the computer science lab at MIT - Ronald Rivest, Adi Shamir and Len Adelman - while working on developing a practical multi-user cryptographic system, based on Diffie-Hellman (DH)[3] proposition, they realized a basic fact: It is very easy to multiply two prime numbers to get a large composite number, but it is difficult to take that composite number and find its prime number components. The outcome of this research is the technique simply referred to by the initials of its three inventors: RSA[3]. RSA algorithm makes use of any publicly available key to encrypt the information, but only the person who holds the matching secret-key can do the decryption. The RSA technique is one of the most powerful encryption tool known to-date. It is also used as the public-key system in PGP (Pretty Good Privacy)[13]. RSA can also be used as a key transport and digital signature system as will be seen later. The most widespread current use of RSA algorithm is in the Secure Sockets Layer (SSL) protocol for data protection on the Internet. We can also use RSA to verify the integrity of our data through electronic signatures and electronic certificates.

RSA Crypto-Algorithm

Generation of RSA modulus: An RSA modulus N (= pq) is obtained by multiplying two prime numbers, p and q, of roughly the same size. Furthermore, the two factors must not be too close in order to be far enough from the square root of the modulus.

If we let p and q be the two prime factors of the modulus N, we can require that, for example, $0.5 < |\log_2(p) - \log_2(q)| < 30$. This implies that: $\log_2(N)/2 - 15 < \log_2(q) < \log_2(N)/2 + 15$. The generation of an RSA modulus of exactly k-bits could be done with the following algorithm[1]:

- Choose a random prime number p in the range $[2^{kk/2-1}, 2^{k/2+15}]$;
- Choose a random prime number q in the range $[2^{k-1}/p, 2^k/p]$;

- If the condition $0.5<|\log_2(p)-\log_2(q)|<30$ is not satisfied, go back to the first step;
- Let N be the product of p and q.

A more complicated method that avoids the third step altogether but produces differently distributed primes is:

- Choose a random prime number p in the range $[2^{k/2-1/4}, 2^{k/2+15}]$;
- Choose a random prime number q in the range[a,b] where a = $\max(2^{k-1}/p, p\cdot 2^{-30})$ and b = min $(2k/p, p\cdot 2^{-1/2})$;
- Let N be the product of p and q.

Accordingly, one may note that it is easy to find random large primes by choosing random integers and performing test on them-either very efficiently strong primality tests or the deterministic polynomial time primality test discovered by Agrawal *et al.*[14].

RSA Key Generation: Generating cryptographic keys is extremely important. If the security of a cryptographic system is reliant on the security of keys then clearly care has to be taken when generating keys. Cryptographic keys need to be as random as possible so that it is infeasible to reproduce them or predict them. Therefore, a trusted random number generator is essential[7].

Let e and d be two integers satisfying $(e)(d) = 1 (\mod\varphi(N))$, where $\varphi(N) = m = (p-1)(q-1) = N+1-(p+1)$ is the Euler φ-function of N, equal to the 1<N that are relatively prime to N. These integers N,e,d are called, respectively the RSA modulus, the encryption exponent. Two pairs of numbers are important. The number N = (p)(q), known as the RSA modulus, is a component of both pairs. The first pair is (N,d), which is known as the RSA private-key. The second pair is (N,e), which is known as the RSA public-key. The number d is the RSA private exponent. The number e is the RSA public exponent. You publish or otherwise make known the public-key. You keep the private-key (in particular d) and the original prime numbers (p and q) securely secret. Typically, one sends the public-key (N,e) to a certificate authority (CA) to obtain a certificate for it.

RSA encryption/decryption mechanics: The RSA encryption algorithm begins with the random and independent selection of two large primes p and q. For simplicity, let's take p = 773 and q = 557. The next step is to calculate N = (773)(557) = 430561. Next an integer e, also known as encryption key or public exponent, is selected between 3 and N-1 (inclusive) that has no factors in common with (p-1)(q-1). In this case, m = (773-1)(557-1) = 429232. The number 429232 can be prime factored as: $2^4\cdot 139\cdot 193$. So e must not be a multiple of 2,139 or 193. The algorithm does not require you to choose a prime number for e. For example 83 or 95 would do. To keep the numbers small and simple, let's choose e = 5. (Note that public exponent may be chosen as small as e = 3).

Next, the algorithm requires an integer d, also known as decryption key, such that, $(d)(e) = 1(\mod m)$. The optimization that consists of first choosing the secret exponent d and then computing the public exponent e can be used but in this case d must be randomly chosen. The value of d should not be too small otherwise there are known attacks which can factor the modulus[15,16]. A conservative method would be to choose d randomly in a range at least $N^{1/2}$ from its minimum and maximum values. In practice, choosing d uniformly in the range [3,N] has negligible probability of producing an exploitable value.

For security reason d must never be much smaller than N. Wiener[17] showed that if d has fewer than n/4, more precisely, $d<1/3\ N^{1/4}$, then an unauthorized person knowing only the public key can efficiently compute d. Thus, according to Boneh and Durfee[15], the communicating partners should always choose d with more than n/2 bits; preferably, d should have n-bits. Alternatively, Bob can probably get away with choosing his public exponent e (which is used for encryption of messages and also to verify signatures) to be small. In fact, most implementations of RSA use e = 3 or $e = 12^{16}+1 = 65537$. But Håstad[18] found a flaw when e is small. This difficulty can be avoided by padding messages[7], that is, by inserting a short sequence of random symbols in message units before sending them (in which case the recipient can easily delete the added symbols before reading the text). Of course, a different random sequence must be inserted each time Bob sends a message.

Furthermore, for security reason and secure communication, let us also remember that a new modulus N must be used for each user of the signature scheme. A modulus must not be shared by same users, even if different public exponents are used. Furthermore, notice that if the RSA keys are generated as explained above, the probability of generating two keys with the same modulus or the same secret exponent is totally negligible, even if many keys are computed.

The Java programmer's equivalent of e = 5 is the remainder operator. In this example, e = 5 and $\varphi(N) = m = 429232$, so the number d requires that $5d = 1 (\mod m)$. One way to find this number is to look at one more than the multiples of 429232 (since 5d =429232k+1, where k is an integer in the range [0, N-1]), then check for the first one that is divisible by 5. In other words, you would check 429232, 8584465 and so on (or simply compute $d = e^{-1}$

(mod m) = 5^{-1} (mod 429232) = 171693). The number 858465 is divisible by 5 (i.e., 171693 times 5 is 858465), so d = 171693.

Recall that in real practice, public-key encryption schemes are many times slower than their symmetric-key counterparts. Moreover, with public key crypto-schemes only a small block size can be processed, typically 8 or 16 bytes. However, asymmetric cryptographic schemes are considered much more flexible. Thus, RSA is typically used either to encrypt a short message (such as a credit card number) or else to encrypt a randomly chosen key k, which in turn is used with a symmetric-key encryption scheme, which uses variable length buffer such as DES, 3DES or Advance Encryption Standard (AES) to encrypt the message itself. The key k is usually quite short (e.g., 128, 192, or 256 bits for the AES) and can therefore be regarded as an integer M in the interval [0, N-1][2].

RSA key lengths: When thinking about key lengths it is all too easy to think the bigger, the better. While a large key will indeed be more difficult to break under most circumstances, the additional overhead in encrypting and decrypting data with large keys may have significant effects on the system. The key needs to be large enough to provide what is referred to as cover time. Cover time is the time the key needs to protect the data. If, for example, you need to send time critical data across the Internet that will be acted upon or rejected with a small time window of, say, a few minutes, even small keys will be able to adequately protect the data. There is little point in protecting data with a key that may take 250 years to be broken, when in reality if the data were decrypted and used it would be out of date and not be accepted by the system anyhow. A good source of current appropriate key lengths can be found[19].

Relative Crypto-Key Sizes: So what does this mean in practice? NIST has recommended that 128-bit protection is necessary to achieve relatively lasting security (to the year 2036 and beyond) for the symmetric key crypto-schemes. This means moving from 3DES to AES[2]. To avoid compromising the security of the system, NIST's FIPS 140-2 standard[20] indicates that keys for symmetric ciphers such as AES must be matched in strength by public-key algorithms such as RSA and ECC. For example, a 128-bit AES (or Rijndael) key demands an RSA key size of 3,072-bits for equivalent security but for the same strength, the ECC key size is only 256-bits.

As you can observe from Table 1, while ECC (or AES) key sizes scale linearly, RSA does not. The result is that the gap between systems grows as the key sizes increase.

Table 1: NIST guidelines for public-key sizes with equivalent security levels[19]

Security (Bits)	Symmetric encryption algorithms	Minimum size (Bits) of public keys			
		DSA/DH	RSA	ECC	RSA/ECC
80	Skipjack	1024	1024	160	6:1
112	3DES	2048	2048	224	9:1
128	AES-128	3072	3072	256	12:1
192	AES-192	7680	7680	384	20:1
256	AES-256	15360	15360	512	30:1

This is especially relevant to implementations of AES where at 256-bit security you need an RSA key size of 15,360-bits compared to 512-bits for ECC. This will have a significant impact on a communication system as the relative computational performance advantage of ECC versus RSA is not indicated by the key sizes but by the cube of the key sizes. The difference becomes even more dramatic as the greater increase in RSA key sizes leads to an even greater increase in computational cost. So going from 1024-bit RSA key to 3072-bit RSA key requires about 27 times (3^3) as much computation while ECC would only increase the computational cost by just over 4 times (1.6^3).

Security of RSA: In RSA, public and private keys denoted by {e, N} and {d, N} respectively, are generated by relying on the hardness of Integer Factorization Problem (IFP)[5,15] which, generally stated, is: Given two large prime numbers p and q, it is a straightforward task to multiply them together and find the resulting multiplicand, N(= p·q). However, given a large composite integer that is a product of two large prime factors, it is extremely difficult to find those two primes[5]. These primes are then used to generate the associated private and public, respectively. The most common way of trying to break this system, is by trying to factor N (from the public-key) to obtain the two primes from which the private-key can be determined[21]. This is very costly. Unfortunately, encrypting a message M, involves exponentiation,$C = M^e$ (mod N), which involves a lot of computation.

FACTORIZATION ATTACK ON RSA

Factorization was once primarily of academic interest. It gained in practical importance after the introduction of the RSA public-key cryptosystem in 1977[1,4-6], which takes it security from the hardness of the Integer Factorization Problem (IFP). The most basic attack on RSA consists of factoring the modulus N = pq. Factoring N would enable an enemy cryptanalysis (Eve) to break RSA crypto-method. The factors of N enable her to compute $\varphi(N) = \varphi(p)\varphi(q) = (p-1)(q-1)$ and thus d. Fortunately, factoring a number seems to be much more

difficult than determining whether it is prime or composite. A large number of factoring algorithms exist. Knuth[22], gives an excellent presentation of many of them.

Let N be an n-bit integer. Most of the subexponential-time algorithms-those that take fewer than 2^{n^c} -steps with c<1 - are of index calculus type. The index calculus method is based on the elementary observations that if $x^2 \equiv y^2 \pmod N$, then $N = pq \mid (x+y)(x-y)$ and so p and q each must divide either x+y or x-y. If x and y were formed independently of one another then one expects that 50% of the time the two primes will divide different factors, say $p \mid x+y$ or $q \mid x-y$. In that case we can factor N by using the Euclidean algorithm to compute gcd(x+y, N) = p.

We start the index calculus factoring algorithm by choosing a factor base F consisting of all primes less than some bound B along with the number -1: F = $\{p_0, p_1,..., p_r\}$, where, $p_0 = -1$, $p_1 = 2$, $p_2 = 3$,.... We next choose positive integers a<N (either randomly or according to some convenient criteria) and compute the least absolute residue of a^2. If this residue cannot be written as a product of numbers in our factor base, we choose another value of a. We finally arrive at a system of modN relations of the form:

$$a_i^2 \equiv \prod_{j=0}^{r} p_j^{a_{i,j}}$$

where, i = 1,2,..., s. We try the product :

$$\prod_i a_i^{2\nu_i} \equiv \prod_{i,j} p_j^{\nu_i \alpha_{i,j}}$$

where, $\nu_i \in \{0,1\}$ in such a way that we get a perfect square on the right. In other words, we need each prime on the right to occur to an even power; that is $\sum_i \nu_i \alpha_{i,j}$ must be even for each j = 0, 1..., t. This amounts to solving a system of r+1 simultaneously equations in s unknowns over the field $F_2 = \{0,1\}$. Once we have such a product, we can set:

$$x = \prod_i a_i^{\nu_i} \text{ and } y = \prod_i p_i^{\mu_i} \text{ where } \mu_j = \tfrac{1}{2}\left(\sum_i \nu_i \alpha_{i,j}\right)$$

Then $x^2 \equiv y^2 \pmod N$ and there is a 50% chance that we can immediately factor N. Alternatively, we may find another solution to the simultaneous equations over F_2 and try again.

As an example, let N = 403 = (13·31) and choose F = {-1, 2, 3, 5, 7, 11, 13}. After squaring some 2-digit numbers, we find that we can take a_i, $1 \le i \le 7$ and present the factors as shown in the table below:

i	a_i^2	a_i^2 mod N	-1	2	3	5	7	11	13
1	19^2	$-1\cdot2\cdot3\cdot7$	1	1	1	1	0	0	0
2	22^2	3^4	0	0	0	0	0	0	0
3	26^2	$-1\cdot2\cdot5\cdot13$	1	1	0	1	0	0	1
4	28^2	$-1\cdot2\cdot11$	1	1	0	0	0	1	0
5	33^2	$-1\cdot2^3\cdot3\cdot5$	1	1	1	1	0	0	0
6	34^2	2^4	0	0	0	0	0	0	0
7	38^2	$-1\cdot2^3\cdot3\cdot7$	1	1	1	0	1	0	0

Next we extract the exponent information from $\prod_i p_i^{\mu_i}$, from the left of the table to form a matrix A, which we transpose to get A^T,

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{A}^T = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

and then solve the $A^T \cdot \upsilon = 0 \pmod 2$, where υ is the exponent vector, which gives three set of possible solutions as:

$$\nu = \{ [0,1,0,0,0,0,0], [0,0,0,0,0,1,0], [1,0,0,0,1,0,0] \}$$

which we shall identify as follows: $\upsilon = \{[\upsilon_{s1}], [\upsilon_{s2}], [\upsilon_{s3}]\}$. Here υ_{s2} and υ_{s3} give no factors of N = 403, while υ_{s1} is able to factor our composite number N. For example, υ_{s1} = [0, 1, 0, 0, 0, 0, 0], gives, $x \equiv 22 \pmod{403}$ and $y \equiv 3^2 \pmod{403}$. These yield the gcds, gcd(x-y, N) = 13 and gcd (x+y, N) = 31, which gives a factorization of N = 403 = 13·31.

It can be shown that the time required to factor an n-bit integer by the above index calculus factorization

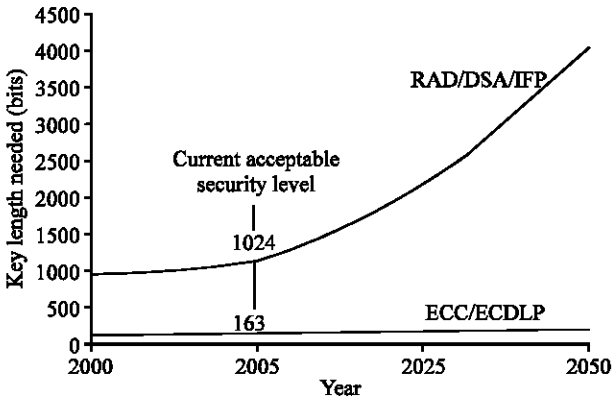

Fig. 1: Proposed the minimum key sizes (in bits) to be regarded as safe for RSA and ECC

method is of the order $2^{n^{1/2+\epsilon}}$ for any $\epsilon>0$ (More precisely, the number of steps is $\exp(O(\sqrt{n \log n}))$.) Throughout the 1980's modifications and generalizations were introduced that improved upon the performance of index calculus methods; however, no one was able to reduce the exponent of n below $1/2+\epsilon$. Even when Lenstra[23] developed an exciting and conceptually very different factorization methods based on elliptic curves, asymptotically his method required roughly the same amount of time as the index calculus algorithms. Some people wondered whether the exponent $1/2+\epsilon$ might be best possible for a general integer factorization algorithm.

However, in the 1990's ideas of Pollard[24] led to a major breakthrough in factorization, called the number field sieve (NFS). By carrying over index calculus to algebraic field, it was possible to factor an arbitrary n-bit integer in time bounded by $2^{n^{1/3+\epsilon}}$ for any $\epsilon>0$. (More precisely, $\exp(O\sqrt[3]{n \log^2 n})$). The number field sieve is at present the fastest method for factoring an RSA modulus; the current record is a number of 248 decimal digits.

The reduction of the exponent n from $1/2+\epsilon$ to $1/3+\epsilon$ has important consequences in the long run. It means that even modest improvements in hardware and software can increase the size of the numbers that can be factored. For this reason the current recommendation for implementation of RSA is to use numbers of at least n = 1024 bits, (Fig. 1).

SECURE IMPLEMENTATION OF RSA

One may note that the implementation RSA algorithm is secure only if the factorization of the carefully chosen sufficiently large two prime numbers requires a super-polynomial amount of time with respect to the size of the number. To date it has not been proved that the process of factorization of numbers requires an exponential amount of time. However, no classical polynomial time algorithm has been found and most researchers generally believe that none will ever been found! This is a practical motivation for the current interest in integer factorization algorithms. Currently, you need a crypto-keylength of 1024-bit number to get the same security you got from a 512-bit number in the early 1980s. If you want your keys to remain secure for 20 years, 1024 bits is probably too short (Fig. 1).

Because the security of RSA is so dependent on an adversary's inability to factor a large composite number, much more research has been done to find ways to quickly factor such numbers as noted earlier. The key question is: how large is sufficiently large to make this recovery virtually impossible? In the 1980s it was generally held that prime numbers of a fifty odd digits (i.e., 10^{50}) would suffice. As a case in point take, for example, when Rivest challenged the world in 1977 to factor RSA-129, a 129-digit number (from a special list), he estimated that on the basis of contemporary computational methods and computer systems of the day, this would take about 10^{16} years of the computing time. Seventeen years later it took only eight months in a worldwide cooperative effort to do the job[25]. This gave a false credence to a modulus number of 512-bit (155-digits) which in the mid 80s was a popular RSA encryption keylength used, for example, on the Internet and to secure transactions in the financial world-it too, was factored at CWI on August 22, 1999 using the Number Field Sieve factoring method (NFS)[26]. The current record in factoring a generally hard integer is that of the 200 decimal digits challenge integer from RSA Data Security, Inc., RSA-200, which was accomplished with general number field sieve (GNFS) was factored on May 9, 2005 by Bahr, Boehm, Franke and Kleinjung[27]. Among the Cunningham integers, the record is the factorization of 248 decimal digit integer by special number field sieve (SNFS) was factored by Aoki, Kida, Shimoyama, Sonoda and Ueda (CRYPTREC) on April 04, 2004[28,29].

Therefore, the baseline and trade-off, is the size of N should be chosen such that the time and cost for performing the factorization exceeds the value of the secured/encrypted information. But even then, great care must still be taken in the overall crypto-design, as current development in integer factorization have gone much faster than foreseen and it is a precarious matter to venture upon quantitative forecasts in this field. Moreover, one should realize that it always remains

possible that a new computational method could be invented from unsuspecting quarter, which makes factoring 'easy' (e.g., quantum computing[30,31], if an operative quantum computer were to be realized in the not-so distance future)-fortunately or unfortunately depending on which side you are on-no one knows how to build one yet!

A recent research trend has been to design special-purpose hardware on which factoring algorithm such as the number field sieve might be faster or more cost-effective than on conventional general-purpose computers. Among the noteworthy proposals are Shamir's TWINKLE machine[32], Bernstein's circuits[33] and the TWIRL machine of Shamir and Tromer[34]. Shamir and Tromer[34] estimate that the relation-generation stage of the number field sieve for factoring a 1024-bit RSA modulus can be completed in less than a year by a machine that would cost about US$ 10 million to build and that the linear algebra stage is easier. Such special-purpose hardware has yet to be built (unless it has been built in secret), so it remains to be seen if this study will have any impact on the size of RSA moduli used in practice.

So, be warned that factoring large numbers is hard but not as hard as it used to be. This has grave implications for the effectiveness of public-key cryptography, which relies on the difficulty of factoring long keys for its security. Today, the wise cryptographer is ultraconservative when choosing key lengths for a public-key system. You must consider the intended security, the key's expected lifetime and the current state of the factoring art. Moreover, the Number Field Sieve (NFS) is currently at the cutting-edge of research into integer factoring algorithm capable of factoring such large composite numbers over 200 digits. It is recommended that N be 300 digits long. Longer or shorter lengths can be used depending on the relative importance of encryption speed and security in the application at hand. A 200-digit N is considered moderate security against an attack using current technology; using over 300 digits provides a margin of safety against future developments. The flexibility to choose a keylength (and the level of security) to suit a particular application is a feature not found in many cryptographic schemes.

Computing power is measured in MIPS-years: A million-instructions-per-second computer running for one year or about 3×10^{13} instructions. A 100-MHz Pentium III is about a 50-MIPS machine; a 1600-node Intel Paragon is about 50,000 MIPS. In 1983, a Cray X-MP supercomputer factored a 71-digit number in 0.1 MIPS-years, using 9.5 CPU hours. That's expensive. Factoring the 129-digit number in 1994 required 5000 MIPS-years and used the idle time on 1600 computers around the world over an eight-month period. Although it took longer, it was essentially free. These two computations used what's called the quadratic sieve. The number field sieve is faster and more powerful than the quadratic sieve for numbers well over 100 digits and can factor a 512-bit number over 10 times faster-it would take less than a year to run on an 1800-node Intel Paragon. Figure 2 shows the current security level involving MIPS-years estimation featuring public keys.

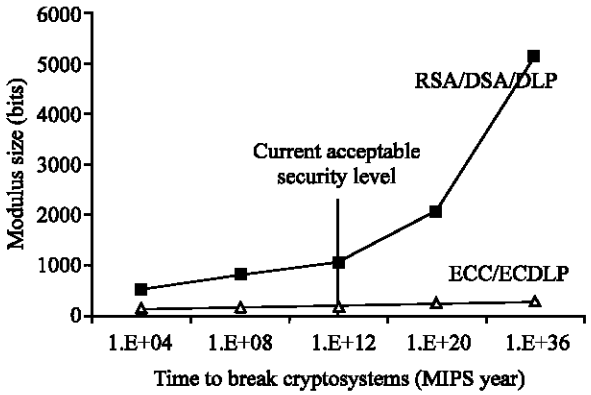

Fig. 2: Comparison of security levels of ECC and RSA/DSA/DLP

CRYPTOGRAPHIC TOOLKITS AND LIBRARIES IN JAVA™

A java implementation of cryptography algorithm: There are several questions that we want to answer in completing this part of the work:

- How can we use Java to perform encryption of data and what limitations are there?
- Theoretically, encryption algorithms operate on numbers, but how can Java handle data that is not numeric in nature, i.e. bytes?
- How does a program know where to find an algorithm?
- When we implement a provider in Java, where in the file system is the best place to put it?
- How do we encrypt large amounts of data from files?

Fortunately for us, the Java programming language includes the Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE), which include many algorithms to help us. The JCA and JCE also allow us to add our own algorithms or a third party algorithm, which, in the future, may be even more secure and efficient than

those we have today. Throughout this study, we will examine the JCA and JCE and show how to utilize them, including how to implement one's own cryptographic provider using the RSA encryption and signature algorithm.

Java Cryptography Architecture (JCA): The Java security has been evolving continuously allowing for adjust to new crypto-schemes issues. The JCA is a framework that provides cryptographic functionality development capabilities for a Java platform. It was introduced early in Java's evolution as an add-on package. The first release of the Security API (Application Programming Interface) was an extension of JCA API's for encryption, key exchange and coding message authentication. Prior to J2SDK 1.4, JCE was optional, in part due to US Government export restrictions on strong encryption crypto-algorithms. The Java Secure Socket Extension (JSSE) and Java Authentication and Authorization Service (JAAS) security features have also been integrated into J2SDK, starting from version 1.4. Two new security features have been introduced: Java GSS-API (Java Generic Security Services Application Programming Interface) that can be used for securely exchanging messages between communicating applications e.g., using the Kerberos V5 mechanism and Java Certification Path API that includes classes and methods in the java.security.cert package. These allow the designers and developers to build and validate certificate chain with ease and without the baggage of having to know the underlying implementation of a particular security mechanism.

Java cryptographic service provider: A provider is the underlying implementation of a particular security mechanism. The JCA includes the provider architecture that has two main design principles: First, is independence from implementation and interoperability, which derives from using the services without knowing their implementation details. Second, is algorithm independence and extensibility; meaning that new service providers and/or algorithms can be added without affecting existing providers.. The notion of Cryptography Service Provider (CSP), or just provider, has been introduced in JCA. The provider architecture allows for multiple and interoperable cryptographic implementations. An application developer can create or specify his/her own CSP. The Service Provider Interface (SPI) presents a single interface for implementer. Classes, methods and properties are accessible to applications through JCA-API. The SPI allows a CSP to plug in implementations for java applications. A provider can be

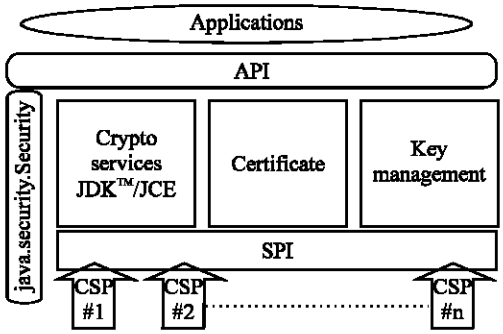

Fig. 3: Java cryptography architecture

used to implement any security service. Several providers can be available and they may or may not provide similar crypto-services and algorithms. Figure 3 depicts the layers of JCA[35]. There are several providers available today, some of which are freely available and others that are quite costly. Companies that offer providers include IBM[36], Bouncy Castle[37] and RSA[38]. Later in this study, we'll examine the RSA implementation from Bouncy Castle. Sun[35] provides details on how to implement these providers.

THE JCE AND JSSE PROVIDERS

JCE [http://java.sun.com/products/jce/] and JSSE [http://java.sun.com/products/jsse/]-Now an integral core part of JDK 1.4, the Java Cryptography Extensions (JCE) and the Java Secure Socket Extensions (JSSE) are a natural choice if you are developing in Java. The JCE Application Programming Interface (API) consists of numerous classes and interfaces for working with several kinds of algorithms and security features. The JCE provides a framework and implementations for encryption, key generation, key-agreement and, message authentication code (MAC) schemes. Support for encryption includes symmetric, asymmetric, block and stream ciphers. The software also supports secure streams and sealed objects. Unfortunately, SunJCE does not support the implementation of RSA public-key algorithm due to US export restrictions. Hence, one must implement a third party JCE provider. In this article we will explore the installation and utilization of JCE Extensions with the Java 2 platform, Standard Edition (J2SE). We will also cover how to use Sun's and third party security providers and demonstrate how to create a key and a cipher and how to perform basic data encryption and decryption and message authentication and signature algorithm. To be able to support all the necessary crypto-provider schemes, we will implement crypto-provider scheme from BouncyCastle.

The Legion of the Bouncy Castle [http://www.bouncycastle.org] - An open source clean-room implementation of the Java Cryptography Extensions (JCE), produces a first rate Java cryptographic library for both JSSE and J2ME. Javasoft cannot provide some of its international customers with an implementation of the JCE because of US export restrictions. BouncyCastle JCE is being developed by a team of dedicated Australian volunteers to address this problem. BouncyCastle JCE is a complete clean-room implementation of the official JCE 1.2 API as published by Sun. BouncyCastle also produces a PGP library for those developers needing to integrate Java applications with PGP systems. To date, Bouncy Castle supports over 20 engines. This sounds good, as there is a great deal of flexibility as to how data can be encrypted.

Implementation of JCA and JCE: The Java Cryptography Architecture (JCA) is included in the Java 2 run-time environment distributed by Sun under open source license. Which means you can download it and use it freely. It includes algorithms to perform message digests, create digital signatures and generate key pairs. There are no algorithms to perform ciphers, which is due to the fact that when the JCA was first released, export restrictions would not allow Sun to distribute such algorithms. The classes included in the JCA are available in java.security package, including: MessageDigest, Digital Signature, KeyPairGenerator and SecureRandom.

The lack of cipher algorithms led to the release of the Java Cryptographic Extension (JCE), which includes the encryption and decryption cipher algorithms. It also includes algorithms to generate single keys and secret keys. The classes included in the JCE can be found in the javax.crypto package, including: CipherSuits, KeyGenerator and SecretKey.

The usage of the classes in the JCA and JCE are virtually identical. In order to create an instance of a class, one does not use the new operator as is normally true with objects. 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. Here is an example of such usage:

KeyPairGenerator kpg =
KeyPairGenerator.getInstance ("RSA", "Crypto Provider Name");

This is done so that it is easier to change which algorithm someone might want to use (the first parameter) by simply altering a string, rather than having to change a line of code with a new statement. Also notice the second parameter above, which is the provider name. Every algorithm must be associated with a provider and multiple providers can support any single given algorithm. 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. The default provider shipped with the JCA and JCE is Sun's provider, java.security.provider.Sun.

It is unfortunate, however, that Sun's JCE does not support as many algorithms that could be useful for us as we would like. Therefore, the JCE that we have used is one provided by Bouncy Castle and can be found for download at www.bouncycastle.org, (file name: bcprov-jdk14-123.jar) at the time of writing[35]. We found that this version of the JCE is the most complete available that is free for download (open source) and is the easiest for implementing our own provider. In order to gain access to Bouncy Castle's JCE: If you are running a Windows machine with Borland's JBuilder 9, there are two directories that needed to be accessed. The first is the Java Runtime Environment (JRE) in the JBuilder directory, which is:

C:\JBuilder9\jdk1.4\jre\lib(\ext)

This represents the Software Development Kit (SDK) for the machine. (If you are using purely a command line approach, the SDK can be found in C:\jdk1.4.2) We needed to add the Bouncy Castle file, bcprov-jdk14-123.jar in both locations.

The same process is to be repeated for the Java 2 Run-time environment, which is located at:

C:\Program Files\Java\JRE\1.4\lib(\ext).

There is one more step to complete in order to be able to use the Bouncy Castle JCE with provider. This involves configuring the service provider preferences.

Configuring the service provider preferences enables the client access the service(s) by registering the provider and defining default preferences where more than one provider is registered for the same service algorithm. There are two types of provider registration: Static registration and Dynamic registration. Static registration consists of editing the java.security file (located in lib\security subdirectory of the Java Runtime Environment (JRE) to add a provider name to the list of approved providers. For each available provider for a given algorithm, there is a corresponding line in the java.security file with form:

security.provider.<n>=<providerClassName>

Where, n is the preference number of the provider. For example, the line:

security.provider.2=org.bouncycastle.jce.provider.BouncyCastleProvider

registers our provider with an order of preference 2. This process gave us access to a full JCE, which we could easily extend as our own provider. The Dynamic registration can be done by a client application upon requesting service(s) from a provider. The client application calls a class method, such as:

Security.addProvider (Provider providerName)

A JAVA IMPLEMENTATION OF RSA ALGORITHM

So where does the Java programming language come into the RSA picture? It comes in through the java.security package. Using this package, you can generate the pair of keys for the RSA algorithm. You do this by first creating an instance of a KeyPairGenerator. This is accomplished by invoking the static method getInstance of the KeyPairGenerator class. Here, we use a plain vanilla RSA algorithm with no mode or padding scheme and initializing it with the desired key size in number of bits; however, in real application padding etc. must be implemented for secure data protection. Next, you invoke the generateKeyPair() method to generate the RSA key pair:

```
//Create an RSA key pair
    KeyPairGenerator KPG =
    KeyPairGenerator.getInstance("RSA");
     SecureRandom r = new SecureRandom();
```

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. The Cipher class manipulates public key algorithms using keys produced by the KeyPairGenerator class.

We can create the key pair (KP) using the generateKeyPair() method. The key size is set to 1024-bits:

```
    KeyPairGenerator.initialize(1024, r);
    //Random generate 1024-bit key
    KeyPair KP = KPG.generateKeyPair();
    //Generates the key
```

The private and the public keys can be retrieved using the getPrivateKey() and the getPublicKey() methods of the KeyPair class, respectively:

```
PrivateKey priv = KP.getPrivate();
PublicKey publ = KP.getPublic();
```

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. You can simply generate the key pair with one line of code. However, if you need to generate more than one key pair, you can reuse the KeyPairGenerator object. This gives you much better performance than using a new KeyPairGenerator object every time. Listing 1 shows the complete implementation of key pair generation.

Listing 1: AsymmetricKeyMaker source code for generating key pair.

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

public class AsymmetricKeyMaker {

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

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

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

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

}
```

The String is not case sensitive and so RSA could be entered as rsa, Rsa, or any other variation. You compile the program using:

```
javac AsymmetricKeyMaker.java
```

If you run the program like this:

java AsymmetricKeyMaker RSA

Output that looks like shown in Listing 2.

Listing 2: Generated RSA Cryptographic key pair parameters.

```
SunJSSE RSA public key:
    public exponent:
        010001
modulus:
        b24a9b5b    ba01c0cd    65096370    0b5a1b92    08f8555e    7c1b5017    ec444c58    422b4109
        59f2e15d    43714d92    031db66c    7f5d48cd    17ecd74c    39b17be2    bf9677be    d0a0f02d
        6b24aa14    ba827910    9b166847    8154a2fa    919e0a2a    53a6e79e    7d2933d8    05fc023f
        bdc76eed    aa306c5f    52ed3565    4b0ec8a7    12105637    af11fa21    0e99fffa    8c658e6d

SunJSSE RSA private CRT key:
private exponent:
        78417240    9059965d    f3843d99    d94e51c2    52628dd2    490b731e    6fb2317c    66451e7c
        dc3ac25f    519a1ea4    198df4f9    817ebe17    f7c73c00    a1f96082    348f9cfd    0b63421b
        7f45f131    c363475c    c1b25f57    ee029f5e    0848ba74    ba81b730    ac4c0135    ce46478c
        e462361a    650e3356    f9b7a0c4    b682557d    3655c052    5e3554bd    970100bf    10dc1b51
modulus:
        b24a9b5b    ba01c0cd    65096370    0b5a1b92    08f8555e    7c1b5017    ec444c58    422b4109
        59f2e15d    43714d92    031db66c    7f5d48cd    17ecd74c    39b17be2    bf9677be    d0a0f02d
        6b24aa14    ba827910    9b166847    8154a2fa    919e0a2a    53a6e79e    7d2933d8    05fc023f
        bdc76eed    aa306c5f    52ed3565    4b0ec8a7    12105637    af11fa21    0e99fffa    8c658e6d
        public exponent:
        010001
prime p:
        e768033e    21646824    7bd031a0    a2d9876d    79818f8f    2d7a952e    559fd786    2993bd04
        7e4fdb56    f175d04b    003ae026    f6ab9e0b    2af4a8d7    ffbe01eb    9b81c75f    0273e12b
prime q:
        c53d78ab    e6ab3e29    fd98d0a4    3e58ee48    45a366ac    e94dbd60    ea24ffed    0c67c5fd
        3628ea74    88d1d1ad    58d7f067    20c1e3b3    db52adf3    c421d88c    4c4127db    d03592c7
prime exponent p:
        e09942b4    76029755    f9da3ba0    d70edcf4    337fbdcf    d0eb6e89    f74f5a07    7ca94947
        6835a805    3dfd047b    17310dc8    a39834a0    504400f1    0ce6e5c4    413df83d    4e0b1cdb
prime exponent q:
        829b8afd    a1984168    c2d1df4e    f32e2653    5b31b17a    cc5ebb09    a2e26f4a    040def90
        15be104a    ac92ebda    72db4308    b72b4ce1    bb58cb71    80adbcdc    625e3ecb    92daf6df
crt coefficient:
        4d8190c5    7730b729    00a8f1b4    ae526300    b22d3e7d    d64df98a    c1b19889    5240141b
        0e618ff4    be597979    95195c51    0866c142    30b37a86    9f3ef519    a3ae6469    14075097
```

If you are new to Java programming, this output highlights the value of properly overloading the toString() method. The text that begins SunJSSE RSA public key: is the result of calling the toString() method in the class RSAPublicKey. It contains the value of the public exponent which was referred to as e and the modulus N. The text that begins SunJSSE RSA private key: is the result of calling the same method in the class RSAPrivateKey. It contains the private exponent d and the modulus N. It also contains the public exponent e and the generating primes as well as some other data. If you are encrypting data you should be careful about how you send the public key values. 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 for encryption? In the case of RSA, you begin by taking some text message that you want to encrypt and turn it into a number M that is less than the modulus, M. However, it's easy to see the importance of choosing large primes. That's because the product of the primes determines the size of what can be encrypted. For example and for simplicity suppose the message M = 4356. Then you encrypt it by calculating C = M^e(mod N). In other words, you raise the number that you are encrypting to the power of your public exponent and take the remainder when dividing by the modulus, N. In this example, C = 4356^5(mod 430561) = 225382, so you send the encrypted message C = 225382.

Decryption requires you to repeat the process with the private-key, d, i.e., take C^d(mod N). In this example, calculate 225382^171693. Then take its remainder when dividing by 430561. You get back the original message of M = 4356. This is, of course, no accident and will always happen. The result follows from an application of Fermat's little theorem, which results in M^(e d) = M (mod N)[1]. For security reasons, each party encrypts with the other party's public-key. However, in real practice the encryption and encryption is performed as follows:

RSA Encryption: To encrypt a message X using an RSA public-key (N, e), one first formats the bit-string X to obtain an integer M in Z_N = {0, 1, 2, ... N-1}. This formatting is often done using public-key crypto system i.e., PKCS#1 standard[39]. The ciphertext is then computed as C←M^e (mod N).

RSA Decryption: To decrypt a ciphertext C the decrypter uses its private-key d to compute an e'th root of C by computing M←C^d(mod N). Since both d and N are large numbers (each approximately n-bits long) this is a lengthy computation for the decrypter. The formatting operation from the encryption algorithm is then reversed to obtain the original bit-string X from M. Note that d must be a large number (on the order of N) since otherwise the RSA system is insecure[15].

It is standard practice to employ the Chinese Remainder Theorem (CRT) for RSA decryption[1]. Rather than compute M←C^d(mod N), one evaluates:

$$M_p = C_p^{d_p} \pmod{p} \qquad M_q = C_q^{d_q} \pmod{q}$$

Here $d_p = d \pmod{p-1}$ and $d_p = d \pmod{q-1}$. Then one uses the CRT to evaluate M from M_p and M_q. This is approximately four times as fast as evaluating, C^d(mod N), directly[40].

A JAVA IMPLEMENTATION OF GENERATING A RSA CIPHER

We generate a cipher in much the same way that we generate a key pair. We must invoke the static method getInstance in the Cipher class. The parameters for this method work exactly as with KeyPairGenerator:

```
//Create a cipher using public-key to initialize it
 Cipher rsaCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
                    //Creates a Cipher object and specifying the algorithm, mode and padding
```

Listing 3 shows a complete implementation of RSA crypto-algorithm. The code is well commented to allow for ease of understanding the implementation procedure.

Listing 3: RSA Crypto-implementation (FileEncryptorRSA.java)

```
package com.rsc.rsafilecrypto;

import java.security.*;
import java.security.spec.*;
```

```
import javax.crypto.*;
import javax.crypto.spec.*;
import java.io.*;
import java.util.*;

/**
 *    This class encrypts and decrypts a file using CipherStreams
 *    and a 256-bit Rijndael key (AES). The key is then encrypted using
 *    a 1024-bit RSA key, which is password-encrypted.
 */
public class FileEncryptorRSA {
 /**
 *    When files are encrypted, this will be appended to the end
 *    of the filename.
 */
 private static final String ENCRYPTED_FILENAME_SUFFIX=".encrypted";

 /**
 *    When files are decrypted, this will be appended to the end
 *    of the filename.
 */
 private static final String DECRYPTED_FILENAME_SUFFIX=".decrypted";

 /**
 *    Number of times the password will be hashed with MD5
 *    when transforming it into a TripleDES key.
 */
 private static final int ITERATIONS = 1000;

 /**
 *    FileEncryptor is started with one of three options:
 *
 *    -c: create key pair and write it to 2 files
 *    -e: encrypt a file, given as an argument
 *    -d: decrypt a file, given as an argument
 */
 public static void main (String[] args)
 throws Exception {
 if ((args.length < 1) || (args.length > 2)) {
 usage();
 } else if ("-c".equals(args[0])) {
 createKey();
 } else if ("-e".equals(args[0])) {
 encrypt(args[1]);
 } else if ("-d".equals(args[0])) {
 decrypt(args[1]);
 } else {
 usage();
 }
 }

 private static void usage() {
```

```
System.err.println("Usage: java FileEncryptor-c|-e|-d [filename]");
System.exit(1);
}

/**
*    Creates a 1024 bit RSA key and stores it to
*    the filesystem as two files.
*/
private static void createKey()
throws Exception {
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Password to encrypt the private key: ");
String password = in.readLine();
System.out.println("Generating an RSA keypair...");

// Create an RSA key
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
keyPairGenerator.initialize(1024);
KeyPair keyPair = keyPairGenerator.genKeyPair();

System.out.println("Done generating the keypair.\n");

// Now we need to write the public key out to a file
System.out.print("Public key filename: ");
String publicKeyFilename = in.readLine();

// Get the encoded form of the public key so we can
// use it again in the future. This is X.509 by default.
byte[] publicKeyBytes = keyPair.getPublic().getEncoded();

// Write the encoded public key out to the filesystem
FileOutputStream fos = new FileOutputStream(publicKeyFilename);
fos.write(publicKeyBytes);
fos.close();

// Now we need to do the same thing with the private key,
// but we need to password encrypt it as well.
System.out.print("Private key filename: ");
String privateKeyFilename = in.readLine();

// Get the encoded form. This is PKCS#8 by default.
byte[] privateKeyBytes = keyPair.getPrivate().getEncoded();

// Here we actually encrypt the private key
byte[] encryptedPrivateKeyBytes =
passwordEncrypt(password.toCharArray(),privateKeyBytes);

fos = new FileOutputStream(privateKeyFilename);
fos.write(encryptedPrivateKeyBytes);
fos.close();
}
```

```
/**
*    Encrypt the given file with a session key encrypted with an
*    RSA public key which will be read in from the filesystem.
*/
private static void encrypt(String fileInput)
throws Exception {

BufferedReader in = new BufferedReader
(new InputStreamReader(System.in));
System.out.print("Public Key to encrypt with: ");
String publicKeyFilename = in.readLine();

// Load the public key bytes
FileInputStream fis = new FileInputStream(publicKeyFilename);
ByteArrayOutputStream baos = new ByteArrayOutputStream();

int theByte = 0;
while ((theByte = fis.read()) !=-1)
{
baos.write(theByte);
}
fis.close();

byte[] keyBytes = baos.toByteArray();
baos.close();

// Turn the encoded key into a real RSA public key.
// Public keys are encoded in X.509.
X509EncodedKeySpec keySpec = new X509EncodedKeySpec(keyBytes);
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
PublicKey publicKey = keyFactory.generatePublic(keySpec);

// Open up an output file for the output of the encryption
String fileOutput = fileInput + ENCRYPTED_FILENAME_SUFFIX;
DataOutputStream output = new DataOutputStream
(new FileOutputStream(fileOutput));

// Create a cipher using that key to initialize it
Cipher rsaCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
rsaCipher.init(Cipher.ENCRYPT_MODE, publicKey);

// Now create a new 256 bit Rijndael key to encrypt the file itself.
// This will be the session key.
KeyGenerator rijndaelKeyGenerator = KeyGenerator.getInstance("Rijndael");
rijndaelKeyGenerator.init(256);
System.out.println("Generating session key...");
Key rijndaelKey = rijndaelKeyGenerator.generateKey();
System.out.println("Done generating key.");

// Encrypt the Rijndael key with the RSA cipher
// and write it to the beginning of the file.
byte[] encodedKeyBytes= rsaCipher.doFinal(rijndaelKey.getEncoded());
```

```
output.writeInt(encodedKeyBytes.length);
output.write(encodedKeyBytes);

// Now we need an Initialization Vector for the symmetric cipher in CBC mode
SecureRandom random = new SecureRandom();
byte[] iv = new byte[16];
random.nextBytes(iv);

// Write the IV out to the file.
output.write(iv);
IvParameterSpec spec = new IvParameterSpec(iv);

// Create the cipher for encrypting the file itself.
Cipher symmetricCipher = Cipher.getInstance("Rijndael/CBC/PKCS5Padding");
symmetricCipher.init(Cipher.ENCRYPT_MODE, rijndaelKey, spec);

CipherOutputStream cos = new CipherOutputStream(output, symmetricCipher);

System.out.println("Encrypting the file...");

FileInputStream input = new FileInputStream(fileInput);

theByte = 0;
while ((theByte = input.read()) !=-1)
{
cos.write(theByte);
}
input.close();
cos.close();
System.out.println("File encrypted.");
return;
}

/**
*   Decrypt the given file.
*   Start by getting the RSA private key
*   and decrypting the session key embedded
*   in the file. Then decrypt the file with
*   that session key.
*/
private static void decrypt(String fileInput)
throws Exception {

BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Private Key to decrypt with: ");
String privateKeyFilename = in.readLine();

System.out.print("Password for the private key: ");
String password = in.readLine();

// Load the private key bytes
FileInputStream fis = new FileInputStream(privateKeyFilename);
```

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();

int theByte = 0;
while ((theByte = fis.read()) !=-1)
{
baos.write(theByte);
}
fis.close();

byte[] keyBytes = baos.toByteArray();
baos.close();

keyBytes = passwordDecrypt(password.toCharArray(), keyBytes);

// Turn the encoded key into a real RSA private key.
// Private keys are encoded in PKCS#8.
PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec(keyBytes);
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
PrivateKey privateKey = keyFactory.generatePrivate(keySpec);

// Create a cipher using that key to initialize it
Cipher rsaCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");

// Read in the encrypted bytes of the session key
DataInputStream dis = new DataInputStream(new FileInputStream(fileInput));
byte[] encryptedKeyBytes = new byte[dis.readInt()];
dis.readFully(encryptedKeyBytes);

// Decrypt the session key bytes.
rsaCipher.init(Cipher.DECRYPT_MODE, privateKey);
byte[] rijndaelKeyBytes = rsaCipher.doFinal(encryptedKeyBytes);

// Transform the key bytes into an actual key.
SecretKey rijndaelKey = new SecretKeySpec(rijndaelKeyBytes, "Rijndael");

// Read in the Initialization Vector from the file.
byte[] iv = new byte[16];
dis.read(iv);
IvParameterSpec spec = new IvParameterSpec(iv);

Cipher cipher = Cipher.getInstance("Rijndael/CBC/PKCS5Padding");
cipher.init(Cipher.DECRYPT_MODE, rijndaelKey, spec);
CipherInputStream cis = new CipherInputStream(dis, cipher);

System.out.println("Decrypting the file...");
FileOutputStream fos = new FileOutputStream(fileInput + DECRYPTED_FILENAME_SUFFIX);

// Read through the file, decrypting each byte.
theByte = 0;
while ((theByte = cis.read()) !=-1)
{
fos.write(theByte);
```

```
}
cis.close();
fos.close();
System.out.println("Done.");
return;
}

/**
*    Utility method to encrypt a byte array with a given password.
*    Salt will be the first 8 bytes of the byte array returned.
*/
private static byte[] passwordEncrypt(char[] password, byte[] plaintext) throws Exception {

// Create the salt.
byte[] salt = new byte[8];
Random random = new Random();
random.nextBytes(salt);

// Create a PBE key and cipher.
PBEKeySpec keySpec = new PBEKeySpec(password);
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBEWithSHAAndTwofish-CBC");
SecretKey key = keyFactory.generateSecret(keySpec);
PBEParameterSpec paramSpec = new PBEParameterSpec(salt, ITERATIONS);
Cipher cipher = Cipher.getInstance("PBEWithSHAAndTwofish-CBC");
cipher.init(Cipher.ENCRYPT_MODE, key, paramSpec);

// Encrypt the array
byte[] ciphertext = cipher.doFinal(plaintext);

// Write out the salt, then the ciphertext and return it.
ByteArrayOutputStream baos = new ByteArrayOutputStream();
baos.write(salt);
baos.write(ciphertext);
return baos.toByteArray();
}

/**
*    Utility method to decrypt a byte array with a given password.
*    Salt will be the first 8 bytes in the array passed in.
*/
private static byte[] passwordDecrypt(char[] password, byte[] ciphertext) throws Exception {

// Read in the salt.
byte[] salt = new byte[8];
ByteArrayInputStream bais = new ByteArrayInputStream(ciphertext);
bais.read(salt,0,8);

// The remaining bytes are the actual ciphertext.
byte[] remainingCiphertext = new byte[ciphertext.length-8];
bais.read(remainingCiphertext,0,ciphertext.length-8);

// Create a PBE cipher to decrypt the byte array.
```

```
PBEKeySpec keySpec = new PBEKeySpec(password);
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBEWithSHAAndTwofish-CBC");
SecretKey key = keyFactory.generateSecret(keySpec);
PBEParameterSpec paramSpec = new PBEParameterSpec(salt, ITERATIONS);
Cipher cipher = Cipher.getInstance("PBEWithSHAAndTwofish-CBC");

// Perform the actual decryption.
cipher.init(Cipher.DECRYPT_MODE, key, paramSpec);
return cipher.doFinal(remainingCiphertext);
}
}
```

Some of the required tools for implementing secure RSA algorithm: Hash Values:-Hash algorithms are one-way mathematical algorithms that take an arbitrary length input and produce a fixed length output string[7,41]. A hash value is a unique and extremely compact numerical representation of a piece of data. MD5 produces 128-bits for instance[7,42]. While SHA-1 produces 160-bits for instance[7,43]. 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[7]. However, one should be warned that SHA-1 was recently found to be insecure by the research team of Xiaoyun Wang, Yiqun Lisa Yin and Hongbo Yu (mostly from Shandong University in China) who showed a result, which describes collisions in the full SHA-1 in 2^{69} hash operations, much less than the brute-force attack of 2^{80} operations based on the hash length. This pretty much puts a bullet into SHA-1 as a hash function for digital signatures (although it doesn't affect applications such as HMAC where collisions aren't important), for update on this[44].

Random Number requirements truerand: Let us first understand what we mean by a true random number generator (TRNG); 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, in real practice the generation of random bits appears to be a difficult task. Consequently, pseudorandom bit generators (PRNG) often have to be used in many applications. They are not generators of truly random bits since they are deterministic but, starting with a random seed, they are able to generate sequences of bits that are random looking.

For example, a physical random generator is based on a physical noise source (primary noise) and a cryptographic or mathematical post-treatment of the primary noise. The primary noise must be subjected to an adapted statistical test on a regular basis[45] (Statistical random number generator tests). Annex D[46] for more detailed information on generating random numbers. The expected effort of guessing a cryptographic key shall be at least equivalent to guessing a random value that is EntropyBits long.

In practice, however, perfect random sources are not available, so the idea is to use a source of bits that may not be perfectly random and then to hash the bits in order to obtain really random bits. In this process, we assume that a hash function, such as SHA-1, is able to extract the randomness from a biased bit string. Formally, the amount of randomness of such a string is measured by its entropy. The notion of entropy has to be used very carefully since it applies to probability distributions and not to actual bit strings. 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, for detailed implementation[7].

In the implementation of RSA, the class java.security.SecurityRandom 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);
```

Why Pad Messages?: If you are familiar with cryptographic algorithms, you know that message block padding is quite common. 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[39], which is the subject of the next section.

PKCS1v1.5 Padding: This is the first type of padding and is definitely the simpler and more widely used of the two types mentioned. However, it is also less secure than OAEP, as it is more susceptible to attacks because it tends to repeat bytes of data that are in patterns. The encode method takes two parameters, the message to pad (which is a byte array) and the desired length of the padded message. A padded message of this type must be ten bytes longer than the message by itself. The first byte of the padded message must be a 2, which denotes that this block is an encrypted block. Immediately following this byte is a sequence of non-zero random bytes that must be at least eight bytes long. Then there must be a 0, which signifies the end of the padding and the beginning of the message itself. It is easy to parse such a padded message because the first zero byte will always be the delimiter between the padding and message. The role of the desired length is to find the length of the array of non-zero bytes. The algorithm takes the desired length and subtracts the length of the message, then subtracts two to account for the extra two bytes we must add. Decoding will search through the array to find the first zero byte and then throw away the padding to get the original data. In overall, this padding method is rather simple in comparison to OAEP.

Stream ciphers: Finally, 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, for example 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 different keys.

RSA ENCRYPTION AND DECRYPTION IMPLEMENTATION

Encrypting and decrypting data: Encryption works at the byte level, so almost anything can be encrypted. Once you have a key and a cipher, you're ready to go. It should be noted that the same algorithm must be used for both the key and cipher. You cannot have a key initialized with DES and a cipher initialized with RSA. The Cipher object uses the same methods to encrypt and decrypt data, so you must initialize it first to let it know what you want done with the data:

```
rsaCipher.init(Cipher.ENCRYPT_MODE, publicKey);
//Initializes the Cipher object.
```

This call initializes the Cipher object and gets it ready to encrypt data. The simplest way to encrypt data is invoking the doFinal method on the Cipher object passing in a byte array:

```
byte[] data = Hello World! .getBytes();//Calculates the
ciphertext with a plaintext string.
byte[] result = cipher.doFinal(data);
```

The result will now contain the encrypted representation of the passed-in data. It's just as easy to decrypt the same data. But before we can do that, we must reinitialize the Cipher object and get it ready for decryption:

```
// Perform the actual decryption.
        rsaCipher.init(Cipher.DECRYPT_MODE,
key.getPrivate()); //Decrypts the ciphertext
```

Once we do this, we are ready to decrypt:

```
byte[] plaintext = cipher.doFinal(result);
```

Now, plaintext should now be identical to data.

```
init(Cipher.DECRYPT_MODE, key.getPrivate());
doFinal(cipherText); //Decrypts the ciphertext
```

DIGITAL SIGNATURE AND AUTHENTICATION

The problem of authentication is perhaps an even more serious barrier to the universal adoption of telecommunications for business transactions than the problem of key distribution. Authentication is at the heart of any systems involving contracts and billing which is the cornerstone of e-commerce. Without it, business cannot function. Hence, the need for secure design and implementation of crypto-signature algorithm and certificate authority for secure data communication and authentication.

Therefore, if electronic data transactions are to replace the existing study document 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 message. The significance of a digital signature is comparable to the significance of a handwritten signature. 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 signature provides a high degree of nonrepudiation-that is, 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, for detailed mathematical implementation and analysis[1,7].

RSA Signature Algorithm: For a crypto-signature scheme, the main idea is no longer to disguise what a message says, but rather to prove that it originates with a particular sender[7,46,47]. A particular attractive feature of RSA crypto-algorithm is that there is a natural way to digitally sign messages. Suppose that Bob sends a message to Alice and wants to append a short signature S_B to his message that will convince Alice that it was really Bob who sent the message and that the message she received was not altered during transmission. (However, for secure implementation of RSA signature algorithm one requires prior implementation of message digest or hash function[7], followed by the intended signature scheme.) Therefore, his whole message M might be long, consisting of a large number of message units. Bob's first step is to apply a publicly hash function to M. That a function $M \rightarrow h(M) = H$, where, H is no longer than a single message unit. The function must be easy to compute and must satisfy two properties: (i) it must be computationally infeasible to find two different messages with the same hash value and, (ii) given H, it must be computationally infeasible to find any message with same hash value H.

The RSA, for example, has a simple way of doing this. If the owner of a code (N, e_B) wants to prove that he's the sender of a message H, he can use his private decoding exponent d_B on H to compute: $C = H^{d_B} \pmod N$ and then send both H and C, i.e., (C,H). The receiver can then persuade herself that the message truly originated with the owner of dB by computing: $C^{e_B} \bmod N$ and checking that it's the same as H.

From the previous example, the message H = 4356. Digital signature algorithm with RSA requires you to initially perform decryption using your private-key, $C = H\hat{}d_B \pmod N = 4356\hat{}171693 \pmod{430561} = 59978$. Then you send the deciphered message C = 59978. The receiver then performs the encryption process by calculating. $M = C\hat{}e_B \pmod N = 59978\hat{}5 \pmod{430561} = 4356$[1].

A Java implementation of RSA signature algorithm: The package java.security provides APIs for the message and digital signatures. The package java.security also offers DigestInputStream and DigestOutputStream classes for reading and writing to I/O. The signature class provides applications with functionality of the signature algorithms, such SHA-1/DSA, MD2/RSA, MD5/RSA, or SHA-1/RSA, while the SignatureSpi class defines the SPIs for the signature[7,46,47].

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(RSA),
- initialize(1024, r) and
- generateKeyPair() //Generates the keys.
- Signature.getInstance(MD5withDSA) //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 illustrates the implementation generating key pair, signing a file and then verifying the signature.

Listing 4: Implementation of sign file and then verify the signature (SignVerifyFile.java)

```
package com.rsc.rsadigisignverify;

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

class SignVerifyFile
{
 public static void main(String arg[])
 {
 if (arg.length != 3)
 System.out.println(Usage: java SignVerifyFile 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("RSA");
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();
```

```
// We intialize the signature
Signature rsasig = Signature.getInstance("SHA1withRSA");
rsasig.initSign(priv);

// We get the file to be signed
//FileInputStream fis = new FileInputStream(arg[0]);
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);
rsasig.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 = rsasig.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();

X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encKey);
KeyFactory KeyFac = KeyFactory.getInstance("RSA");
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
//Signature rsasigv = Signature.getInstance("SHA1withRSA");
rsasig.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);
rsasig.update(buff1, 0, len1);
}
buf.close();
fis.close();

// Verify the signature
boolean verifies = rsasig.verify(sigToVerify);
if (verifies)
System.out.println("Verified: The signature on the file is correct.");
else
System.out.println("Warning: The signature on the file has been tampered with.");
}

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

The comments embedded in the code 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 RSA public-key using KeyFactory class and then generate the key from it:

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

X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encKey);
KeyFactory KeyFac = KeyFactory.getInstance("RSA");
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.

Notice that the names of the three files used in this program should be passed by the user on the command line. 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 SignVerifyFile.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 SignVerifyFile DataFile SignatureFile PublicKeyFile

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

Verified: The signature on the file is correct

Step II-tamper any of the three: In this step we rerun the program but this time we subsequently tamper with any of the three files. To test this, this time we comment out the section implementing, write the signature to a file and write the public-key to a file, since we have already done so in Step 1, 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 from the SignatureFile and use the existing public-key from the PublicKeyFile 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: The signature on the file is correct.

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

Warning: The signature on the file has been tampered with.

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 demonstrates how you can successfully use the Java 2 APIs to send documents with proof of data integrity and authenticity.

Trusted Third Party: Certificate Authority (CA): Verification of public keys is an important step. Failure to verify that the public key really does belong to entity B (Bob), for example, leaves open the possibility that entity A (Alice) is using a key whose associated private key is in the hands of an enemy, say entity E (Eve). Public Key Infrastructures or PKI's deal with this problem by providing certification from the Certificate Authority (CA) that sign keys by a supposedly trusted party and make them available for download or verification.

Therefore, for an added level of transaction security, a certificate authority (CA) can be used. A CA is a third party that is trusted to perform the service of validating information about each user and creating signed certificates to that effect. A certificate represents the public-key identity of an entity. That is, it is a signed document from the CA that says: I certify that the public key in this document belong to the entity named in this document; signed (CA). It contains a packet of information, which includes the users (e.g., Bank's) public key, email address, name, address and other useful information, such as expiration date of the certificate and user privileges. A CA creates, distributes, revokes and generally manages these certificates. For, example, Alice wants to obtain the Bank's public key; she retrieves its certificate from the public key server directory and verifies the CA's signature on the certificate itself. Provided this signature verifies correctly, she has the CA's assurance that Bank's identity, its public key and all other information in the certificate is correct. Alice can now go ahead and use Bank's public key to encrypt confidential information transaction to send to the Bank or to verify the Bank's signature, protected by the assurance of the certificate. Well-known CAs include Verisign, Entrust and Thawte. The Certificates used with SSL today are X.509 certificates.

SECURE SOCKET LAYER (SSL) OPERATION

Overview: Secure Socket Layer (SSL)[48] is the most widely deployed and used crypto-security protocol on the

Internet today. SSL offers encryption, source authentication and integrity protection for data exchanged over insecure, public networks. The protocol has withstood years of scrutiny by the crypto-security community and is now trusted to secure virtually all sensitive web-based application ranging from online banking and stock trading to e-commerce transactions such as credit card payments. The SSL is used to assure an individual user (the client) of the authenticity of the web site (the server) he/she is visiting and, to establish a secure communications channel for the remainder of the session. Web pages that have addresses that start with https are protected with SSL while those that simply start with http are unprotected.

Among the features of SSL that have made it the de-facto standard vehicle for e-commerce transactions is its support for negotiable encryption and authentication algorithms. The designers of SSL realized that not all parties will use the same client software and consequently not all clients will include any particular encryption algorithm. The same is true for servers. The client and server at the two ends of a connection negotiate the encryption and decryption algorithms (cipher suites) during initial handshake. It may turn out that they do not have sufficient algorithms in common, in which case the connection attempt will fail.

Secure socket layer operates above a reliable transport service like TCP (Transmission Control Protocol)[49] and has the flexibility to accommodate different cryptographic algorithms for key agreement, encryption and hashing. However, the specification does recommend particular combinations of these algorithms, called cipher suites, which have well-understood security properties. For example, a cipher suite such as RSA-RC4-SHA would indicate RSA as the key exchange mechanism, RC4 for bulk (block cipher) encryption and SHA for hashing[7].

When a client first visits a secure web page (e.g., https://www.nsa.gov), the server transmits its certificate to the client. The certificate has two components: a data part containing the server's identifying information and RSA public-key and; a signature part which is the RSA signature of a certifying authority (CA) that vouches for the data part. In the circumstances, it is assumed that the CA has carefully verified the server's identifies before issuing the certificate. Upon receipt of the certificate, the client verifies the signature using the CA's public-key, which is pre-installed in the browser. A successful verification confirms the authenticity of the server and of its RSA public key. Although, the SSL does have client-to-server authentication capability, however, in real practice this is seldom used because it is difficult to implement a system to certify the public keys of individual users on a large scale. Clients are customarily authenticated in the application layer, through the use of passwords sent over an SSL-protected channel. This pattern is common in banking, stock trading and other secure Web applications. A closed padlock icon in Internet Explorer or Netscape browsers indicates the establishment of a secure link. Clicking on the icon reveals the server's certificate and information about the CA. If you accept the certificate you will be able to see the page behind the secure connection and future access to the same Web site will not cause the browser to issue a security alert. Note that when you accept the certificate, it is only for that session. In other words, once you completely exit the browser it is forgotten. However, both Netscape and Internet Explorer allow you to install a certificate permanently.

The two main components of SSL are the Handshake protocol and the Record Layer protocol. The Handshake protocol allows the SSL client and server to negotiate a common cipher suite, authenticate each other and establish a shared master secret using public-key cryptographic algorithms. Client authentication is optional. Only the server is typically authenticated at the SSL layer and client authentication is achieved at the application layer, e.g., through the use of passwords sent over an SSL-protected channel. However, some deployment scenarios do require stronger client authentication through certificates. The Record Layer derives symmetric-keys from the master secret and uses them with faster symmetric-key algorithms for bulk encryption and authentication of application data.

Public-key cryptographic operations are the most computationally expensive portion of SSL processing. SSL allows the re-use of a previously established master secrete resulting in an abbreviated handshake that does not involve any public-key cryptography and requires fewer and shorter messages. However, a client and server must perform a full handshake on their first interaction. Moreover, practical issues such as server load, limited session cache and naïve load balancers can adversely impact the ability to use an abbreviated handshake. Therefore, speeding up the public-key operations in SSL still remains a very active area for research and development.

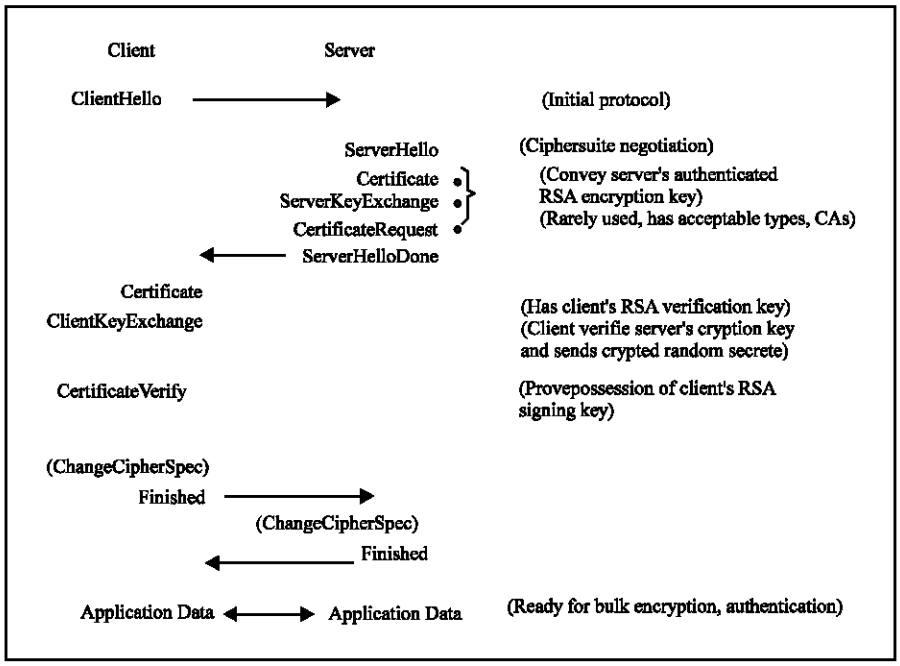

Fig. 4: RSA-based SSL Handshake

RSA-based Handshake: Today, the most commonly used public-key cryptosystem for master-key establishment is RSA. Figure 4 shows the operation of an RSA-based SSL handshake. In general, the client selects a random session key, encrypts it with the server's RSA public key and transmits the resulting ciphertext to the server. The server decrypts the session key, which is then used with a symmetric-key cryptosystem to encrypt and authenticate all sensitive data exchanged for the remainder of the session.

In this type of SSL handshake, the client and server first exchange random nonces (used for reply protection) and negotiate a cipher suite with ClientHello and ServerHello messages. The server then sends its signed RSA public-key either in the ServerCertificate message or the ServerKeyExchange message. To verify the server's RSA public-key, the client performs an RSA public-key operation. Then the client generates a 48-byte random number (the pre-master secret), encrypts it with server's public-key (an RSA public-key operation) and sends it in the ClientKeyExchange message. The server uses its RSA private-key to decrypt the premaster secret. Both end-points then use the premaster secret to create a master secret which, along with previously exchanged nonces, is used to derive the cipher keys, initialization vectors and MAC (Message Authentication Code) keys for bulk encryption by the Record Layer[7] .

The server can optionally request client authentication at the SSL layer by sending a CertificateRequest message listing acceptable certificate types and certificate authorities. In this case, besides performing the operations as described above, the client sends its RSA public-key in a ClientCertificate and proves processing of the corresponding private-key by including a digital signature in the CertificateVerify message. Producing this signature requires the client to perform an RSA private-key operation.

CONCLUSIONS

To-date RSA is the most popular cryptosystems for securing of electronic commerce transactions on the web. Therefore, the design and implementation of the crypto-web security is paramount in the Internet world. We have shown that JCE is a powerful API, allowing numerous types of encryption, as well as several other security-related tasks. We have also seen how to install the JCE both statically and dynamically, as well as use a asymmetric RSA algorithm to perform key pair generation, signature algorithm and encrypt and decrypt a simple message. We have shown schematically how to implement RSA based secure socket layer (SSL). We have also shown that ability to factor the RSA modulus will render the RSA crypto-algorithm useless. Therefore, care

must be taken in choosing the crypto-keylengths for securing data now and over coming years. Finally, be warned that the use of strong crypto-algorithm is prohibited in most countries! So check out jurisdiction you live in before venturing into crypto-world.

REFERENCES

1. Kefa Rabah, 2005. Data Security and Cryptographic Techniques: A review PJIT., 3: 106-132.
2. Rabah, K., 2005. Theory and Implementation of Data Encryption Standard: A review. In Press.
3. Rivest, R., A. Shamir and L.M. Adleman, 1983. Cryptographic Communications Systems and Method. U.S. Patent 4,404,829,20 Sept.
4. Diffie, W. and M.E. Hellman, 1966. New Directions in Cryptography. IEEE Transaction on Inform. Theory, IT-22: 644-654.
5. 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, Jan.
6. FIPS 186, National Institute of Standards and Technology, 1993. Digital Signature Standard. FIPS PUB 186.
7. Rabah, K., 2005. Secure Implementing Message Digest. Authentication and Digital Signature. (In Press).
8. ElGamal, T., 1985. A public-key cryptosystem and a signature scheme based on discrete logarithms. IEEE Trans. Info. Theory, 31: 469-472.
9. Koblitz, N., 1987. Elliptic curve cryptosystems, Mathematics of Computation, 48: 203-209.
10. Miller, S., 1986. Use of elliptic curves in cryptography. In CRYPTO '85, pp: 417-426.
11. Rabah, K., 2005. Theory and Implementation of Elliptic Curve Cryptography. J. Applied Sci., 5: 604-633.
12. Merkle, R.C., 1978. Secure Communication Over Insecure Channels. Communications of the ACM., 21: 294-299.
13. Zimmermann, P., 1992. PGP User's Guide. 4 Dec 1992.
14. Agrawal, M., N. Kayal and N. Saxena, 2005. Preprint. Primes is in P. http://www.cse.iitk.ac.in/primality.pdf.
15. Boneh, D. and G. Durfee, 1999. Cryptanalysis of RSA with private key less than $N^{0.292}$, Proc. Eurocrypt '99, LNCS, J. Stern (ed.), Springer-Verlag, 1999. Final version in IEEE Trans. Information Theory, Vol. 46, 2000.
16. Durfee, G. and P. Nguyen, 1990. Cryptanalysis of RSA with short private exponent, Proc. Asiacrypt '99, LNCS, Springer-Verlag, 1999.
17. Wiener, M., 1990. Cryptanalysis of short RSA secret exponents, IEEE Trans. Inform. Theory, 36: 553-558.
18. Håstad, J., 1998. Solving simultaneous modular equations of low degree, SIAM J. Computing, 17: 336-341.
19. Cryptographic Key lengths: http://www.distributed.net/.
20. ANSI key schemes FIPS 140-2. http://cio.doe.gov/Publications/profile2000/Profile2000_AppendixA.htm
21. Boneh, D., 1999. Twenty Years of Attacks on the RSA Cryptosystem. Notes of the American Mathematical Society, 46: 203-213.
22. Knuth, D.E., 1969. The Art of Computer Programming. Vol. 2, Seminumerical Algorithms Addison-Wesley, Reading, Mass.
23. Lenstra, H.W. Jr., 1987. Factoring Integers with Elliptic Curves. Ann. Math. 126, 649-673.
24. Pollard, J., 1993. Factoring with cubic integers, The Development of the Number Field Sieve', Lecture Notes in Mathematics, 1554: 4-10.
25. Leutwyler, K., 1994. Superhack: Forty Quadrillion Years Early, a 129-Digit Code Is Broken. Sci. Am., 271: 17-20.
26. Cavie, J., C., B. Dodson, R.-Marije Elkenbracht-Huizing, A.K. Lenstra, P.L. Montgomery and J. Zayer, 1996. A world wide number field sieve factoring record: on to 512 bits, in: Kwangjo Kim and Tsutomu Matsumoto (editors), Advances in Cryptology-Asiacrypt '96, Lecture Notes in Computer Science # 1163, Springer-Verlag, Berlin, pp: 382-394. http://ftp.cwi.nl/herman/ NFSrecords/RSA-155.
27. Bundesamt für Sicherheit in der Informationstechnik. Forscher stellen neuen Weltrekord auf: Zahl RSA200 zerlegt. May 9, 2005
28. A 248-digit factorization using SNFS, http://www.rkmath.rikkyo.ac.jp/~kida/snfs248e.htm.
29. Victor Miller's number theory mailing list archive, available at _ http://www.listserv.nodak.edu_
30. Shor, P.W., 1994. Algorithms for quantum computation: Discrete logarithms and factoring", in Proceedings of the 35th Annual Symposium on the Foundations of Computer Science, (Ed.) Goldwasser, S., (IEEE Computer Society Press, Los Alamitos, CA), pp: 124-134.
31. Shor, P.W., 1997. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM J. Computing, 26: 1484-1509.

32. Lenstra, A.K. and A. Shamir, Analysis and optimization of the TWINKLE factoring device, Advances in Cryptology, EUROCRYPT 2000, Lecture Notes in Computer Science, 1807, Springer-Verlag, pp: 35-52.
33. Bernstein, D., 2001. Circuits for integer factorization: a proposal, preprint.
34. Shamir, A. and E. Tromer, 2003. Factoring large numbers with the TWIRL device, Advances in Cryptology | CRYPTO 2003, Lecture Notes in Computer Science, Springer-Verlag, 2729: 1-26.
35. Sun JCE, http://java.sun.com/jce
36. IBM JCE, http://www7b.boulder.ibm.com/wsdd/wspvtindex.html
37. Bouncy Castle, www.bouncycastle.org
38. RSA JCE, http://www.rsasecurity.com/
39. RSA Labs. Public-key Cryptographic Standards (PKCS). No. 1, Ver. 2.0, Ver. 2.1.
40. Boneh, D. and H. Shacham, 2002. Fast Variants of RSA. RSA Laboratories, CryptoBytes, 5, no. 1, Winter/Spring 2002.
41. Damgård, T.B., 1990. A design principle for hash functions. In G. Brassard, (Ed.), Advances in cryptology. Proceedings of Crypto'89, vol. 435 of Lecture Notes in Computer Science. Springer-Verlag, New York, pp: 416-427.
42. R. Rivest, RFC 121: The MD5 Message-Digest Algorithm RSA Data Security, Inc., April 1992.
43. National Institute of Standards and Technology, FIPS PUB 180: Secure Hash Standard (SHS), May 11, 1993.
44. Schneier on Security, http://www.schneier.com/blog/archives/2005/02/sha1_broken.html
45. National Institute of Standards and Technology, NIST: FIPS Publication 186-2: Digital Signature Standard (DSS), January 2000.
46. 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.
47. Rivest, R., A. Shamir and L. Adleman, 1978. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. Comm. Of the ACM., 21: 120-126.
48. Frier, A., P. Karlton and P. Kocher, 2005. The SSL3.0 Protocol Version 3.0, http://home-netscape.com/eng/ssl3.
49. Rabah, K., 2004. Steganography-The Art of Hiding Data. ITJ., 3: 245-269.